Secure Authorization for RESTful HPC Access

Mohammad Hossein Biniaz
Computing
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
E-Mail: mohammad-hossein.biniaz@gwdg.de

Sven Bingert
eScience
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
E-Mail: sven.bingert@gwdg.de

Christian Köhler
Computing
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
E-Mail: christian.koehler@gwdg.de

Hendrik Nolte
Computing
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
E-Mail: hendrik.nolte@gwdg.de

Julian Kunkel
Computing
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen/Universität Göttingen
Göttingen, Germany
E-Mail: julian.kunkel@gwdg.de

***Abstract*—The integration of external services, such as workflow management systems, with High-Performance Computing (HPC) systems and cloud resources requires flexible interaction methods that go beyond the classical remote interactive shell session. In a previous work, we proposed the architecture and prototypical implementation of an Application Programming Interface (API) which exposes a Representational State Transfer (REST) interface, which clients can use to manage their HPC environment, transfer data, as well as submit and track batch jobs. In the present article, we expand on this foundation by integrating a fine-grained role-based authorization and authentication system, which facilitates the initial setup and increases the user's control over the jobs that services intend to submit on their behalf. The developed *HPCSerA* service provides secure means across multiple sites and systems and can be utilized for one-off code execution and repetitive automated tasks.**

***Index Terms*—HPC, automation, RESTful API, OAuth, authorization, web interface.**

I. INTRODUCTION

Due to the increasing demand on computing power driven by the success of resource-intensive methods in various scientific domains, there is an equally increasing requirement by researchers to utilize HPC resources to satisfy their demand in a cost-effective manner. This has led to the creation of different services, which for instance expose a RESTful API, with which users can remotely interact with an HPC system. There are numerous different use cases for such a requirement. One motivating example can be the ability to manage complex and compute intensive workflows with a graphical user interface to improve usability for inexperienced users [1].

While, one one hand, there are these efforts to ease and open up the use of HPC systems, there is, on the other hand, a constant threat by hackers. Since users typically interact with the host operating system of an HPC system directly, local vulnerabilities can be immediately exploited. Two of the most favored attacks by outsiders are brute-force attacks against a password system [2] and probe-based login attacks [3]. These attacks, of course, become obsolete if attackers can find easier access to user credentials. Therefore, it is of utmost importance to keep access, and access credentials, to HPC systems safe.

In this context, services easing the use of and the access to HPC systems should be treated with caution. For example, if access via Secure Shell (SSH) [4] to an HPC system is only possible using *SSH keys* due to security concerns, these measures are rendered ineffective if users re-establish a password-based authentication mechanism by deploying a RESTful service on the HPC system that is exposed on the Internet. Observing these developments, it becomes obvious that there is a requirement to offer a RESTful service to manage data and processes on HPC systems remotely which is comfortable enough in its usage to discourage concocted and insecure solutions built by inexperienced users with the main objective of "getting it to work", but which adheres to the **highest security standards**.

The *key contributions* of this article are:

1) analysis of possible attack scenarios based on a RESTful service running on an HPC system;
2) presentation of a state-of-the-art REST API design, called *HPCSerA*, to secure the RESTful service;
3) discussion of the usability utilizing explicit use cases.

A *REST Service*, i.e., a web application, is typically deployed in a suitable cloud environment. User requests for code execution on the HPC system are generated manually or automatically and then sent by a *Client* to this *REST Service*. In order to execute the requested task, an *Agent* is deployed on an HPC system that retrieves the tasks and executes them, for instance by submitting a job on the cluster via the batch system.

The remainder of this paper is structured as follows: In Section II, the related work is presented, including state-of-the-art mechanisms to solve this issue. In Section III, existing security issues preventing a wide-spread application of *HPCSerA* are being discussed and an improved architecture

Copyright (c) IARIA, 2022. ISBN: 978-1-61208-961-4

with a security-based scope definition is presented. In the following Section IV, our implementation is presented. At the end, a diverse set of use cases are presented in Section V, as well as a concluding discussion, which is provided in Section VI.

II. Related Work

There is without question a general trend towards remote access for HPC systems, for instance in order to use web portals instead of terminals [5]. These applications actually have a long standing history with the first example of a web page remotely accessing an HPC system via a graphical user interface dating back to 1998 [6].

Newer approaches are the *NEWT* platform [7], which offers a RESTFul API in front of an HPC system and is designed to be extensible: It uses a pluggable authentication model, where different mechanisms like Open-Authorization (OAuth), Leightweight Directory Access Protocol (LDAP) or *Shibboleth* can be used. After authentication via the */auth* endpoint, a user gets a cookie which is then used for further access. With this mechanism *NEWT* forwards the security responsibility to external services and does not guarantee a secure deployment on its own. This has the disadvantage, that *NEWT* is not intrinsically safe, therefore providers of an HPC-system need to trust the provider of a *NEWT* service, that it is configured in a secure manner. Additionally, no security taxonomy is provided, which is key when balancing security concerns and usability.

Similarly, *FirecREST* [8] aims to provide a REST API interface for HPC systems. Here, the Identity and Access Management is outsourced as well, in this case to *Keycloak*, which offers different security measures. In order to grant access to the actual HPC resources after successful authentication and authorization, a *SSH certificate* is created and stored at a the *FirecREST* microservice. Although this is a sophisticated mechanism, there seem to be a few drawbacks. First of all, the *sshd* server must be accordingly configured to support this workflow, secondly it remains unclear how reliable status updates about the jobs can be continuously queried when using short-lived certificates, and lastly these certificates needs to be stored at a remote location, which might conflict with the terms of service of the data center of the user. Additionally, HPC systems are often configured to allow logins from a trusted network only, which means that the *FirecREST* microservice can not serve multiple HPC systems at a time.

While the *Slurm Workload Manager* provides a REST interface that exposes the cluster state and in particular allows the submission of batch jobs, the responsible daemon is explicitly designed to not be internet-facing [9] and instead is intended for integration with a trusted client. Its ability to generate JSON Web Token (JWT) tokens for authentication provides an interesting alternative route for interaction with our architecture, provided both services are hosted in conjunction. Clients that shall execute Slurm jobs authenticate the trusted Slurm controller via the *MUNGE* service [10] that relies on a shared secret between client and server. If either of these is compromised, then it is assumed that the whole cluster is insecure. Slurm can be deployed across multiple systems and administrative sites and there are various options for Slurm to support a meta-scheduling scenario or federation. However, if the Slurm controller is compromised, it can dispatch arbitrary jobs to any of the connected compute systems. In addition, decoupling the API implementation from the choice of the job scheduler, as we propose, allows interoperation of multiple sites, possibly using different schedulers.

An alternative execution model popular with public cloud systems is Function-as-a-Service (FaaS). In this model, a platform for execution of functions is provided, i.e., code can be submitted by the user and execution of the function with parameters are triggered via an exposed endpoint. A runtime system executes the function in an isolated container and automatically scales up the number of containers according to the response time and number of incoming requests. Customers are billed for the execution time of the function. The core assumption is that the function is a sensible unit of work, e.g., running for 100ms, running on a single core, side-effect free, and thus only suitable for embarrassingly parallel workloads. Authentication and security is of high importance for these systems as well. For example, OpenFaaS is a Kubernetes-based FaaS system that utilizes, e.g., OAuth to authorize users and to generate tokens that are verified upon function deployment or execution. While this mechanism has similarities to our approach, FaaS is for short-running (subsecond to several second) single node jobs, we provide different, security-derived authorization processes for the different available operations, while mitigating user impact via push notifications and solve the issue for long-running HPC systems including parallel jobs.

III. Architecture

We first analyze the potential security issues from our initial architecture and describe an approach to address them via an updated authorization and authentication process. Finally, each step of the revised workflow is discussed individually.

A. Problem statement

In the original architecture, static *bearer tokens* were used for user authentication. There was one *bearer token* per user, which means that each client, as well as each agent authenticated towards *HPCSerA* with the same token, compare [11, III. B.]. Although considered state-of-the-art, this approach has different security flaws, which prevented a public deployment. These security problems become apparent, when particularly taking into account that an access mechanism for an HPC system is provided. One problem is that this single *bearer token* can be used to access all endpoints, which means that it can be used to perform any possible operation. This can be maliciously exploited in two different ways:

- If that token is not properly guarded, an attacker can use it to post a malicious job, to gain direct access to the HPC system.

- If an attacker has escalated their privileges, the token used by the agent is left vulnerable. If the user has authorized that token to get access to more than one HPC system, the attacker has immediately gained access to another cluster.

There are two different conclusions one can deduce from these observations: First, it is a highly vulnerable step to allow code ingestion via a RESTful service into an HPC system and one has to take the chance of a token loss into account, when designing such a system. Second, the agent sometimes only needs the permissions to read queued jobs and to update the state of a job, e.g., from *queued* to *running*. It is, therefore, an unnecessary risk to allow a job ingress from the token of an agent.

B. Improved Architecture

The separation of access tokens by the user who created them and the services (clients and HPC agents) to which they are deployed, as described in [11], already enables revoking trust in a setup with multiple services and multiple backend HPC systems easily. However, during operation, there is global access to the entire state, i.e., in-flight jobs, to all parties involved. In order to segment trust between groups of services and HPC backends, our revised architecture (cf. Figure 1) resolves this issue by introducing a dedicated tag field into the design of the database for access tokens. Based on this information, client services and HPC agents can be authorized individually. Moreover, each token can be assigned one or multiple roles that restrict the combination of Hypertext Transfer Protocol (HTTP) endpoints and verbs which can be used for all entities that have been created using the same tag. The token's individual lifetime is implied by the granted role.

User control over each individual task and job that is allowed to be run or submitted, respectively, is enforced by introducing an intermediate authentication step that requires user interaction via an external application. This could be run on a mobile device or hardware token, like the ones being used for two-factor authentication or integrated into the web-based user interface used for token and device management for fast iterations on the workflow configuration. Metadata about the action to be authorized is included in the user prompt in order to allow an informed decision. However, the measure is restricted to this most critical step of the process, while non-critical endpoints, such as retrieving the state of pending jobs, can continue to respond immediately. For submitting a new job, the necessity of individual user confirmation is also determined by whether new code is ingested or an already existing job is merely triggered to run on new input data.

From the user's perspective, setting up the workflow would start with logging into the web interface and creating tokens for each service to be connected to the API and configuring them in each client and agent, respectively. In order to acquire a minimal working setup, at least one token for the client service and one for the agent communicating to the batch system on the HPC backend system would be required. OAuth compatible clients could initiate this step externally, thereby sidestepping the need for the user to manually transfer the token to each client configuration. As soon as each client has acquired the credentials either way, HPC jobs can be relayed between each service and the HPC agent.

While the OAuth 2.0 terminology [12] allows a distinction between an authorization server which is responsible for granting authorization and creating access tokens, and a resource server which represents control over the entities exposed by the API, in our case the tasks and batch jobs to be run, both roles are assumed by our architecture, so the design can be as simple as possible and deployed in a single step. However, since the endpoints for acquiring access tokens and the original endpoints that require these access tokens are distinct, a separation into microservices (which again need to be authenticated against each other) would also be compatible with the presented design.

The steps necessary for code execution are illustrated in Figure 1. As a preliminary, we assume that the HPC agent is set up and configured with the REST service as an endpoint. The arrows indicate the interactions and the initiator. The individual steps are as follows:

1) The workflow starts by a user logging into the web interface. The Single sign-on (SSO) authentication used for this purpose has to be trusted, since forging the user's identity could allow an attacker to subsequently authorize a malicious client to ingest arbitrary jobs.
2) The user can create tokens for the the REST service in the WebUI.
3) The tokens are stored in the Token database (DB), along with the granted role, project tag and token lifetime.
4) The retrieved tokens can then be used by a client, e.g., to run some code on the HPC system or have an automatic process in place, provided the code is already present on the system, rendering manual authentication unnecessary.
5) The request is forwarded to the REST Service, which verifies the information in the Token DB. On success, the code to execute is forwarded to the HPC agent.
6) If the client chooses to use the OAuth flow instead in order to avoid manual token creation, the authorization request is forwarded to the Auth app instead.
7) The user can choose to confirm or deny the authorization request. In the former case, the generated token is stored (cf. 3) in the Token DB. Again, further requests can then in general proceed via step 5 without further user interaction.
8) Like any other client, the HPC agent uses a predefined token or alternatively initiates the OAuth flow in order to get access to the submitted jobs.
9) For the most critical task of executing code on the HPC frontend or submitting batch jobs, the agent can be configured to get consent from the user by using the Auth app for authentication.
   This request is accompanied by metadata about the job to be executed, such as a hash of the job script, allowing an informed decision by the user. This step also avoids the need for trust in a shared infrastructure,

since the authentication part can be hosted by each site individually.

10) Once the user confirmed execution, the HPC agent executes the code, e.g., by submitting it via the batch system. In this case, information about the internal job status is reported back to HPCSerA.

We assume that the HPC agent is secure as otherwise the system and user account it runs on are compromised and, hence, could execute arbitrary code via the batch system anyway. The Web-based User Interface (WebUI), HPC agent, HPCSerA Service and Client are all independent components. For example, a compromised REST Service could try to provide arbitrary code to the HPC agent anytime or manipulate the user's instructions submitted via the client. However, as the user will be presented with the code via the authenticator app and can verify it similarly to a 2 Factor Authentication (2FA), the risk is minimized.

There are multiple approaches to deploy HPCSerA across multiple clusters and administrative domains:

a) Replication: Each center could deploy the whole HPCSerA infrastructure which we develop (cf. Figure 1) independently maximizing security and trust. By adjusting the endpoint URL, a user could connect via the identical client to either the REST service at one or another data center – this is identical to the URL endpoints in S3. Although the user now has two independent WebUIs for confirming code execution on the respective data center, the authenticator and the identity manager behind it could be shared. An additional advantage of this setup would be that the versions of HPCSerA deployed at each center could differ.

b) Shared infrastructure: The maximum shared configuration would be that for each HPC system a user has to deploy a dedicated HPC agent on an accessible node but all the other components are only deployed once.As the HPC agents register themselves with the REST service, now the user can decide at which center they would like to execute any submitted code. While using a single WebUI for many centers and cloud deployments maximizes usability, it requires the highest level of trust in the core infrastructure: If two of these components are compromised, arbitrary code can be executed on a large number of systems.

IV. Implementation

In the following, more details about the technologies chosen for our implementation are provided. Due to the conceptualized architecture in Section III, this section has a focus on the current scope definition and the authentication/authorization scheme employed. Generally, the OpenAPI 3.0 specification [13] was used to define the RESTful API, which is a language-agnostic API-first standard used for documenting and describing an API along with its endpoints, operations, request- and response-definitions as well as their security schemes and scopes for each endpoint in *YAML* format. This API is backed by a *FLASK*-based web application written in *Python*. The token database is in a SQL-compatible format, thus SQLite can be used for development and, e.g., PostgreSQL for the production deployment. The database schema contains only the user (`user_id`) and project (`project_id`) that the token belongs to as well as the individual permission-level (`token_scope`).

A. Definition of Access Roles

In order to give granular permissions for accessing each of the endpoints, OpenAPI 3.0 allows to define multiple security schemes providing different scopes to define a token matching to the security level of each of the endpoints. Eight different roles have been identified, which are listed and described in Table I.

These roles are entirely orthogonal, which means they can be combined as necessary. If, for instance, on one HPC system only parameterized jobs needs to be submitted, the *agent* can be provided with a token which has only the permissions of role 2 and 3, thus lacking role 5, which is required to fetch new files. Similarly, if a token is provided to a client which is not 100% trustworthy, one can choose to only provide a token with the role 6, i.e., to only allow to trigger a predefined job. Important to understand is the difference in mistrust between the role 3, 4, and 5. The security mistrust in role 4 comes from the admins of the *HPCSerA*, which want to ensure that a code ingestion is indeed done by the legitimate user. Therefore, in order to allow code ingestion, the possession of a token with the corresponding permission is not enough, the user has to confirm the code ingestion via a 2FA. The mistrust in role 3 and 5 comes, however, from the user, who wants to ensure that only jobs s/he confirmed are being executed. This is, again, completely orthogonal, to the enforced 2FA in role 4 and can be optionally used by the user. This fine-grained differentiation between the different security implications of the discussed endpoints, minimize user interference while providing a high level of trust.

B. Providing Tokens via Decoupled OAuth

The introduction of OAuth-compatible API endpoints has several advantages: Access tokens can be created on demand in a workflow initiated by a client or HPC agent, respectively. In addition, while there is a default API client provided, a standard-compliant API enables users to easily develop drop-in replacements.

It is important to note here that we modified the usual OAuth flow, where a client gets redirected to the corresponding login page to authorize the client. This "redirect approach" has two problems:

- The client is a weak link, where the Transport Layer Security (TLS) encryption is terminated and therefore becomes susceptible to attacks and manipulation.
- It does not support a headless application, like the HPC agent, which is not able to properly forward the redirect to the user.

Due to these shortcomings, a modified OAuth flow was developed to enable the usage of headless apps and improve security. This modified version decouples the user confirmation from the client, which means that the client is not

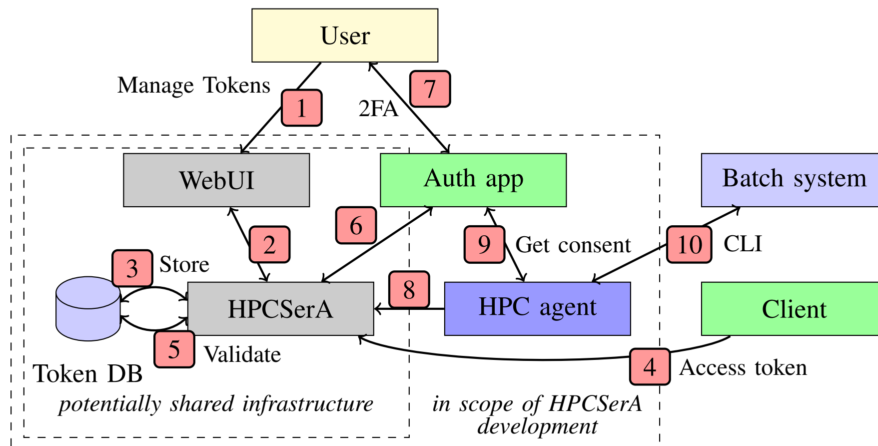

Fig. 1. A sketch of the proposed token-based authorization flow. The following parts are shown: 1) WebUI login 2) Connection to the HPCSerA service 3) Storage of access tokens 4) Client connecting to the API 5) Validation of access tokens 6) Authorization request 7) User interaction with the Auth app 8) HPC agent connecting to the API 9) Authentication request for code execution 10) Interaction with the HPC batch system

TABLE I
DEFINITION OF THE EIGHT ROLES. OPERATIONS MARKED IN RED HAVE TO BE CONSIDERED SECURITY CRITICAL FROM THE ADMIN POINT OF VIEW, WHEREAS THE ORANGE MARKED OPERATIONS FROM A USER POINT OF VIEW.

Role Number	Role	Description
1	GET_JobStatus	Client can retrieve information about a submitted job
2	UPDATE_JobStatus	Used by client/agent to update the job status
3	GET_Job	Endpoint used by the agent to retrieve job information
4	POST_Code	Client to ingest new code to the HPC sytsem
5	GET_Code	Agent pulls new code. Might be necessary to run new job
6	POST_Job	Client triggers parameterized job
7	UPDATE_Job	Client updates already triggered job
8	DELETE_Job	Client deletes already triggered job

being redirected but that the confirmation request is being sent out-of-band, e.g., via the WebUI or via notification on a smartphone device.

Starting with the case that the script does not already come equipped with a token, analogous to the usual OAuth flow, the generation of a token is requested. Since our use case was initially built as an instance of machine-to-machine interaction, i.e., headless, the issue of a lack of user interface is encountered; the usual OAuth flow - implemented in the browser - would redirect the user to an authorization server where the user could actively provide their username and password to the authorization server. The authorization server would then return a code, in the case of the authorization code flow, in the redirect URI which would be posted in a backchannel along with a client secret assigned at the time of registering the client to attain an access token.

In order to circumvent this headless-app problem, this work has implemented a synchronous push notification system analogous to the Google prompt where a notification is pushed to a user's device awaiting a confirmation to proceed. In the Minimum Viable Product (MVP), we have implemented this in the SSO-secured WebUI in order to have a more integrated interface. Eventually, the final product will see an Android and iOS app that receives such notifications. This flow then grants the permission to execute a security critical operation, compare Table I.

This confirmation via push notification cannot solely rely on time-synchronicity since it would be susceptible to an attacker requesting tokens and/or 2FA confirmation for carrying out a security-critical operation in the same approximate time frame. Therefore, a sender constraint has to be implemented. This is done in a similar way to the original authorization code flow: The access code is signed with a client secret, which was configured with *HPCSerA* prior to the execution of this workflow, and then sent to *HPCSerA*. *HPCSerA* verifies the secret and only then sends the actual token. This secret is implemented using public-private key pairs, where the public key is uploaded to *HPCSerA* in the initial setup to register a new client (or agent).

Alternatively, in the case that a token is supplied along with the software or script that is submitting a job to the HPCSerA API, the permissions are validated against a token database. In the case that the token provided contains permissions for accessing a sensitive endpoint, the second factor check is triggered through the WebUI and the notification / confirmation process is once again undergone. It is important to note that this is not a hindrance since already-running jobs and non-sensitive endpoints proceed without user-intervention.

V. USE-CASES

Due to the previously stated changes in the architecture, there are certain adaptions in the previously presented use

Copyright (c) IARIA, 2022.

cases [11]. These changes will be discussed in the following and serve as the basis for a broader user impact analysis.

A. GitLab CI/CD

Since the *GitLab* Runner can be configured to run arbitrary code without including secrets in the repository, thanks to GitLab's project Continuous Integration and Integration Development (CI/CD) variables [14], the required tokens can be made available to the CI/CD job so it can in turn access the API endpoints required to transmit the current repository state to an HPC system where the code can be tested using the HPC software environment or even multiple compute nodes.

A new commit might of course introduce arbitrary code to the HPC environment, therefore it is advisable to enforce the extra authentication step (cf. Section III-B), when code from a new commit is submitted to the HPC system. The corresponding hash, available by default via the `GIT_COMMIT_SHA` variable, would be a helpful piece of information to display to the user when asking to authorize the request.

B. Workflow Engine

In the workflow use case, HPC jobs should be fully automated without user interaction. Due to multiple repetitions and time dependencies, interactions severely limit the functionality and practicability of the workflow. One possibility is to prepare the workflow in such a way that only parameterized jobs are called and thus only safe endpoints of HPCSerA are used. Another possibility is to use dedicated (legacy) endpoints that are only accessible through firewall regulations and fixed network areas. The latter can also be regulated via an additional proxy server, such as a *nginx*.

C. Data Lake

In order to provide high performance computing capabilities to a data lake [15], *HPCSerA* is used to submit jobs on behalf of the data lake users. A user sends a so-called *Job Manifest* to the data lake, where the software, the compute command, the environment, and the input data are unambiguously specified. By transferring the responsibility of scheduling the job from the user to the data lake, it has the control about it. This allows to reliably capture the data lineage and to foster reproducibility. The added benefit of the newly implemented security measures in *HPCSerA* is that, before, users had to trust the data lake, and hereby the admins, with their *bearer tokens*. By introducing *OAuth* and enforcing a 2FA for code ingestion, this is not necessary anymore, since users now need to confirm each submission. Since users submit jobs actively, for instance via a *Jupyter Notebook* using a PythonSDK, the requirement to confirm each submission does interrupt the workflow too much.

VI. CONCLUSION

In the paper presented here, we have examined the issue of security in accessing HPC resources via a RESTful API. The initial situation with a very simplified token model does not meet the requirements. Therefore, a fine-granular token model, coupled with interactive user consent and OAuth flows, was proposed. With this new model, particularly critical interactions, such as code transfer, can be secured. User consent is requested in a prototype via a WebUI, which in turn uses a central Identity Management (IDM) for authentication. This means that no critical user-specific data needs to be managed.

In future work, the possibilities for obtaining user consent will be further analyzed. The development of mobile apps is planned, which will greatly simplify the consent workflow for the user. So far, the focus has been on the transmission and execution of code. However, there is also a requirement to transmit data objects that are necessary for execution. Therefore, it is examined to what extent the current implementation is suitable for such tasks and where possible limits are reached in terms of data quantity and transmission speed.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the "Niedersachsisches Vorab" funding line of the Volkswagen Foundation and "Nationales Hochleistungsrechnen" (NHR).

REFERENCES

[1] Z. Wang et al., "RS-YABI: A workflow system for Remote Sensing Processing in AusCover," in *Proceedings of the 19th International Congress on Modelling and Simulation*. MODSIM 2011 - 19th International Congress on Modelling and Simulation - Sustaining Our Future: Understanding and Living with Uncertainty, 2011, pp. 1167–1173.

[2] A. K. Singh and S. D. Sharma, "High Performance Computing (HPC) Data Center for Information as a Service (IaaS) Security Checklist: Cloud Data Governance." *Webology*, vol. 16, no. 2, pp. 83–96, 2019.

[3] J.-K. Lee, S.-J. Kim, and T. Hong, "Brute-force Attacks Analysis against SSH in HPC Multi-user Service Environment," *Indian Journal of Science and Technology*, vol. 9, no. 24, pp. 1–4, 2016.

[4] T. Ylonen, "SSH - Secure Login Connections Over the Internet," in *Proceedings of the 6th USENIX Security Symposium (USENIX Security 96)*. San Jose, CA: USENIX Association, Jul. 1996, pp. 37–42, [accessed: 2022-03-21]. [Online]. Available: https://www.usenix.org/conference/6th-usenix-security-symposium/ssh-secure-login-connections-over-internet

[5] P. Calegari, M. Levrier, and P. Balczyński, "Web portals for high-performance computing: a survey," *ACM Transactions on the Web (TWEB)*, vol. 13, no. 1, pp. 1–36, 2019.

[6] R. Menolascino *et al.*, "A realistic UMTS planning exercise," in *Proc. 3 ACTS Mobile Communications Summit 98*, 1998.

[7] S. Cholia and T. Sun, "The newt platform: an extensible plugin framework for creating restful hpc apis," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4304–4317, 2015.

[8] F. A. Cruz *et al.*, "FirecREST: a RESTful API to HPC systems," in *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*, 2020, pp. 21–26.

[9] SchedMD. (2022) Slurm REST API. [accessed: 2022-03-18]. [Online]. Available: https://slurm.schedmd.com/rest.html

[10] Chris Dunlap. (2022) MUNGE Uid 'N' Gid Emporium. [accessed: 2022-03-21]. [Online]. Available: https://dun.github.io/munge/

[11] S. Bingert, C. Köhler, H. Nolte, and W. Alamgir, "An API to Include HPC Resources in Workflow Systems," in *INFOCOMP 2021, The Eleventh International Conference on Advanced Communications and Computation*, C.-P. Rückemann, Ed., 2021, pp. 15–20.

[12] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012, [accessed: 2022-03-21]. [Online]. Available: https://www.rfc-editor.org/info/rfc6749

[13] OpenAPI Initiative. (2017) OpenAPI Specification v3.0.0. [accessed: 2022-03-21]. [Online]. Available: https://spec.openapis.org/oas/v3.0.0

[14] GitLab. (2022) GitLab CI/CD variables. [accessed: 2022-03-18]. [Online]. Available: https://docs.gitlab.com/ee/ci/variables/

[15] H. Nolte and P. Wieder, "Realising Data-Centric Scientific Workflows with Provenance-Capturing on Data Lakes," *Data Intelligence*, pp. 1–13, 03 2022. [Online]. Available: https://doi.org/10.1162/dint_a_00141

Copyright (c) IARIA, 2022.