An API to Include HPC Resources in Workflow Systems

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

Christian Köhler
eScience
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
christian.koehler@gwdg.de

Hendrik Nolte
eScience
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
hendrik.nolte@gwdg.de

Waqar Alamgir
Internet Technologies and Information Systems
Technical University of Braunschweig
Braunschweig, Germany
w.alamgir@tu-braunschweig.de

***Abstract*—The demand for processing power by modern data analyses is continuously increasing. High-Performance-Computing (HPC) resources can help but the standard process is for users to log in to use the HPC systems which is often complicated and not well suited for the integration in workflows. In order to bridge the gap between external workflow tools and the usage of HPC resources, we designed and implemented an application interface. This API allows workflow systems to submit HPC jobs along with required artefacts to the queuing system without a direct login of the user. The presented API regards the required safety regulations by ensuring the identity of authorised external workflow systems, as well as the executing HPC systems with a token-based authentication model. In this paper we describe the design of the API and present three use-cases. In the data lake use-case, a novel technique for provenance auditing without runtime overhead is presented which is particularly well suited for HPC systems.**

***Index Terms*—HPC, automation, RESTful API, workflow engine, data management, provenance, data lake**

I. Introduction

A typical workflow one might think of when describing the usage of an High-Performance-Computing (HPC) system can be outlined as follows:

- As a central component of getting started on an HPC system, shell access has to be set up for the user, which is typically done over Secure Shell (SSH). Since HPC systems are a major target for cyberattacks, as exemplified in [1], providers like scientific institutions or private businesses are generally employing extra security measures, such as enforcing key-based authentication, restricting the source IP for user logins, as well as limiting the users' capabilities of accessing the public internet from compute nodes or even frontend nodes.
- Similar issues arise when users want to initiate the transfer of input and output data for their jobs, as well as the transfers that are necessary for setting up software on the system. While the latter is commonly delegated to the operating staff, data transfers are recurring tasks which have to conform to security constraints, as well as policies aimed at maintaining the performance of the system, such as delegating the task of huge data transfers to dedicated hardware.

A. Related Works

Our approach is complementary to the REST API provided for, e.g., the batch system *Slurm* via its own `slurmrestd` in the sense that our reliance on outgoing connections avoids any administrative work on the part of the HPC provider. Moreover, the split into an external API working in tandem with a local script incorporates data management-related tasks outside of the batch system from the outset. In cases where the HPC network can be set up to allow incoming connections to the REST endpoint, a homogeneous set of systems is to be used exclusively, and remote access to the batch system is sufficient, the included API is of course the more effective solution.

NEWT, the *NERSC Web Toolkit*, follows a similar approach of presenting HPC resources over a RESTful API and using JSON formatting for the response. However, the implementation is custom-tailored for the resources at Lawrence Berkeley National Laboratory (LBNL) [2].

The microservice-oriented solution *FirecREST*, which has been developed at Swiss National Supercomputing Center (CSCS) roughly at the same time as our solution, differentiates asynchronous cluster jobs from synchronous shell scripts as well, focuses on Slurm as the HPC workload manager and handles data management also for large files [3].

B. Limitations of the interactive usage model

In addition to the mentioned preparatory steps for setting up a workflow by the user, the manual management of HPC tasks runs into limitations in various usage scenarios:

- There might be external triggers which start an entire pipeline of data ingest, processing, and finally the upload of results, such as acquisition of data via scientific instruments, e.g., electron microscopes. In this case, it is desirable that a user's existing data management tools can delegate the entire chain of tasks to the HPC system.

- External applications or services to which the HPC system acts as a back end via templated jobs that, once initially configured, vary only in the provided input data. These might range from rather sparse (such as user-selected ranges in parameter studies of numeric simulations) to very data-intensive, such as asynchronous processing of image data. An example of this approach is *GenePaint* an online "atlas of gene expression patterns" [4].
- Software development projects working on applications that are intended to run on HPC systems, implying dependencies on the available compilers, libraries, and specialised fabric or accelerator hardware. The collaborative workflow should support automatic testing of each iteration in the native HPC environment without manual intervention.

Custom-tailored architectures employing existing cloud infrastructures are often the answer to these demands, in fact dynamically switching between a cloud provider or an HPC system depending on each instance of an application might be desirable, c.f. [5]. However, various constraints can make the integration into an HPC system necessary, such as an existing software stack that is hard to replicate on a cloud infrastructure, the bare-metal performance achievable without an intermediary virtualisation layer, as well as simple economic considerations, like avoiding the costs of additional software licenses and replicated storage for long-term resident data, such as genomic databases in bioinformatics.

The remainder of this paper is structured as follows: In section II we motivate the the need for an HPC API by introducing potential usage scenarios and extracting our requirements from these. Section III focuses on the design of our solution and gives an overview of the implementation. Finally, in sectionIV we elaborate on three use cases that rely on the presented solution.

II. Motivation for a general-purpose HPC API

Our proposed solution to the problem of automating the HPC tasks outlined in the introduction is the design of an Application Interface (API) that abstracts the notion of an HPC job and, with certain limitations, the artefacts needed for its execution, away from the command-line tools typically employed, thus making the resources available to external services. Viewed this way, the system can itself become a background service that is not visible to the end-user and becomes, to some extent, an implementation detail, much like, e.g., a database instance or a storage back end. The main challenge is that the system should conform to the typical security restrictions so its setup doesn't involve major redesign work in existing security concepts.

A. HPC as a backend service

The envisioned architecture has to be able to accept jobs as part of a workflow that doesn't necessarily have to originate from the system itself, enabling the user interaction to depart from the classical approach (i.e., preparing the application, input data and job script tailored to the available infrastructure in the HPC file systems, submitting the job in an interactive shell session, and handling post-processing and data transfers manually) - one example would be presenting a web interface that allows the customer to

(a) configure a standardised job for a Computational Fluid Dynamics (CFD) application by specifying the parameters and upload the geometry, then using the HPC system as a backend to calculate the flow asynchronously and
(b) visualise the results of finished jobs and automatically attach citable persistent identifiers to them.

B. Requirements for an HPC API

We aim to enable standard users of the system to be able to set up access via their individual accounts through the API. This should happen transparently at the user's discretion and in particular without the need to set up a system-wide solution that would need to be approved and handled by each system's administrators. By setting up their user account to process jobs submitted over the HPC API, the user trusts the implementation to (a) faithfully translate jobs that are accepted from external services that were individually authorised by the user. This notion of trust also has to work the other way around, i.e., any HPC system that accesses the API has to be authorised first as well, since the job's metadata and artefacts might be confidential, and confidence in the results comes from the fact that it has been processed by a known HPC system. To complete the circle, those results should only be accessible to trusted services, in the simplest case the one by which the job has been submitted. Apart from this most critical component of the solution, that is, acting with users' privileges on their behalf through the API, also the remainder of the infrastructure should support being easily provisioned by the user - the service-facing API endpoints (potentially worldwide accessible). However, this part should allow shared operation for multiple users, provided a suitable authentication scheme is in place. The semantics used in the API to describe the metadata and states of batch jobs that are ultimately processed, should be batch system-agnostic. The goal here is not to establish some generic standard of job metadata, which could be a non-exhaustive attempt at integrating the specifics of various existing batch systems at best. Instead, our aim is to establish a suitable common denominator that allows the simultaneous operation of various HPC sites with potentially different batch systems as back ends for the same set of services.

Finally, any services relying on the HPC API should have the possibility to inquire on the status of any jobs which are already submitted yet not fully processed. These could be just received, already fetched by one of the connected

HPC systems, waiting in the system's internal batch queues, or ready for retrieving the results. We expect this to be a useful feature since it is needed to, e.g., provide dashboards on the jobs' processing state or to dynamically decide which system to delegate jobs to (there might be multiple processing back ends in addition to the HPC systems addressed by our solution).

C. Potential Use Cases

In addition to the abstract characterisation of tasks that limit the manual HPC workflow, we give some concrete examples of applications which potentially benefit from our solution, as well as some who are already doing so where indicated:

- scientists considering classic HPC batch jobs only a part of their broader data management workflow and want to automate this process including the transfer to and from the system (An example is given in the "Data Lake " use case.)
- users of Data Analytics tools (e.g. *Apache Spark*) who want to automatically have a cluster of worker nodes (unspecific to their actual project) provisioned as a batch job and afterwards submit their job to this cluster in the same way they would have done so on a non-HPC infrastructure
- customers working on parallel codes in *GitLab* and want to run their continuous integration (CI) jobs in those projects' native software environment, in particular if (a) the compilers and/or libraries are commercially licensed products whose installation in a dedicated *Runner* would mean extra overhead or (b) need to test their build in a distributed job against a high-speed interconnect (c.f. the "GitLab" use case)
- users that often submit jobs which are highly schematic in nature, such as parameter sweeps of simulations or CFD simulations (e.g. *OpenFOAM*) or climate models (e.g. *CESM*) can be provided with an interface that only requires them to state initial values, resolutions, geometries etc. (An application of this kind is the motivation for our "Flowable" use case.)
- researchers who want to contribute to the quality of scientific publications by making the processing from input to output data transparent and reproducible by automatically attaching persistent identifiers (PIDs) to the output of their jobs, enriching them with metadata about the job itself and (ideally, if publicly available) the location of the input data.

III. Design

In the following, we describe the most relevant aspects for the design of the interface. Flexibility to adjust to different environments let us summaries the the desgin as follows:

- A Representational State Transfer (REST) API service that is being accessed over the HTTP(S) protocol is deployed on a host (bare metal or virtual machine) which is reachable by the external services as well as the HPC system. From the point of view of the HPC system all connections to the API are outgoing, so the potential impact of firewall configurations is minimal. Since REST client libraries are ubiquitous in a multitude of programming languages (or at last HTTP clients and JSON parsers), this design choice makes the integration of a new service relatively easy.
- We provide a generic script to be installed by the user in the context of their existing account on one or multiple HPC systems. As long as this script is running, either continuously in, e.g., a GNU Screen session or by being periodically started by a cron job, it will poll the API for jobs that need to be processed on the particular system and submit those to the batch system, query the status of the batch system to determine which jobs have been finished and finally update the status of jobs via the API. It is only at this stage that knowledge about the batch system is needed, thus a heterogeneous collection of systems can process jobs from the same endpoint as shown in Fig. 1.

Jobs can be defined to be executed on the frontend node (one particular machine in the HPC system where outgoing traffic is allowed and where the script runs) as well, because various tasks such as the (un)archiving of artefacts, the transfer of job input and output data as well as the compilation of code as a preparatory per-job step do not warrant the launch of an extra batch job. Our approach for these kinds of tasks is to start with a minimal set (the pass-through of shell commands as well as basic data management tasks) and to formalise recurring tasks only as needed in order to avoid over-engineering the solution.

A. What is not included

The scenarios where we envision an application of our API share a certain uniformity of the jobs that will be submitted and we focus on the automation of those. Therefore the initial setup and testing of any new kind of job should not be shoehorned into the API approach, but rather be carried out manually and only afterwards schematised so that the bulk of jobs can be handled automatically. However, there is demand for the interplay between new software versions and data which is described in the "Data Lake" use case.

B. Security and user management

The API can be provisioned by an individual user or, if multi-user operation on a central setup is desired, authentication against a local user database or LDAP has to be performed. There are two kinds of stateful data on the API server: Authentication tokens which are generated on a per-user basis (this could in theory be outsourced to an external service) and authorised for usage by either a service which needs to submit and manage jobs, inquire about their state and fetch the results or by HPC systems which need to receive the jobs, update their status and uploads the results. These API access tokens are then shared with the client run by the service and the script running on the HPC frontends, respectively. If trust of a system on either side is to be revoked, all that is needed is the removal of the corresponding access token. At

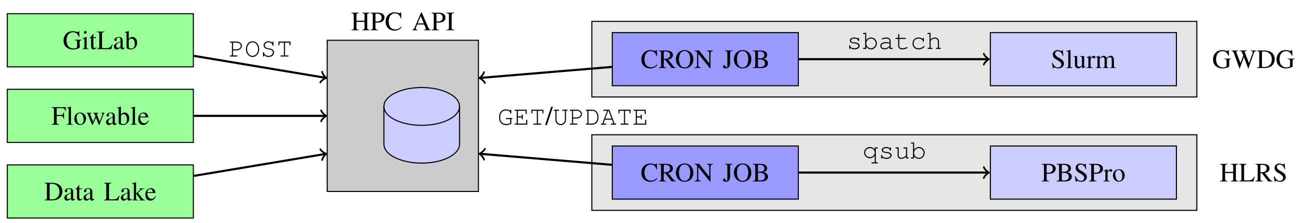

Fig. 1. Components of the architecture: external services, API server, HPC systems (in our use cases the Scientific Compute Cluster of GWDG and HAWK at HLRS).

this level, more fine-grained access control per token could be a sensible extension of our design.

C. Components

The core component to be developed in the project would be an application server ("application" in the following) which provides the following interfaces:

The **HPC system** has access via an API in order to regularly poll the application for new jobs that need to be executed on behalf of the user and to post the results (or references to them in the metadata) once they have finished. Our envisioned implementation of this step would be a standardised cron-job that is developed in the project and provided in a way that is easy to set up for the user. This approach implies that no additional firewall rules and/or accounts have to be set up on the HPC system itself. The user of the application to be developed is authenticated by a token that is created upon initial configuration of the cron-job and reported to the user and vice versa with an API key generated by the application itself.

A **REST API** is needed for importing jobs from third party applications, e.g., GitLab Runners as described above, into the job queue of the system and for querying the state of the queue. Credentials that are necessary to authorise the submission of jobs are handled by the system itself, in particular no SSH key that would grant access to the HPC system has to leave the user's personal machine.

The final step is a **web interface** which

- allows the user to validate that the HPC system is connected and authentication works in both ways, showing basic information such as the status of system (available nodes, current utilisation etc), and the status of the user's own jobs,
- facilitates the submission of supported jobs, such as parameterised simulations, Data Analytics applications etc. as described above and shows their results, possibly visualisations, if applicable, and
- is used to authorise external applications, such as GitLab.

D. Implementation

Our design has been prototypically implemented as HPC Service API (HPCSerA) in [6]. Using the *OpenAPI 3.0* specification (known as *Swagger* until 2015) an authoritative definition of the API was created in the *YAML* format. Therein *component* definitions determine the schema of HTTP responses, e.g., a `Job` component containing information on a job such as its internal identifier (ID) in the batch system, and path definitions assign possible HTTP request methods ("verbs") to individual paths as well as possible response status codes. For example, the following excerpt from the `swagger.yaml` definition states that the `/job/{jobId}` path supports the `GET` method to receive information on the instance of the `Job` component referenced by `jobId`:

swagger.yaml (abbreviated)

```
openapi: 3.0.0
 /job/{jobId}:
  get:
   summary: Finds job by ID
   description: Returns a single job
   parameters:
   - name: jobId
   responses:
    "200":
     description: Successful operation
     content:
      application/json:
       schema:
        $ref: '#/components/schemas/Job'
```

The client script handling jobs on the HPC side is implemented in Python as well, which is conveniently possible since a Python module for the API corresponding client called `swagger_client` is automatically generated alongside the server code. This module includes a corresponding class definition for each component in the API specification which allows straightforward interoperation between Python code and the component instances accessed through the API. In the case of the central `Job` component, the class definition is augmented by methods that use its metadata to execute scripts locally and to interact with the batch system. This is done repeatedly for all jobs that are either new, being queued or currently running on the batch system. Any shell commands being run implicitly are defined in a separate configuration file to allow easy modification, e.g., for interacting with a different batch system or to customise data transfer tasks. The full documentation of HPCSerA is presented at [7]. In the prototypical implementation, a statically configured set of users and access tokens is being used. To make the service fully production ready, dynamic user management and authentication as well as token generation and management has to be implemented as well.

Local persistence of jobs, which is strictly needed only during their runtime, is implemented with a MySQL database that is running on the same host as the API server itself. The local file-system is used for temporary storage of uploaded artefact files.

IV. Use-Cases

We selected three different use-cases two show the strength of our flexible approach. Each use-case has a different technical background and different target user groups.

A. GitLab

For Continuous Integration (CI) workflows relying on GitLab runners, various implementations are available, among which SSH executors are in principle the most suitable for integrating with an HPC system. Of course, private credentials and SSH keys should never leave the personal machine of any user, therefore it is not trivial to deploy these runners on an HPC system. However, running them over the API client is not a problem and the API key can be conveniently handled by GitLab Secrets Management and centrally revoked if necessary.

The client implementation generated by *Swagger Codegen* is used in [6] with a configurable `hpc.yaml` file which directs the HPC frontend to run various `subJobTypes`, such as raw shell commands, file transfers and archiving tasks, as well as batch jobs. This approach allows a clean integration with the `.gitlab-ci.yml` file used in a GitLab repository in order to trigger the API client's tasks. The API supports uploading small file artefacts that should accompany the job and do not warrant the setup of a dedicated file storage by accessing the `/file/uploadFile` path with the `POST` method. This feature is used to ship the source code of the git commit to be tested on the HPC system.

B. Workflow Engine

The Open Forecast project [8] has the goal to integrate HPC resources into a generic workflow system to allow users to process open data. Flowable [9] was chosen as workflow runtime engine to execute user-defined workflows. Although Flowable offers many BPMN-specific (Business Process Model Notation) tasks, a possibility to select a different runtime environment for dedicated workloads is not available. The presented API allows to interact with the HPC system using the built-in HTTP task of Flowable. The HTTP task allows to define and configure REST calls to specify the HPC job. Including this task in a Flowable workflow enables the workflow designer to include user interactions to collect additional information for the HPC job. Eventually the full potential of BPMN based workflows can be used, e.g., data processing on different HPC resources is combined with user interactions such as collecting parameters and sending job status information via email. In our current setup, the HPC job is defined as a singularity container which is pulled from a GitLab container registry, submitted to the queuing system, and started with the previously collected user-defined parameters.

C. Data Lake

A data lake is generally designed as the central repository for all data sets from all data sources in their raw format [10]. In order to ensure proper data integration, comprehensibility and quality some data modelling is required [11]. These models are then being stored in a central data catalogue which is used to perform searches on the data lake and to access descriptive metadata.

Retaining data in their native format prevents a possible information loss due to ETL-Processes and ensures a high re-usability. Due to the high re-usability there is the need to support a wide range of different analyses on these data sets. Since these analyses are potentially extremely computationally demanding it is favourable for the data lake to outsource those computations to an HPC system. Furthermore, since all resulting artefacts will be ingested back into the data lake, maintaining concise and accurate provenance data is recognised to be the key requirement for the manageability of the data lake [12]. Various solutions tailored for specific purposes have been proposed for this. *Goods* [13] analyses log files in a post-hoc manner to determine which jobs created a dataset based on which input, which requires that the application writes a suitable log. Similarly, *Komadu* was integrated into a data lake [12] which supported the messaging of provenance information via RabbitMQ, also relying on the explicit support of the application. *DataHub* [14] was equipped with *ProvDB* [15] where user annotations and special shell commands are used to capture provenance information. However, solely relying on user annotations is very error prone and piping commands through a shell into an auditing tool can not capture the entire execution environment reliably without introducing a noticeable overhead. In order to mitigate these shortcomings we present a novel technique to enable retrospective provenance auditing of generic applications which is, amongst others, ideally suited for HPC Systems, where generic provenance tools are still under discussion [16].

In order to perform a generic analysis job, required to serve the wide range of different applications, the user has to describe it in an unambiguous job manifest. This job manifest contains not only the actual compute commands, but it offers a wealth of options to exactly fine tune a job. The specification of a container image is mandatory to enforce better traceability and reproducibility. Furthermore, comments can be made, a job name can be assigned and the data category must be specified. These user annotations are very useful for better comprehensibility and traceability later on, but do not contribute to the actual recorded retrospective provenance data since user-provided information is potentially error prone. In order to further prepare the execution environment, an arbitrary list of git repositories, with corresponding bash commands to build them, can be declared. In addition, environment variables can be defined which get imported into the container for the execution. Also, the input data is defined, either as a list or as a query on the data catalogue on which the analysis is being performed. The special feature here is that the manifest itself is an entity which is getting stored in the data lake with all its entries being indexed. Hereby, all submitted jobs are searchable for all the specified attributes, artefacts can be linked back to their input data and can also be directly linked to the precise job description as well, enabling easy reproducibility and comprehensibility of the origin of artefacts. This job manifest is then sent to the data lake to execute the specified job. Upon receiving this job specification, the data lake generates three different bash scripts: a preprocessing, a run and a postprocessing script. Together with optionally needed assessor scripts, for example one to download the specified data from an S3-Bucket, these are then zipped and posted via an RESTful request to the the HPC API server. Since the data lake has its own user management, the corresponding tokens of the users for the HPC API are stored locally in a database and are associated to the individual data lake user. Using the `hpc.yaml` file, the cron-job running on the frontend of an HPC system is configured such, that it first executes the preprocessing script as a shell script. Here, first of all is a dedicated folder created which is then writable mounted into the specified container image. This mount is then used to clone and build the git repositories provided in the job manifest. Then the repository names and the corresponding commit hashes are posted back to the data lake via the REST-API to update the job entity, in order to enable later precise traceability of the performed computations. The rest of the dependencies has to be installed in the container image itself which is read-only. In order to allow for later reproducibility, the exact binaries of the container image are also being stored in the data lake and are linked to the job entity correspondingly. If some input data needs to be fetched and staged, the corresponding scripts, which were part of the zip, are being called from within the preprocessing script as well. Hereafter the runscript is being submitted to the queuing system. In this script, the required resources are first specified, followed by the definition of the environment variables as defined in the job manifest. Lastly the compute command stated in the job manifest is executed in a shell inside of the container. Only after the run of this job the cron-job executes the postprocessing script, again on the frontend. Here, the created artefacts are ingested back into the data lake, where they are being indexed, linked to the job manifest entity, as well as their input data and are finally stored. Also, in the job manifest specifically provided environment variables are read and indexed as well, which is very useful to have for instance some metrics about the run easily searchable when querying the job manifest entities in the data lake at a later point. Lastly, some cleanup is necessary to prevent the user's home directory from polluting over time.

In summary, the job manifest unambiguously describes the execution of a job. Since all dependencies, inputs and outputs, the used software with the specific version, as well as the actual run commands are defined or recorded each run can be precisely understood and reproduced later on. Here we want to emphasise that there is no requirement for the application to support this provenance recording.

V. Conclusion

The presented HPC API is a powerful and flexible tool to integrate HPC resources in different kinds of workflows. The described use cases feature the deployment in productive environments and exemplify how the HPC API can be used to react on changing demands from users or can even be utilised to solve long-standing problems. The HPC API can be used in communities where diverse working groups have access to more than one HPC provider. Thus, it brings the strength of HPC to a broader audience. In future work the integration of an external and trustworthy token provider will be developed. This will increase the acceptance of this new service by both the users and HPC providers.

References

[1] EGI-CSIRT. (2020) Attacks on multiple HPC sites. [accessed: 2021-05-16]. [Online]. Available: https://csirt.egi.eu/attacks-on-multiple-hpc-sites/

[2] S. Cholia, D. Skinner, and J. Boverhof, "NEWT: A RESTful service for building High Performance Computing web applications," in *2010 Gateway Computing Environments Workshop (GCE)*, 2010, pp. 1–11.

[3] 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.

[4] G. Diez-Roux *et al.*, "A high-resolution anatomical atlas of the transcriptome in the mouse embryo," *PLOS Biology*, vol. 9, no. 1, pp. 1–13, 01 2011.

[5] F. Korte, M. Baum, G. Brenner, J. Grabowski, T. Hanschke, S. Hartmann, and A. Schöbel, "Transparent model-driven provisioning of computing resources for numerically intensive simulations," in *Simulation Science*. Cham: Springer International Publishing, 2018, pp. 176–192.

[6] W. Alamgir, "Design and implementation of an api to ease the use of hpc systems," Master's thesis, Inst. Comp. Sci., Univ. Göttingen, 2021.

[7] HPCSerA - The HPC Service API Documentation. [accessed: 2021-05-17]. [Online]. Available: http://hpc-api.open-forecast.eu/

[8] Open Forecast. [accessed: 2021-05-17] co-financed by the Connecting Europe Facility of the European Union (Action Number 2017-DE-IA-0170). [Online]. Available: http://open-forecast.eu/en/

[9] Flowable AG. (n.d.) Flowable - award-winning intelligent automation platform. [accessed: 2021-05-16]. [Online]. Available: https://flowable.com/

[10] C. Madera and A. Laurent, "The next information architecture evolution: The data lake wave." New York, NY, USA: Association for Computing Machinery, 2016.

[11] R. Hai, C. Quix, and C. Zhou, "Query rewriting for heterogeneous data lakes," *Advances in Databases and Information Systems*, vol. 11019, 2018.

[12] I. Suriarachchi and B. Plale, "Crossing analytics systems: a case for integrated provenance in data lakes," in *2016 IEEE 12th International Conference on e-Science (e-Science)*. IEEE, 2016, pp. 349–354.

[13] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang, "Managing google's data lake: an overview of the goods system." *IEEE Data Eng. Bull.*, vol. 39, no. 3, pp. 5–14, 2016.

[14] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran, "Datahub: Collaborative data science & dataset version management at scale," *arXiv preprint arXiv:1409.0798*, 2014.

[15] H. Miao, A. Chavan, and A. Deshpande, "Provdb: Lifecycle management of collaborative analysis workflows," in *Proceedings of the 2nd Workshop on Human-in-the-Loop Data Analytics*, 2017, pp. 1–6.

[16] D. Dai, Y. Chen, P. Carns, J. Jenkins, and R. Ross, "Lightweight provenance service for high-performance computing," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 117–129.