Enhancing HPC Service Management on Alps using FirecREST API

Juan Pablo Dorsch
Swiss National Supercomputing Centre
ETH Zürich
Lugano, Switzerland
juanpablo.dorsch@cscs.ch

Eirini Koutsaniti
Swiss National Supercomputing Centre
ETH Zürich
Zurich, Switzerland
eirini.koutsaniti@cscs.ch

Andreas Fink
Swiss National Supercomputing Centre
ETH Zürich
Zurich, Switzerland
andreas.fink@cscs.ch

Rafael Sarmiento
Swiss National Supercomputing Centre
ETH Zürich
Zurich, Switzerland
rafael.sarmiento@cscs.ch

***Abstract*—With the evolution of scientific computational needs, there is a growing demand for enhanced resource access and sophisticated services beyond traditional HPC offerings. These demands encompass a wide array of services and use cases, from interactive computing platforms like JupyterHub to the integration of Continuous Integration (CI) pipelines with tools such as GitHub Actions and GitLab runners, and the automation of complex workflows in Machine Learning using AirFlow.**

This paper addresses the challenges faced by HPC centers providing multi-purpose HPC infrastructure in scaling these services to meet the diverse needs of various scientific communities. Our proposed solution involves the adoption of a web-facing RESTful API, such as FirecREST, to streamline access to HPC resources for those services. We methodically demonstrate, through various use cases, how FirecREST can significantly simplify the configuration of complex services and enhance the efficiency of HPC service management for HPC providers. This approach not only addresses the immediate needs of HPC centers but showcases the benefits of a web-facing interface to access HPC resources.

***Index Terms*—RESTful API, FirecREST, web-facing interface, Services, CI, Interactive computing, Regression Tests, Workflow engines**

I. Introduction

The advent of cloud technology has significantly influenced the expectations of the scientific community regarding the services and capabilities offered by HPC centers. There is an increasing demand for enhanced resource access and advanced services atop traditional HPC offerings. These requirements span a diverse range of applications, including interactive computing platforms like JupyterHub, the integration of Continuous Integration (CI) pipelines such as GitHub Actions and GitLab CI, and the automation of complex workflows, notably in Machine Learning (ML) using tools like AirFlow.

From the perspective of HPC centers, which provide multi-purpose HPC machines, accommodating these varied services for different scientific communities presents scalability challenges, particularly in terms of operational effort. This includes considerations related to security, maintenance, service management, and the allocation of human and infrastructure resources.

An example of such facilities providing multipurpose HPC infrastructure is the Swiss National Supercomputing Centre (CSCS), which serves a broad spectrum of scientific disciplines. CSCS provides the Alps infrastructure, a Cray HPE EX machines with heterogeneous hardware such as AMD and Nvidia CPUs and GPUs including Grace-Hopper superchips. One of the key requirements identified by CSCS is the facilitation of workflows for submitting computational tasks and managing data transfer in and out of the data center. To address this need, CSCS has developed a web-facing, RESTful API known as FirecREST [1], leading the path of the HPC community to provide RESTful access to HPC resources. This API is specifically designed to simplify resource access for various workflow engines. It offers a more straightforward and consistent approach compared to traditional SSH connectors, thereby simplifying the implementation of these engines for the scientific community. A notable example of a workflow engine using this API is the AiiDA workflow engine [2], developed by the Materials Science community, which incorporates a FirecREST backend to interact seamlessly with CSCS's infrastructure.

The introduction of a RESTful API like FirecREST extends beyond facilitating resource access to workflow engines. It opens up a plethora of possibilities for enhancing the efficiency and effectiveness of services by scientific communities to access HPC resources.

In this paper, we will explore a range of existing use cases from these communities, demonstrating how FirecREST has emerged as an effective tool in facilitating the deployment of various services and applications. This exploration aims to highlight the API's role in not only meeting but also advancing the evolving needs of HPC users in a dynamic scientific landscape.

II. FirecREST in a nutshell

FirecREST, a RESTful API designed for HPC, offers an HTTP-based interface to facilitate access to computational and data resources. Commonly, users leverage this API to develop web client applications that orchestrate automated workflows in an HPC environment.

The specification of FirecREST provides endpoints that abstract functionalities for submitting and querying jobs on the workload manager and scheduler (`/compute/jobs`), transferring data to and from the data center facility (`/storage/xfer-external/upload[|download]`), inspecting systems and file systems availability (`/status/systems`), and other file systems operations such as listing files (`/utilities/ls`), and creating directories (`/utilities/mkdir`), among others.

A. PyFirecREST

The functionality and user experience of FirecREST have been significantly augmented by the introduction of pyFirecREST [3]. This Python library streamlines the integration of REST API calls within Python classes, thereby enhancing the ease of use and efficiency in managing HPC workflows.

In Listing 1 we can outline a range of methods defined in pyFirecREST to enhance the integration of the REST API within a script.

```
# System availability
client.systems(system_name)

# Job submission
client.submit(system_name, script)

# Job querying
client.poll(system_name, jobid)

# Data transfer (upload)
client.external_upload(system_name, local_path, remote_path)

# Data transfer (download)
client.external_download(system_name, remote_path)

# List files and directories
client.ls(system_name, remote_path)
```

Listing 1: Most common pyFirecREST methods

Throughout this paper, we will notice the important role that pyFirecREST plays in terms of integration with services that expose their SDKs (Software Development Kits) and APIs using scripting languages.

B. Gateway

The utilization of the API through a web interface requires the presence of a dedicated gateway at HPC centers, tasked with managing web-based requests. This gateway serves as a singular access point for each system within the CSCS framework. To accurately direct requests to the intended system, they are uniquely identified by incorporating the machine's name within the request headers as described in Fig. 1, step (2).

Additionally, gateways provide features to prevent abuse of the resources behind the API using rate limiters per endpoint; and enhance access management and security with authorization and authentication plugins, monitoring tools, traffic management, and developer portals.

C. Authorization and authentication

Regarding identity and access management (IAM), FirecREST API employs the Open ID Connect (OIDC) - OAuth2 [4] protocol, utilizing an Identity Provider (IdP) for user authentication. In this framework, CSCS users are required to manage their API keys for accessing FirecREST as shown in Fig. 1, steps (0.a) and (0.b). This process involves authenticating through the IdP to obtain an access token (Fig. 1, step (1)), which is defined by a specific validity scope and application access permissions.

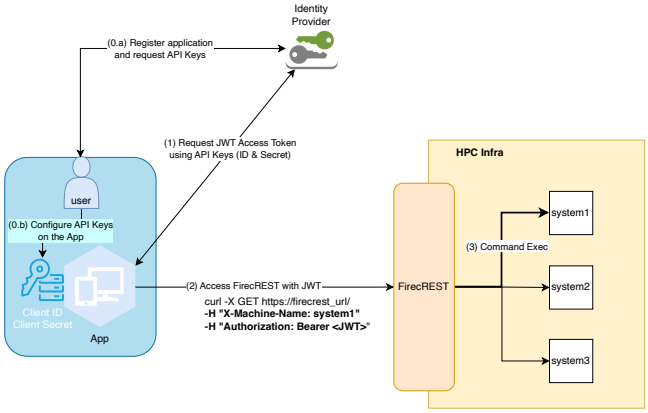

Fig. 1: IAM Workflow and Gateway request dispatching diagram from FirecREST to various systems.

The generation and renewal of access tokens can be efficiently automated using the IdP's API and the pyFirecREST library as shown in Listing 2. This automation facilitates the seamless integration of access token management within user applications, enhancing both the security and usability of the API.

Is important to mention that this specific OIDC workflow is called "client credentials" and is meant for machine-to-machine communication without human intervention when creating or refreshing the access token. This workflow enables the usage of FirecREST by web applications, pipelines, and scheduled tasks facilitating its integration on services operated in this fashion.

III. Use cases

In this study, we aim to comprehensively review a range of services, tools, and workflows that are commonly needed in HPC settings to support various scientific communities. Our focus is to examine how the implementation of an HPC-specific API, like FirecREST, can facilitate a smooth and efficient integration and management process, benefiting both end-users and system administrators. This exploration will

```python
# importing FirecREST module
import firecrest as f7t

# definition of IdP configuration
## API keys (usually stored in a secret engine):
CLIENT_ID     = "my_client_id"
CLIENT_SECRET = "my_client_secret"

## Access Token URL (provided by the IdP)
TOKEN_URI = "https://identity_provider/token/endpoint"

# creation of the authentication object
auth = f7t.ClientCredentialsAuth(CLIENT_ID, CLIENT_SECRET,
↪  TOKEN_URI)

# creation of the FirecREST Client Object
client = f7t.Firecrest("https://firecrest_url",
↪  authorization = auth)
...
```

Listing 2: FirecREST authorization and client definition

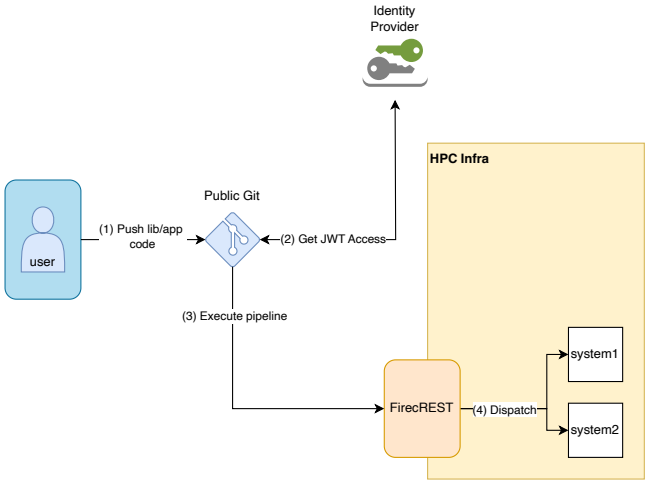

Fig. 2: CI pipelines using Public Git repository and FirecREST

highlight the API's role in streamlining operations, enhancing user experience, and optimizing system administration in HPC environments such as the one based on Cray HPE systems.

A. Continuous Integration Pipelines

Continuous Integration (CI) represents an emerging service tailored for the scientific community, designed to facilitate the testing of software releases across various programming environments and hardware systems. The integration of CI services within HPC centers empowers the community to conduct continuous testing following each code enhancement, thereby fostering a more sustainable software development ecosystem [6].

Establishing CI pipelines for the consistent testing and deployment of scientific software on HPC infrastructure presents several challenges for scientific software engineers. These challenges include:

- source code repository access which needs to be cloned into a node of the target machine, which means that SSH credentials (i.e., username/password, private/public key, or certificates) have to be handled inside the application, raising security concerns.
- the SSH session management to keep alive the connection during the pipeline execution.
- overall lack of interfaces from the SSH libraries to provide consistent outputs from diverse executed commands.

With FirecREST's assistance, we can tackle these challenges while also applying the same approach across various technologies such as GitLab CI, GitHub Actions, Jenkins CI, etc. Additionally, leveraging the HPC abstraction layer allows us to develop pipelines for diverse architectures and software stacks, facilitating portability and distribution.

As shown in Fig. 2, there is no requirement for a Git repository or a Runner to be installed on the HPC center to execute a pipeline. Instead, only access to the API from a public repository and a pair of API keys is necessary.

To exemplify the integration of a pipeline for HPC using FirecREST, we can showcase a GitHub Actions [5] workflow that submits a job through a step of the pipeline using a Python script as shown in Listing 3.

```python
# importing PyFirecREST
import firecrest as f7t

# Setup variables of the client
CLIENT_ID = os.environ.get("FIRECREST_CLIENT_ID")
CLIENT_SECRET = os.environ.get("FIRECREST_CLIENT_SECRET")
FIRECREST_URL = os.environ.get("FIRECREST_URL")
AUTH_TOKEN_URL = os.environ.get("AUTH_TOKEN_URL")

# Auth Object definition
idp = f7t.ClientCredentialsAuth(CLIENT_ID, CLIENT_SECRET,
↪  AUTH_TOKEN_URL)

# FirecREST client definition
client = f7t.Firecrest(firecrest_url=FIRECREST_URL,
↪  authorization=idp)

# Check System Status via pyFirecREST
system_state = client.system(system_name)

if system_state["status"] == "available":
    # Submit job via pyFirecREST
    job = client.submit(system_name, "submission_script.sh")
    print(f"Submitted job: {job['jobid']}")

    # Poll job status via pyFirecREST
    poll_result = client.poll(system_name,
    ↪  jobs=[job["jobid"]])
    if poll_result[0]["state"] != "COMPLETED":
        print(f"Job was not successful, status:
        ↪  {poll_result[0]['state']}")
        exit(1)

else:
    print(f"System {system_name} is not available")
    exit(1)
```

Listing 3: Script for testing job execution in a pipeline

In the named script, the intention is to assert that the pipeline will fail if the `system_name` is not available for utilization or if the job submitted fails in its execution on the workload scheduler. In any other case, if the job is completed successfully the pipeline will succeed.

Following the example, the pipeline will be executed each time a commit is pushed to the `main` branch. This can be seen on the GitHub workflow specification (Listing 3), where the pipeline is configured to install pyFirecREST, obtains the API keys from the secret store of GitHub, sets them as part of the environment of the runner, and runs the tests for a specific `system_name`.

```yaml
name: CI
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  test_mycluster:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        system_name: ["mycluster"]

    steps:
      - uses: actions/checkout@v3

      - name: setup python
        uses: actions/setup-python@v4
        with:
          python-version: '3.7'

      - name: install python packages
        run: |
          python -m pip install --upgrade pip
          pip install pyfirecrest==2.1.0

      - name: Run testing script
        env:
          FIRECREST_CLIENT_ID: ${{ secrets.F7T_CLIENT_ID }}
          FIRECREST_CLIENT_SECRET: ${{
          ↪  secrets.F7T_CLIENT_SECRET }}
          FIRECREST_URL: ${{ secrets.F7T_URL }}
          AUTH_TOKEN_URL: ${{ secrets.F7T_TOKEN_URL }}
        run: ci/ci_script.py \\
             --system=${{ matrix.system_name }} \\
             --branch=${{ github.ref_name }} \\
             --repo=${{ github.server_url }}/${{
             ↪  github.repository }}.git \\
             --account=ci_user
```

Listing 4: CI workflow specification for GitHub Actions

As an extension to this use case, we will delve into a complete feature developed by CSCS that allows users to utilize CSCS infrastructure to run integration tests of their scientific application hosted on public repositories. This is achieved through the utilization of a CI GitLab-Runner, which operates in conjunction with FirecREST as described in Figure 3.

This service facilitates the testing of scientific software

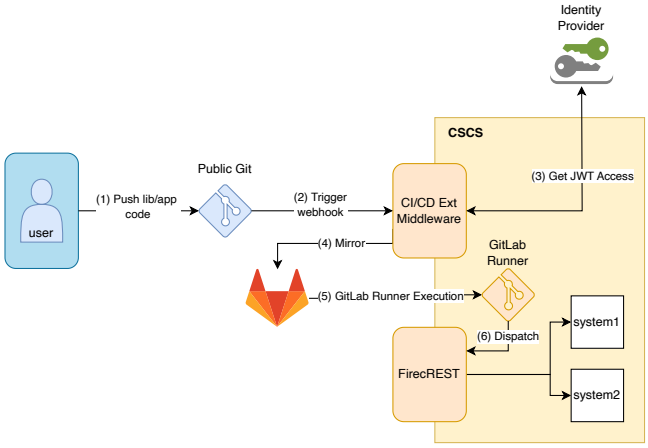

Fig. 3: CICD Ext service at CSCS

applications at CSCS by mirroring the public repository of the application onto a GitLab repository managed by CSCS.

Users set up a Git webhook on the CI/CD Ext Middleware platform exposed by CSCS. This setup automatically mirrors commits done on the application repository to the GitLab instance managed by CSCS and transparently executes tests, ensuring continuous integration with dependencies and libraries installed on CSCS systems.

The approach works as a spinoff of the CI pipeline execution showcased earlier, albeit with a centralized service overseen by CSCS. This centralized service enables users to bypass pipeline configuration, thus facilitating usage.

B. Interactive Computing

JupyterHub [7] serves as a multi-user hub designed to initiate, oversee, and proxy multiple Jupyter notebooks. Frequently employed in HPC centers, it enables the launching of notebooks directly from the browser onto compute nodes. Scientists use Jupyter to develop proof of concept code, explore datasets, or as an educational tool to teach their communities.

The integration with workload managers within JupyterHub is facilitated by the `batchspawner` package. This package implements Jupyter notebooks execution for various workload schedulers, such as Torque, and SLURM. When JupyterHub is used with `batchspawner`, they're typically deployed on a machine with a workload manager configured for the target system. Such a setup introduces significant maintenance overhead. Configuring a system to interface the SLURM controller requires system administrators to set up a scheduler daemon and share a key between a controller and daemons. This complicates the deployment of JupyterHub and restricts the systems on which it can operate.

To address this at CSCS, we've developed a spawner based on FirecREST which offers the same functionality as `batchspawner` but employs pyFirecREST as a backend. This approach significantly reduces the requirements for deploying JupyterHub on HPC systems since any system able

to connect to the web can target compute nodes through the FirecREST service.

In our FirecREST spawner, following the `batchspawner` implementation, we use the JupyterHub's `Spawner` class as base and reimplement the `start()`, `poll()`, and `stop()` methods to interact with the target HPC system via pyFirecREST (Listing 5).

```python
import firecrest as f7t
from jupyterhub.spawner import Spawner

class FirecRESTSpawnerBase(Spawner):
  # Start Jupyter notebook
  def start(self):
    self.job = client.submit(self.host, script_str=script)

  # Polling Jupyter notebook status
  def poll(self, jobid):
    self.job = client.poll(self.host,jobid)

  # Stop Jupyter notebook
  def stop(self, jobid):
    client.cancel(self.host, self.job_id)
```

Listing 5: FirecREST spawner definition for JupyterHub

An interesting outcome of this integration is that the same OIDC-OAuth2 client used for JupyterHub's Web UI can be used to authenticate with FirecREST to manage the notebook lifecycle. This way, user authentication for both services is significantly simplified. Additionally, this approach allows the creation of different instances of JupyterHub interfacing with different systems by simply changing the `X-Machine-Name` header to specify the target system, as described in Figure 4.

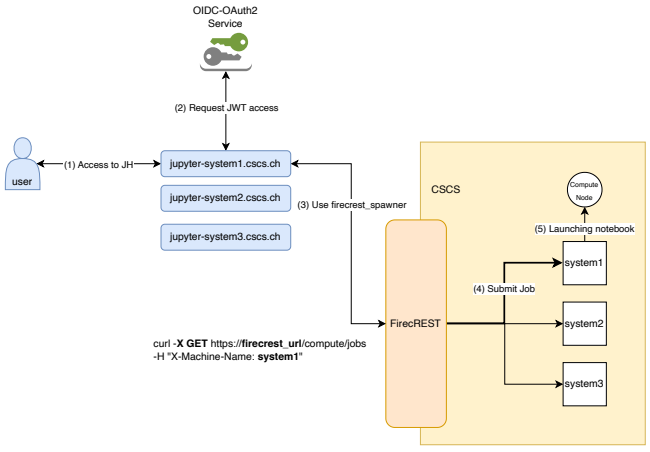

Fig. 4: Jupyter notebook execution using FirecREST

C. Regression Testing

ReFrame [8] is a framework designed for regression testing on HPC systems. Its primary function is to facilitate the periodic testing of key scientific applications, ensuring their operational integrity and consistent performance as a way to assess quality of services to the scientific communities. A typical scenario where ReFrame is employed involves verifying the optimal performance of scientific applications following maintenance, particularly when libraries undergo updates, in an HPC environment.

The installation and maintenance of this service need to be addressed by the administrators of the data center where the software is being tested. With FirecREST, we can broaden the utilization of ReFrame for testing and integration, allowing administrators, maintainers, developers, and users to carry out these tasks from an instance in a public cloud provider.

To understand how to do this, it should be noticed that the pipeline of ReFrame for each test goes through different stages: setup, compile, run, sanity, performance, and cleanup, and that ReFrame presents a Python class for execution on HPC facilities using the SLURM workload manager: the `SlurmJobScheduler` class.

By extending this class, we can register a new one for scheduling jobs with the help of pyFirecREST (`firecrest-scheduler`) and mapping the abstract classes `submit()`, `poll()`, `cancel()`, and `wait()` methods. This is shown in Listing 6.

```python
from reframe.core.schedulers.slurm import SlurmJobScheduler
import firecrest as f7t

@register_scheduler('firecrest-scheduler')
class FirecrestJobScheduler(SlurmJobScheduler):

    def __init__(self, *args, **kwargs):
      (...)
      # Setup the FirecREST Client
      self.client =
      ↪  f7t.Firecrest(firecrest_url=firecrest_url,
      ↪  authorization=f7t.ClientCredentialsAuth(CLIENT_ID,
      ↪  CLIENT_SECRET, TOKEN_URL))

    def submit(self, job):
        # Job Submission
        submission_result =
        ↪  self.client.submit(self._system_name,
        ↪  os.path.join(job._remotedir,
        ↪  job.script_filename) )

    def poll(self, *jobs):
        # Update the status of the jobs
        poll_results = self.client.poll(
            self._system_name, [job.jobid for job in jobs]
        )

    def cancel(self, job):
        # Cancel a job
        self.client.cancel(job.system_name, job.jobid)
        job._is_cancelling = True
```

Listing 6: FirecREST scheduler definition for ReFrame

This scheduler must be set on the site configuration of ReFrame along with the target system and environment to test. This setup enables its on-demand execution or through a scheduled pipeline for periodic testing.

Utilizing ReFrame in conjunction with FirecREST offers a key advantage: users are not required to log in to the HPC

system to perform Quality of Service (QoS) tests. Instead, this process can be initiated from any machine with access to FirecREST. This design ensures flexibility and independence for users, allowing them to seamlessly execute QoS tests without being tethered to the HPC system itself.

D. Workflow orchestrator for Machine Learning

Apache Airflow [9], a popular workflow orchestration tool, offers a robust framework for defining, scheduling, and monitoring diverse workflows in particular in the Machine Learning (ML) domain. However, integrating Airflow with HPC workload managers, such as SLURM, presents certain challenges.

A primary obstacle is the absence of seamless integration between Apache Airflow and workload managers, which rely on custom commands and APIs for job submission and monitoring. Consequently, to interact with the HPC system, Apache Airflow must either run on a machine where the workload manager is installed and configured, or Airflow tasks must establish remote login connections to such a machine.

The flexibility of Airflow's operator API makes it particularly convenient for integration with pyFirecREST, enabling the definition of operators, such as file transfers to and from HPC systems as well as job submission (Listing 7) and monitoring. These operators can then be used to construct Airflow Directed Acyclic Graphs (DAGs) (Listing 8), streamlining the orchestration of complex workflows.

In this DAG, when the `FileSensor` task indicates that a new file has been created by an external process, it will trigger the DAG execution by uploading a couple of input files, submitting a job, and downloading the output on the job finishes. Finally, some post-processing is done.

The outcome of the DAG execution can be seen graphically on the AirFlow instance (Fig. 5). The screenshot shows how the usage of the RESTAPI provides basic operators that can be arranged in different ways to compose the DAGs which are executed via a web application without the need to provide a workflow orchestrator at the HPC premises.

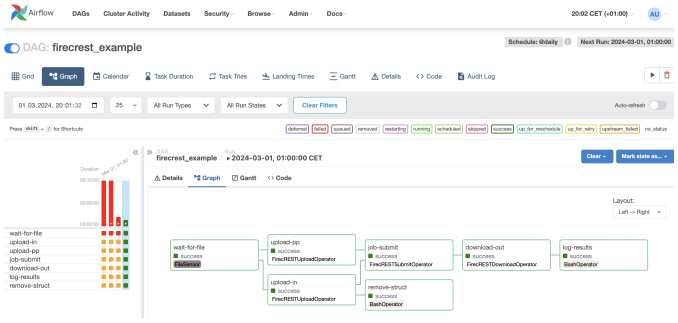

Fig. 5: AirFlow using FirecREST operators

The availability of such pyFirecREST operators will open new possibilities for enhancing the scaling and performance of Airflow DAGs, especially in the machine learning domain, where the tool has been widely adopted, as it often involves complex and resource-intensive computations that demand efficient orchestration and coordination and requires access to large-scale HPC systems like Cray EX with Alps as an example.

```python
import firecrest as f7t
from airflow.models.baseoperator import BaseOperator
from airflow import AirflowException

# setting up the FirecREST Base Operator for AirFlow

class FirecRESTBaseOperator(BaseOperator):
    (...)
    #  FirecREST client object
    client = f7t.Firecrest(firecrest_url=firecrest_url,
    ↪  authorization = f7t.ClientCredentialsAuth(CLIENT_ID,
    ↪  CLIENT_SECRET, TOKEN_URL))

class FirecRESTSubmitOperator(FirecRESTBaseOperator):
    """Airflow Operator to submit a job via FirecREST"""

    def __init__(self, system: str, script: str, **kwargs)
    ↪  -> None:
        super().__init__(**kwargs)
        self.system = system
        self.script = script

    def execute(self, context):
        (...)
        while True:
            if self.client.poll_active(self.system,
            ↪  [job['jobid']]) == []:
                break
            time.sleep(10)
        job_info = self.client.poll(self.system,
        ↪  [job['jobid']])
        if job_info[0]['state'] != 'COMPLETED':
            raise AirflowException(f"Job state:
            ↪  {job_info[0]['state']}")
        return job
```

Listing 7: FirecREST Submit Operator definition for Apache Airflow

```python
from airflow import DAG

from airflow.operators.bash import BashOperator
from airflow.sensors.filesystem import FileSensor

from firecrest_airflow_operators import
↪  (FirecRESTSubmitOperator, FirecRESTUploadOperator,
↪  FirecRESTDownloadOperator)

with DAG( dag_id="firecrest_example",
↪  tags=["firecrest-executor"]) as dag:

    wait_for_file = FileSensor( task_id="wait-for-file",...)
    upload_in =
    ↪  FirecRESTUploadOperator(task_id="upload-in",...)
    upload_pp =
    ↪  FirecRESTUploadOperator(task_id="upload-pp",...)
    submit_task =
    ↪  FirecRESTSubmitOperator(task_id="job-submit",...)
    download_task =
    ↪  FirecRESTDownloadOperator(task_id="download-out",...)
    log_results = BashOperator(task_id="log-results",...)
    remove_struct =
    ↪  BashOperator(task_id="remove-struct",...)
```

Listing 8: Definition of the DAG in Fig 5

IV. Conclusion

In this paper, the benefits of utilizing REST API in HPC environments have been demonstrated concerning the installation, customization, and maintenance of user-interfacing services and tools for various types of communities.

It is important to mention the wide range of use cases that can be covered by this approach, including many not addressed in this work, such as the creation of scientific portals through web applications, AiiDA workflow engine, and scientific data collaboration platforms, to name a few.

It must be acknowledged that the use of pyFirecREST, its ability to automate authentication management through OIDC/OAuth2, and its ease of use with APIs and SDKs exposed by the services to be connected have been paramount for such integrations.

We wish to emphasize the improvement that this approach introduces in the management of services by reducing the workload of the responsible staff while facilitating support and assistance to user's workflows.

This is especially important in architectures such as Alps Cray EX systems soon to be available for multiple user communities at CSCS. The blueprint of Alps includes different versatile HPC clusters ("vClusters") designed to address different use cases (machine learning, weather & climate, general HPC usage, etc), where service configuration and deployment on the top of multiple clusters are needed. API as a service becomes very important in these multi-tenancy environments to address user's needs and avoid management overhead.

Looking ahead, the use of APIs as a service layer providing uniformity in support and accessibility to HPC infrastructures could allow the scientific community to establish a standard for service management and workflow execution between and across different supercomputing centers and data infrastructure worldwide.

References

[1] Cruz F and Martinasso M, "FirecREST: RESTful API on Cray XC systems," CUG 2019 Proceedings, Montreal, Canada, May 5-9, 2019.

[2] Huber S.P., Zoupanos, S., Uhrin, M. et al. "AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance". Sci Data 7, 300 (2020). https://doi.org/10.1038/s41597-020-00638-4

[3] ETH-CSCS, "Welcome to PyFirecREST," pyfirecrest.readthedocs.io, published November 17, 2023. [Online]. Available: https://pyfirecrest.readthedocs.io/en/stable/. [Accessed: January 15, 2024].

[4] OKTA Developers, "Open ID Connect & OAuth 2.0 API,", okta.com, published January 5, 2024. [Online]. Available: https://developer.okta.com/docs/reference/api/oidc/. [Accessed: January 15, 2024].

[5] GitHub, "GitHub Actions documentation," github.com, published April 6, 2024. [Online]. Available: https://docs.github.com/en/actions. [Accessed: April 7, 2024].

[6] US Department of Energy, "IDEAS Productivity," https://ideas-productivity.org/index.html, published January 3, 2024. [Online] Available: https://ideas-productivity.org/index.html [Accessed: January 15, 2024].

[7] Jupyter Hub, "Project Jupyter,", jupyter.org, published January 7, 2024. [Online]. Available: https://jupyter.org/hub. [Accessed: January 15, 2024].

[8] Karakasis, V. et al. "Enabling Continuous Testing of HPC Systems Using ReFrame,". In: Juckeland, G., Chandrasekaran, S. (eds) Tools and Techniques for High Performance Computing. HUST SE-HER WIHPC 2019 2019 2019. Communications in Computer and Information Science, vol 1190. Springer, Cham.

[9] Apache Airflow, "What is Airflow,", apache.org, published January 10, 2024. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/index.html. [Accessed: January 15, 2024].