Secure Authorization for RESTful HPC Access with FaaS Support

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

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

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 that clients can use to manage their HPC environment, transfer data, as well as submit and track batch jobs. In this article, we expand on this foundation by including a full Function as a Service (FaaS) interface which allows it to be a drop-in replacement for functions with high resource demands. In order to enable automated processes without any manual interaction while maintaining the highest security standards, 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 is presented. 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, while adhering to the highest security standards.**

***Keywords*—HPC; RESTful API; OAuth; authorization; FaaS.**

I. INTRODUCTION

This work is an extension of [1] by the inclusion of the *Function as a Service* (FaaS) idiom, which becomes increasingly popular due to several advantages, including the cost effectiveness, fault tolerance, and ease of use. However, there are usually strict limitations on the execution time of a function, resource requirements and the size of input and output data [2]. Driven by the large success of data- and compute-intensive methods, there is an increasing demand for computing power in various scientific domains which are outside of these limits. Historically, HPC systems were used to satisfy those requirements in a cost-effective manner. Meanwhile it is similarly attractive to use a RESTful interface to easily deploy preconfigured functions and use those in an automated setup. 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 [3].

While, on 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 or intruders. 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 [4] and probe-based login attacks [5]. 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) [6] 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 spontaneously 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**.

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

In order to prevent those security risks by users, HPC systems are increasingly secured, including a two-factor authentication (2FA) for SSH connections [7], which is a problem for automated workflows since they need to run without any manual interaction. In this paper we present *HPCSerA*, a REST API which is compatible to the FaaS idiom, therefore allowing clients to use it for large, data and compute-intensive functions similar to *OpenFaaS* [8]. In order to enable this functionality, a detailed security analysis was done and a secure authorization method was developed to enable automated data processing without any manual intervention, while maintaining a similar security standard compared to a SSH access secured by 2FA. The *key contributions* of this article are:

1) discussion of the FaaS usage model and its capabilities on HPC systems using HPCSerA;
2) analysis of possible attack scenarios based on a RESTful service running on an HPC system;
3) presentation of a user-friendly and secure authorization method inspired by OAuth;
4) discussion of the usability, including the resolution of complicated dependencies.

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. This is followed by a presentation of the fundamental idea of HPCSerA and its three basic components in Section III. Based on this, the FaaS functionality is discussed in Section IV. In Section V, existing security issues preventing a wide-spread application of HPCSerA are being discussed and an improved architecture with a security-based scope definition is presented. In the following Section VI, our implementation is presented. At the end, a diverse set of use cases are presented in Section VII, as well as a concluding discussion, which is provided in Section VIII.

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 [9]. 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 [10].

Newer approaches are the *NEWT* platform [11], 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), Lightweight 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* [12] 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, an *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. A similar approach is used by *HEAppE* [13] where the communication is between the API server and the HPC system is done via SSH. To do so, for each project an SSH key is managed by the API server. Users are not supposed to connect to the system via SSH at all. However, in order to upload data via secure copy users obtain a temporary SSH key. To manage the exposure of a possible data breach of the API server, the developers recommend to use one instance of HEAppE per HPC account.

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 [14] 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 [15] 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 the execution of functions is provided, i.e., code can be submitted by the user and execution of the function with parameters is 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. Cus-

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

tomers 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 seconds) 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. General Architecture

HPCSerA consists in total of three components which enable the access and remote control of an HPC system via a REST API.

These three components as well as their interactions are depicted in Figure 1: The main component is the API server, which at first glance looks like a simple message broker. Clients, shown on the left side in green, can use the REST API of the *API Server* to post a new HPC task. On the opposite side, there is a cronjob running, in the following called *Agent*, which periodically queries the API server for available tasks and pulls them if available. Once pulled, the agent will execute the task and will update the state of the task on the API server accordingly. This simple approach has several advantages:

- If the egress firewall rules allow access to the API server, which would be possible even for HPC systems which do not allow general internet access, the entire setup can be done in user space.
- The agent is independently configurable. This means that the agent does not require a fixed interface, like a certain resource manager, and can be customized to work with any kind of system.
- The agent can only do, what it is configured to do. Therefore, a user can configure what should be exposed. The highest form of exposure would be to allow arbitrary code ingest and execution, like sending a shell script and executing it. A smaller level of exposure would be to just allow the submission of preconfigured batch jobs to the resource manager.
- A user can hook an authorization mechanism into the agent in user space and therefore does not need to completely trust the administrators of the API server or HPCSerA. This mistrust allows a large exposure of the agent in a secure manner.

In the following the three components are presented in more detail.

A. The API Server

As a central component of the *HPCSerA architecture*, the *API server* handles HTTP connections from the *Client* and *Agent* (described below), maintains the internal state of all jobs and functions and resolves dependencies between functions. In addition it provides the necessary maintenance endpoints to allow configuration via the Web UI. It communicates with the database for persistence of the internal state and verification of any authentication tokens. Since every job has to be kept in the database until it is completed, the API server is not stateless. All other connections are initiated by other components, therefore the API server is the only part of the architecture that has to allow incoming connections. It is also the responsibility of the API server to ensure separation between jobs of different projects, i.e., these are only visible in response to requests which are authorized for the same project.

B. The Client

Any application or service that needs HPC as a backend implements the *Client* component, which initiates HTTP connections to the API server in order to submit jobs, call functions (cf. Section IV) and retrieve information on the job state. Examples of use cases for the Client are given in Section VII.

C. The Agent

On the HPC system the *Agent* component regularly connects to the API server in order to retrieve jobs that are ready for execution. Depending on the function being called, the batch system might be involved and is regularly queried on the state of each pending or running cluster job. This information is used for further calls to the API server in order to keep the job state up to date. In the case of function calls which depend on each other only via the cluster jobs they need to run a corresponding set of jobs including the dependency information is being submitted to the batch system.

IV. Advanced execution models

Extending on this general idea, a more formal execution model can be defined. Generally one can observe that the execution model of predefined tasks triggered by a REST call is a well known concept in the cloud ecosystem known as Function as a Service (FaaS) [16].

A. FaaS for HPC

In FaaS it is typical that a user has a preconfigured task or function which is packaged into a container to be called with varying inputs. These functions are available by user-defined REST endpoints. Since in HPCSerA every user has a dedicated namespace on the API server, this expected behaviour can be replicated on an HPC system using the respective scheduling mechanism for batch jobs.

The basic mechanism for this is shown in Figure 2. It can be seen that the user can send REST requests to the API server resembling FaaS requests. For this, each user has their own namespace `/<username>/function/<functionname>`, where custom functions can be registered at their own discretion. It is important to state that the function name must be

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

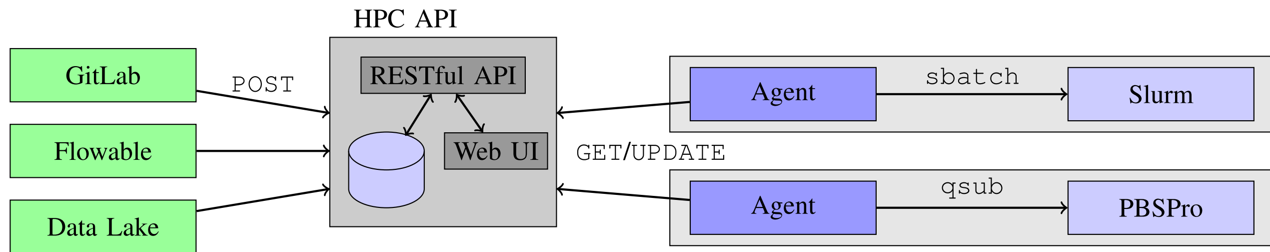

Fig. 1. Components of the architecture: external services, API server, HPC systems (in our use cases we show Slurm and PBSPro as examples, which are used in the Scientific Compute Cluster of GWDG and HAWK at HLRS, respectively).

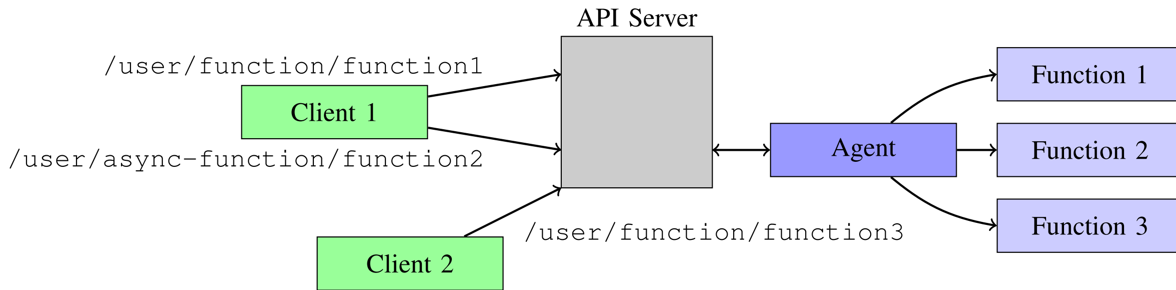

Fig. 2. Basic schema of the FaaS methodology.

unique within the namespace of each user and is not being further isolated by additional structures like the notion of projects. Once a client has posted a function call to the API server, it will be available for the agent to be pulled with the corresponding GET request. The agent then actively pulls these function calls and dispatches them by calling a starter script with the same `<functionname>` located in a preconfigured path, e.g., in the user's home directory in `~/hpcsera/functions/<functionname>`. These functions, which are then being executed, can be anything. It can be a Bash script which is being executed on the frontend, it can be a script fetching data from a remote source and staging it on the fast parallel filesystem of the HPC system, or it can be a simple job submission to the respective batch system, to give just a few examples. Since only executables are being executed, there are no inherent limits to the capabilities of these functions.

B. Long Running vs. Short Running Functions

Since HPCSerA does not enforce any boundary condition on the user-defined functions, it is important to differentiate between long-running and short-running functions from the beginning. The most important difference from the user's perspective is that long-running functions will be usually executed asynchronously, whereas short running-functions can be executed synchronously as well. The reason is that in this case a TCP connection between the client and the API server can stay established during the entire time. Therefore, the HTTP response will correspond to the output of the function (cf. Section IV-G), e.g., a response code `200` would directly mean that the function ran successfully. There might even be some payload data attached to the response, which can be immediately used by the client. The client process is blocking for the duration of the HTTP request. These functions, however, do not only need to have a short runtime, but also need to have limited resource requirements. In those cases, an oversubscribed queue (commonly used to enable interactive jobs) can be used, which can be created and managed by typical resource managers like Slurm.

In the other case, during an asynchronous function execution the client would get an immediate HTTP response from the API server. Here, the return code `202` would however only mean that the request to execute the function was successfully accepted from the API server. This allows for the established TCP connection between the client and the API server to be terminated. Therefore, the client process would only be blocking for the duration of the initial communication with the API server, but not for the entire time the function needs for processing. However, this leaves the client without the optional output data of the function, which might be required. This can be solved on the client side by providing a callback URL in the HTTP header when the initial REST request is made. The API server would in that case make a callback to the client once the function has finished. This is possible since the API server offers statefulness of the functions. Since the API server itself is not meant to handle large data transfers, usually S3 will be used for these cases. Therefore, it might be advantageous to implement some event handling using S3 rather than the API server.

About the difference between synchronous and asynchronous jobs which require access to the compute nodes that are managed by a dedicated resource manager like Slurm it can be stated from the HPC perspective that the synchronous

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

case is using `srun`, whereas the asynchronous function call is using `sbatch`.

For HPCSerA a single function configuration is enough to execute the same function synchronously and asynchronously. The client can then choose the mode of execution at runtime and just distinguishes between those modes by using a different Endpoint, i.e., either the `/<username>/function/<functionname>` for the synchronous execution, or the `/<username>/async-function/<functionname>` for the asynchronous execution.

C. Remotely Building Complex HPC Jobs

Offering a FaaS infrastructure based on HPC which enables long-running, data intensive and highly parallelized functions is a useful addition for those users who are already in the FaaS ecosystem. There is, however, also the HPC-native user group, where people would like to be able to access and use an HPC system as before, just with a RESTful interface. In order to combine these two scenarios, a closer look at the typical HPC usage is necessary. The usual workflow for users working on HPC systems can consist of several steps:

- The environment and binaries for the computation are prepared. This is mostly done interactively on the frontend.
- Input data for IO-intensive applications is staged on a fast parallel filesystem prior to the job submission.
- Last changes to the batch script are done and the job is submitted to the batch system.
- After job completion the results can be inspected and possibly backed up.

Since there are no restrictions on the capabilities of the functions, one can recreate the workflow described above under two conditions:

- The execution of a function can depend on conditions.
- Code ingest needs to be supported.

The first condition is derived from the requirement that a job can only be submitted to the batch system once its environment is built and its input data is staged. There can also occur other examples and more complex conditions. Since HPCSerA is not in any way supposed to replace a workflow engine it is also not supposed to handle complex conditions on its own. Therefore, one can only add the condition to a function that it should start only after one or more other functions have (successfully) finished. The logic to determine whether a function call has been successful or not has to be within the function itself.

In order to build complete end-to-end HPC jobs with this mechanism these function calls need to be embedded in a suitable data structure.

In Figure 3 it is shown that all function calls are organized within a data structure called Job. A user has to first create a new Job, which gets a unique `JobID` assigned by the API server. Afterwards, a user can call functions within the context of a Job. These function calls then get a Function-ID associated to them. Conditions can be assigned to these function calls, i.e., other functions within this Job structure have to be (successfully) finished before this function can start. This mechanism allows to build up a typical, multi-step HPC job as described above, by calling consecutively the exposed FaaS REST API. Independent function calls will be executed concurrently. In our example, this applies to both the *Prepare Binary* and *Stage Input Data* functions which have no dependencies. *Dispatch Batch Job*, on the other hand, can only be run once both previous function calls are completed. Finally, *Postprocess Data* is run once all other function are completed.

Alternatively, one can define a single HPC job in HPCSerA using a single `YAML` file. In this case all functions need to be known when submitting the job request to the API server. If not specified, the functions are executed in the order they appear in the `YAML` file, and an implicit dependency on the previous function is assumed. In the consecutive buildup where independent functions are called in the context of a job, additional function calls can be issued to the same Job-ID at a later time.

D. Virtual Function Calls

Since all dependencies that HPCSerA is supposed to resolve should only cover the exit status of a function, i.e., with or without error, a mechanism is needed to map more complicated conditions onto this boolean. One example for such a more complicated condition would be that a function should only start after some special resource, like a certain block device, was provisioned.

To cover these cases, one can define *Virtual Functions*. These functions differ from normal functions in that they do not need to be pulled by the agent and then need to be executed. Instead these functions are only existing on the API server. There they expose a REST endpoint, where from an external source the state of the Virtual Function can be changed. This means that some external program can make a REST call to that endpoint to set the state of the function to (successfully) finished. This is an alternative, similar to the call-back URLs provided by clients when triggering an asynchronous function. When an external REST call is used, the necessity to execute functions which do busy waiting to check if a certain condition is met, can be avoided. However, functions which do busy waiting can also be used in a straightforward manner, as long as the necessary logic is implemented to differentiate between the waiting state since the condition is not yet satisfied and the failed state where the condition will be never satisfied. In the latter case the function should be terminated with the corresponding unsuccessful state. If a proper failure condition can not be formalized, when a condition failed and will not be met in the future, a final wall-clock time should be specified after which the function is terminated and the state of the function is unsuccessful.

E. Function Configuration

There are two different ways to configure and deploy a function. The first option is to connect to the HPC system

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

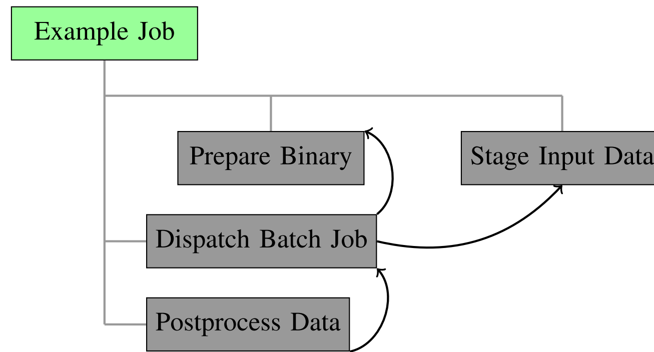

Fig. 3. Sketch of a Job data structure in which three different functions with corresponding Function-ID's are organized.

via SSH and prepare the executable which is called when the function is triggered. This executable can be a binary, or a shell script, for instance. In case a binary should be executed within a certain environment, one can wrap the call of the binary within a shell script. If more complex environments are required, the executable can be packaged togteher with its dependencies into a container, e.g., Singularity/Apptainer [17] can be used. Once the function is configured within such a SSH session, it can be called afterwards by the agent, therefore it is also immediately available for the client on the API server.

The alternative is to configure a new function via the API server, i.e., completely within the context of HPCSerA. For this, all necessary files can be zipped or tarred, and sent along with the configuration request, which is available on a dedicated REST endpoint. The agent will then accept the archive, unpack it into a temporary directory, and will execute a preparation executable. This preparation executable can be as simple as to just copy the the function executable into the function directory of the agent. More complicated examples may include some code which needs to be compiled. For large files, like a Singularity/Apptainer container, it is recommended to upload those via REST to an S3 Bucket. Then just passing a preparation executable to the agent, which fetches this large file from the remote bucket and places it in the necessary path on the HPC system, is enough to configure such a function.

F. Passing Arguments to Functions

Some or even most functions will require that some arguments are passed to them when calling them. These can be passed to the API server of HPCSerA either as Uniform Resource Locator (URL) query parameters, or as a JSON file. In the first approach an arbitrary list of key value pairs can be passed with the calling REST endpoint, e.g., `/<username>/function/<functionname>?k=val`. This call would forward the key `k` with the value `val` to the function in two possible ways: Either the agent would export an environment variable `<PREFIX>_k` with the value `val` (where `<PREFIX>` can be set by the user) before calling the executable corresponding to the called function or alternatively, these key value pairs can be formatted into a single command line string which is appended to the binary call, as it is common when executing an executable on a Linux shell.

In case a function requires more extensive arguments, this previous discussed method is not handy anymore. Instead, one can use a JSON file which is send along with the REST request to trigger the function. This JSON file is then simply forwarded from the API server to the agent which accepts the JSON file and stores it locally. The file path can then be passed as an argument to the function. Neither the API server, nor the agent will in any way process the content of the JSON file. If a function requires this kind of complex input data, the logic needs to be implemented by the function itself or a wrapper script.

G. Returning Output Data to the Client

When a function call is completed, the method of returning its results depends on the mode of execution and the volume of the produced data:

- For synchronously executed functions (cf. Section IV-B) the results are available by the time of completion and can be included in the HTTP response. If applicable, the results can be completely included in the form of a JSON structure produced by the function, e.g., for scripts that query the status of the system, such as custom calls of the batch system or storage CLI tools. Binary data, such as base64-encoded BLOBs can be included, although for the purpose we recommend, especially in the case of a high volume of produced data, that the JSON structure merely contains information about the location of the output data, for example a file system path on shared storage or the URL of an S3 bucket.
- If the function call is asynchronous, only information about the data structures created on the API server can be relayed, in particular the `JobID`. This has to be kept on the client side and used for later status requests.

H. Error Handling

Since the functions have a state, which is managed by the API server, and for instance distinguish between a successful and an unsuccessful exit, the user-defined functions are ideally able to distinguish between those states. However, some error in the code execution itself is not the only error which can

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

occur. It could also happen that during the function execution the process is unexpectedly killed, or the host is suddenly turned of, for instance due to a power outage. When the agent is able to detect those interruptions, where a function stopped processing without sending a proper exit code, it assumes that a crash unrelated to that particular function has happened and will trigger the function execution again. In order to support this behaviour a function should be idempotent, i.e., it should be possible to execute a function multiple times with the same input parameters and it will always produce the same output.

I. External Job Dependencies

With a slight modification in the data structure describing a job and the included functions, our architecture can support medium-term storage of campaign data as well: The set of `subJobTypes` was originally designed to be run in order, but a more generic solution is given by implementing a directed acyclic graph (DAG) of dependencies. Hereby each function can define one or more dependencies on another function, which can either exist in HPCSerA or represent an external event via a virtual `FunctionID`. The latter is marked as done via an external source, for example when campaign storage or a data source is ready to be used as job input data. This workflow is typically used for research projects and can include dependencies between compute jobs, storage provisioning and data migration. However, the conventional linear chain of `subJobTypes` is still included as the special case where each step depends on predecessor. As shown in the first half of Figure 4 this variant is implemented via dependencies between functions. However, in the more general case, as depicted in the second half, multiple functions could depend on the same prerequisite, in this case `B1`. If a subset of the function calls is implemented via batch jobs and all other dependencies pointing outside of the set are already fulfilled, the corresponding subgraph can be submitted in one step, thereby delegating further resolution of the remaining dependencies to the batch system.

V. Security 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 [18, 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*. Therefore, it is 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 [18], 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 5) 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 2FA 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

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

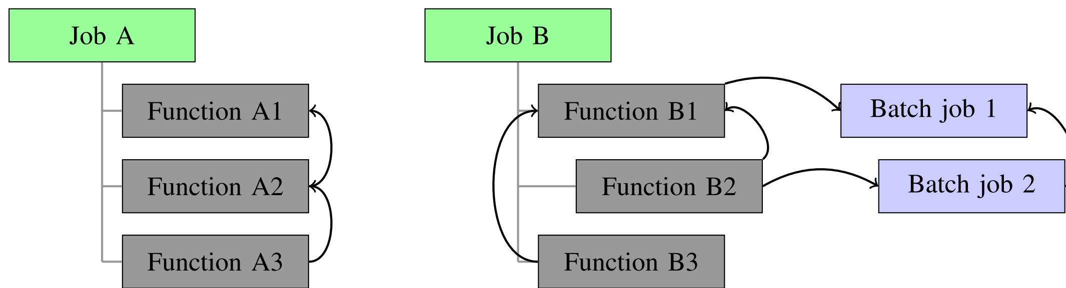

Fig. 4. Jobs with implicitly defined dependencies and a custom dependency DAG.

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 [19] 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 5. 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. step 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 has 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 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 5) 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. However, authentication for access to the WebUI via the user's existing account from their HPC center can be implemented as SSO using OpenID Connect Federation.

VI. Implementation

In the following, more details about the technologies chosen for our implementation are provided. Due to the conceptualized architecture in Section V, this section has a focus on the current scope definition and the authentication/authorization scheme employed. Generally, the OpenAPI 3.0 [20] specification, 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, was used to define the RESTful API. 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*, who 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 minimizes 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 authorization code 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 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

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

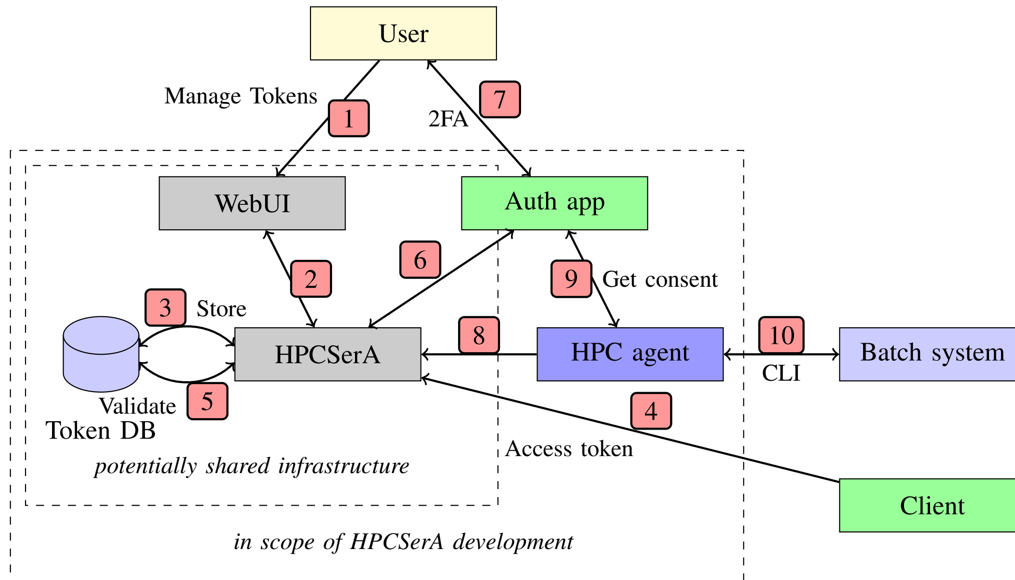

Fig. 5. A sketch of the proposed token-based authorization flow. The following parts are shown: 1) WebUI login 2) Connection to the HPCSerA service 3) Storage of access tokens 4) Client connecting to the API 5) Validation of access tokens 6) Authorization request 7) User interaction with the Auth app 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

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.

C. Mapping of Roles to Functions

In order to provide the user with a FaaS interface which is capable of handling automated machine-to-machine communication of headless apps the previously defined roles need to be mapped on the FaaS endpoints. The most important differentiation is still between the `POST_Job` role and the `POST_Code` role. The latter is required, when a user wants to configure a new function via the API server. Here, the user can upload new code either directly as an archive, or via an external storage. Therefore, the configuration of a new function corresponds to the `POST_Code` role. The client making that request needs to have this elevated access rights.

On the other hand, simply triggering the execution of an already configured job, for instance on new input data, corresponds to the `POST_Job` role. As shown in Table I, this role is not security sensitive. Thus, it can be used by a client without any manual interaction as long as the client has a

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

token with the corresponding role. Therefore, HPCSerA can support automated FaaS functionality towards its client.

The agent side was not considered critical for the admins, but optionally critical for the users if they distrust the API server, if they want to implement their own 2FA mechanism here. To support the previously discussed endpoints, the client needs to either use the `GET_Job` role to receive the request to execute a function, or the `GET_Code` role to pull some new code and to configure a new function. The agent would then also require the `GET_JobStatus` role and the `UPDATE_JobStatus` role to manage the state of the functions, which is maintained on the API server. Via the `UPDATE_JobStatus` role the output data of a function can be send to the requesting client. The client would then also need the `UPDATE_JobStatus` role in order to be allowed to receive the output data.

D. Assigning Roles to Clients and Agents

The fine-grained distinction between those different roles, as discussed in Section VI-C, is an important part to provide the highest level of security while enabling a high degree of automation. This means especially that users should only use tokens with the minimum roles attached to them. For instance, if a user will configure functions always manually within a SSH session, the token of the agent should not have the `GET_Code` role enabled. Users define the roles of the tokens in the current setup within the WebUI. Here, users can either check the needed roles, create a token, and copy it out of the WebUI or they can, upon request from a client or agent via the presented detached OAuth flow, choose which roles should be associated to the token which will be created. Once a token has been created, it can also be revoked if it is not needed anymore or a potential breach is assumed.

VII. USE-CASES

Due to the previously stated changes in the architecture, there are certain adaptions in the previously presented use cases [18]. 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 [21], 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 V-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*.

There are various levels where dependencies between jobs can be managed. The following descriptions and examples refer to Figure 6:

1) Dependency resolution can be completely handled by the *workflow engine*. In this case, workflow tasks are submitted as individual jobs via HPCSerA. If there is a dependency between two jobs that require a batch job to finish, on completion of the first cluster job the agent updates the job state on the API server from which the workflow engine eventually obtains the new state. In our example, this is the requirement to proceed from Task I to the dependent Task II. Only then can the second job be submitted to the API and if finally retrieved by the agent and submitted to the batch system. In conclusion, this variant is the easiest to implement but involves a high amount of latency for resolving job dependencies.
2) For jobs that are submitted with multiple Function-IDs, the *API Server* will handle dependencies by only providing function calls to the HPC agent for which all function calls on which they depend have been successfully completed. Comparing to the previous scenario, once the agent has marked the last batch job of Job A (A2 in our example) as completed, the function status of A2 on the API server is updated and the next one (function A3) can be immediately retrieved and run. While the dependency chain has to be implemented by building more complicated calls to the REST interface, there is no back and forth communication with the client contributing to the latency.
3) In view of Section IV-I the most low-latency resolution of job dependencies occurs when multiple Function-IDs which contain batch jobs are presented by the API server to the *Agent*. In this case, the completed first batch job (A1) directly leads to the scheduling of the second batch job (A2) by the batch system without interference from any *HPCSerA* components.

C. Data Lake

In order to provide high-performance computing capabilities to a data lake [22], *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

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

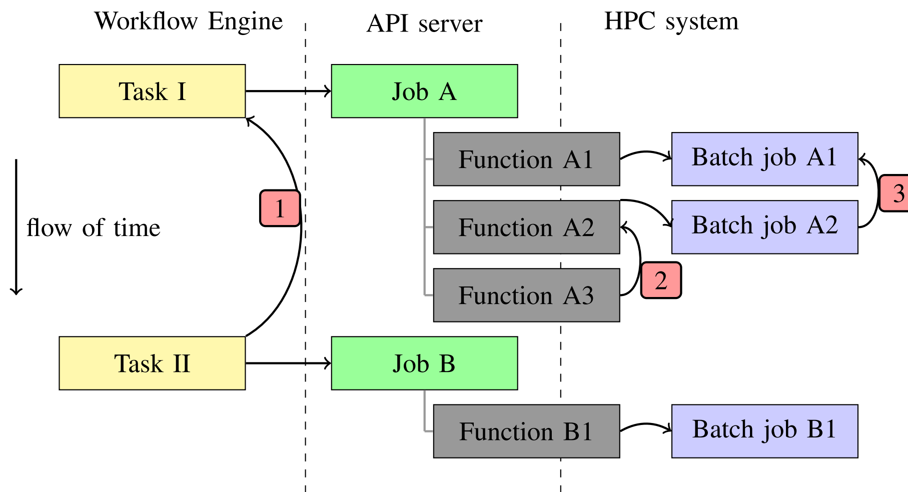

Fig. 6. Overview of the levels at which function dependencies can be resolved.

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 users had to trust the data lake, and hereby the admins, with their *bearer tokens* before. By introducing *OAuth* and enforcing 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.

VIII. Conclusion and Future Work

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.

Moreover, we presented an extension on the execution models that are possible in our architecture by supporting a *Function as a Service* (FaaS) idiom. Here users can define dependencies between function calls and choose between synchronous and asynchronous execution in analogous way to how HPC jobs can be immediately run on an oversubscribed queue vs batched for running with guaranteed resources.

Compared to the related work discussed in Section II, this paper presents a RESTful API which does not require the users to provide full SSH access to a potentially untrusted API server, as it is required in [12], [13]. Instead, using an agent which pulls from user space the incoming tasks guarantees that the user alone stays in full control the entire time. This approach is paired with a fine-granular role-based access model, and a novel authorization flow to enable OAuth-like authorization for headless applications. This mechanism allows automated workflows to access an HPC system to execute pre-configured tasks on new input data while still enforcing a similar security level to an SSH access with 2FA enabled.

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. This is supposed to extend the currently used consent mechanism based on a WebUI. 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.

In addition, the FaaS approach lends itself well to collecting statistics about the frequency of function calls as well as metrics about their runtime behaviour. The most convenient way of presenting this information to the user would be inside the Web UI that is already required in our architecture for project and token management.

It would be advantageous to ease the configuration process, ideally to a degree where a user can just insert some code in the web interface of the API server. The agent would need to be extended to automatically work with predefined templates for different languages. For example, if a user inserts Python code in the interface, the agent would prepare a virtual environment with the necessary modules and insert the Python file in the correct place and transparently manage the corresponding environment information.

Acknowledgments

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

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

REFERENCES

[1] M. H. Biniaz, S. Bingert, C. Köhler, H. Nolte, and J. Kunkel, “Secure Authorization for RESTful HPC Access,” in *INFOCOMP 2022, The Twelfth International Conference on Advanced Communications and Computation*, C.-P. Rückemann, Ed., 2021, pp. 12–17.

[2] J. Decker, P. Kasprzak, and J. M. Kunkel, “Performance evaluation of open-source serverless platforms for kubernetes,” *Algorithms*, vol. 15, no. 7, p. 234, 2022.

[3] 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.

[4] 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.

[5] 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.

[6] 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

[7] J. Buchmüller *et al.*, “Extending an open-source federated identity management system for enhanced hpc security.”

[8] OpenFaaS. (2022) Invocations. [accessed: 2022-12-13]. [Online]. Available: https://docs.openfaas.com/architecture/invocations/

[9] 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.

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

[11] 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.

[12] 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.

[13] V. Svaton, J. Martinovic, J. Krenek, T. Esch, and P. Tomancak, “Hpc-as-a-service via heappe platform,” in *Conference on Complex, Intelligent, and Software Intensive Systems*. Springer, 2019, pp. 280–293.

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

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

[16] J. Decker, P. Kasprzak, and J. M. Kunkel, “Performance evaluation of open-source serverless platforms for kubernetes,” *Algorithms*, vol. 15, no. 7, 2022. [Online]. Available: https://www.mdpi.com/1999-4893/15/7/234

[17] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[18] 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.

[19] 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

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

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

[22] 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

2022, © Copyright by authors, Published under agreement with IARIA - www.iaria.org