Cluster Usage Policy Enforcement Using Slurm Plugins and an HTTP API

Matthew Li

Berkeley Research Computing
Lawrence Berkeley National Laboratory
meli@lbl.gov

Viraat Chandra

Berkeley Research Computing
University of California, Berkeley
viraat_chandra_24@berkeley.edu

Nicolas Chan

Berkeley Research Computing
University of California, Berkeley
nicolaschan@berkeley.edu

Krishna Muriki*

Berkeley Research Computing
Lawrence Berkeley National Laboratory
kmuriki@lbl.gov

ABSTRACT

Managing and limiting cluster resource usage is a critical task for computing clusters with a large number of users. By enforcing usage limits, cluster managers are able to ensure fair availability for all users, bill users accordingly, and prevent the abuse of cluster resources. As this is such a common problem, there are naturally many existing solutions. However, to allow for greater control over usage accounting and submission behavior in Slurm, we present a system composed of: a web API which exposes accounting data; Slurm plugins that communicate with a REST-like HTTP implementation of that API; and client tools that use it to report usage. Key advantages of our system include a customizable resource accounting formula based on job parameters, preemptive blocking of user jobs at submission time, project-level and user-level resource limits, and support for the development of other web and command-line clients that query the extensible web API. We deployed this system on Berkeley Research Computing's institutional cluster, Savio, allowing us to automatically collect and store accounting data, and thereby easily enforce our cluster usage policy.

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; • **Information systems** → *Extraction, transformation and loading*; • **General and reference** → Design.

KEYWORDS

account management, accounting, high performance computing, Slurm, Django, plugin, application programming interface

ACM Reference Format:

Matthew Li, Nicolas Chan, Viraat Chandra, and Krishna Muriki. 2020. Cluster Usage Policy Enforcement Using Slurm Plugins and an HTTP API. In *Practice and Experience in Advanced Research Computing (PEARC '20)*.

*This author supervised the work, contributed much of the initial design, and participated in design decision-making.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

PEARC '20, July 26–30, 2020, Portland, OR, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6689-2/20/07...\$15.00
<https://doi.org/10.1145/3311790.3397341>

July 26–30, 2020, Portland, OR, USA. ACM, New York, NY, USA, 7 pages.
<https://doi.org/10.1145/3311790.3397341>

1 INTRODUCTION

Our task was driven by the following primary needs:

- Cluster administrators need the ability to limit resource usage on a per-project basis, and enforce these limits automatically and accurately, as proactively as possible.
- Administrators, Principal Investigators (PIs), and cluster users should be able to monitor usage.
- The system should be extensible, allowing for easy ways to access accounting data from any number of command-line and web applications, such as user portals or allocations dashboards.

To complicate our task, the Savio cluster is divided into partitions, which are charged at different rates using a system of service unit prices. For instance, one CPU-hour on a large memory partition costs more service units than one CPU-hour on a normal compute partition, because the former is more costly to add to the cluster than is the latter. Furthermore, on our cluster the charge for using GPUs is based on the number of CPU cores used, rather than the number of GPU cores. We enforce that all jobs requiring GPUs must request at least two CPU cores for each GPU core requested. To handle these complications, we decided to implement Slurm plugins. We also needed to persist the accounting data generated by Slurm in a database that would be easily accessible to any number of clients, without impacting the functioning or performance of Slurm or its controller.

We developed and tested the plugins against Slurm 18.08.7. They should work in higher versions if the Slurm API stays consistent, but may require some minor modifications if it does not [4]. We also developed multiple web API endpoints that expose the accounting data stored in our database for access by various clients. The plugins and API specification are available on GitHub¹ under an open source license.

In the following sections, we present related work in the area of cluster usage policy enforcement, provide a high level overview of plugin and API design, discuss implementation details of both, and finally propose points of improvement as well as potential applications of and extensions to the system.

¹<https://github.com/ucb-rit/slurm-banking-plugins>

Figure 1: In this example, there are three users on a project (and each user could also be part of other projects). The usage for each user must not exceed the limit, and the total usage across all users must not exceed the project’s limit.

2 RELATED WORK

Before deciding on our approach, we investigated a number of alternatives that already existed.

The first choice would have been to take advantage of Slurm’s built-in accounting capabilities. However, we quickly found these to be insufficient for our purposes. Firstly, our service unit pricing model is more customized than that provided by Slurm. Secondly, Slurm charges on a per-account basis, whereas we needed a more granular way to divide units among individual users of an account, and charge them accordingly. Thirdly, Slurm only charges accounts after usage, but one of our design goals was to be able to preemptively block jobs that would exceed service unit allocations. Finally, we wanted to store the accounting data in a traditional database for other applications, but Slurm explicitly does not claim to store historical state [8]. Even if it did, we wanted to avoid placing a load on its database.

SLURM Bank is a collection of wrapper scripts that provides a simple banking system on top of slurmdbd [12]. In particular, it does prevent jobs that would exceed resource allocations from being started. However, allocation is based entirely on CPU hours, which was inadequate given our customized service unit pricing per partition model. And, as aforementioned, we wanted to avoid relying heavily on slurmdbd. Lastly, since the tool is comprised of wrapper scripts, users would need to learn new commands and modify their existing workflows.

Many clusters have existing solutions for usage policy enforcement, but these are often closed-source or highly specific to their own policies, workflows, and needs. For example, NERSC (National Energy Research Scientific Computing Center) also implemented Slurm plugins to modify job submission behavior based on enforcement limits. In its case, accounting data was stored in its NIM (NERSC Information Management) database, and scripts were developed to share information between NIM and Slurm, as well as to perform billing [7]. This implementation is specific to NERSC, so it would have been both difficult and inappropriate for us, and more importantly, for others, to port it for use.

By contrast, we wanted to develop a maximally general, modular approach that solves the base problem all clusters face: enforcing

resource usage limits at the account and user levels. In particular, it was important that it be modular—that is, clusters should be able to take advantage of one part of our design without having to adopt all of it. For example, any cluster that uses Slurm can implement our prescribed API (using our database models or their own), install the Slurm plugins, and configure them for their use case.

We considered open-source solutions that we could potentially extend, rather than starting from scratch. In particular, ColdFront is an HPC resource allocation system that provides a user portal for managing user and project data, including project allocations, with built-in Slurm integration [2]. Upon investigation, we found that it was missing a number of key features we needed for our design. Firstly, its interaction with Slurm is periodic: data need to be repeatedly synced between the two. We wanted our system to operate in real-time, preventing new jobs from being submitted once allocations are exceeded, without delay. Secondly, ColdFront’s database models did not support the notion of individual user-level sub-allocations within an account. We ultimately wanted to allow account managers to distribute their allocations amongst their users in arbitrary ways. Database models, and associated views, would need to be added or updated. Most fundamentally, we wanted to maintain an API layer around the database, so that data access would be centralized and uniform across all clients. ColdFront has no such API: its portal accesses data directly via Django’s model layer. We would need to build this API, and re-implement its portal logic using API layer calls rather than the lower-level model layer calls. In other words, whether or not we extended ColdFront, we would still need to implement Slurm plugins, various database models, an API, and additional user portal logic.

Without appropriate existing solutions to extend, and with the knowledge that many clusters have existing databases, we opted to design our own database models, making them as generic and straightforward as possible, so that others would be able to enforce their usage policies by implementing our plugins and API.

3 HIGH LEVEL DESIGN

3.1 Accounting

Resource usage is calculated and charged to users in terms of service units. The cost of a job is given by the rate for the requested partition times the number of CPU cores used by the job times the length of the job in hours. There may optionally be a quality of service (QoS) multiplier, but this is set to 1 by default. Each user may be a part of multiple projects, and each user-project pair has a maximum number of service units that may be used. In Slurm, the project is represented using the account field.

3.2 Life Cycle

We developed three Slurm plugins, and three accompanying web API endpoints, each for a different stage of a compute job's life cycle: submission, start, and completion.

- The *job submit plugin* runs when the user submits a job. It attempts to estimate the number of service units the job will cost, asks the web API if both the user and the account have sufficient funds, and allows or blocks the job accordingly.
- The *spank plugin* runs when the job starts running on a compute node. It sends preliminary job information to the web API, such as the job's Slurm ID, the job's start time, resource details, and the estimate of the job's cost in service units.
- The *job completion plugin* runs when the job completes. It updates the preliminary job information in the web API with the corrected values, including the actual number of service units charged, and fills in missing fields, such as the job's end time.

4 DESIGN DETAILS

4.1 Plugins

The plugins make use of Slurm's Plugin API. They are written in Rust (using its C foreign function interface) to help with memory safety and to leverage Rust's ecosystem of libraries, including OpenAPI codegen, which generates a client for the web API based on the OpenAPI Specification [11]. To interface our Rust plugins with the existing Slurm C code, we use rust-bindgen [13].

4.1.1 Job Submit Plugin. Slurm defines a job submit plugin API, which is invoked at the time a user submits a job. Critically, a job submit plugin is capable of rejecting job submissions; we use this functionality to reduce the likelihood that a user uses more than has been allocated.

First, the job submit plugin gathers the submission parameters necessary to make an estimate of the job's cost. For our formula, these parameters are: partition type, quality of service (QoS), maximum number of CPUs that may be used, and the time limit (the maximum amount of time the job could run).

The maximum number of CPUs is particularly subtle to compute. If specified, the submit plugin will use the number of CPUs per task times the number of tasks. However, if the partition is exclusive (jobs get access to the entire node), these values need not be specified, and the plugin assumes the most permissive case, estimating the cost to be 0. This could be improved by specifying the number

Figure 2: During a job's life cycle, the three plugins are invoked at job submission, job start, and job completion. If the job is predicted to exceed the account's or the user's allocation, the job is rejected at submission, before hitting the queue.

of CPU cores on each node in the exclusive partitions. This way, no valid jobs would be rejected, but it would be possible to use more service units than allocated. However, if the job cost was zero and the user had already gone over the quota, the job would be rejected.

After the estimate has been made, the plugin makes a call to the web API to check whether the job's estimated cost is allowable based on the user-project pair's resource usage. If the job is allowable, the plugin will allow the job to continue to the queue. If not, the job submission is blocked, and the user is informed of the error.

4.1.2 Spank Plugin. The spank plugin is invoked when the job begins running and has been assigned to compute nodes. The plugin extracts the job information using Slurm's `slurm_load_job` function. It sends the estimated job cost (in service units), job status (always "RUNNING" at this stage), partition name, quality of service, submit date, start date, list of nodes running the job (hostnames), number of CPUs, and the number of allocated nodes.

At this stage, the number of CPUs in use by the job is known, so an accurate estimate is sent to the web API and included in the user's usage. If the job happens to terminate early, this estimate would be high, and would later be corrected by the job completion plugin.

4.1.3 Job Completion Plugin. The job completion plugin runs when the job completes. Now that the job has finished running, it updates the job record in the web API with the missing and corrected values. Using the job's end timestamp to calculate the raw time the job has run, the plugin calculates the exact number of service units the user should be charged. It also updates the job status and populates the job's end time field.

4.2 API Specification

In the context of usage enforcement, the web API serves two functions: accounting and data persistence. During the life cycle of a job, the Slurm plugins make HTTP requests to the web API, providing them with the initial decision to allow or block the job, as well as a means to store and update job state.

At submission time, it determines whether or not the job, based on its expected cost, would overdraw either the allocation of its user or the allocation of its account. At run time, it generates a new Job object in the database and charges the associated account for its cost. At completion time, it updates the existing Job object and makes adjustments to account balances.

The API's underlying application and database models are written in Python 3, on top of the Django web framework [3]; the API itself is built on top of the Django REST framework [5].

4.2.1 Database Models. Data tables in the application's database back end are structured and manipulated through Django's model layer. This largely abstracts away the underlying SQL; in other words, the tables are defined in and migrated using Python code. Moreover, Django provides a built-in web interface that allows administrators to view and update database objects without code. For Savio, the Django application uses PostgreSQL as its database back end [10].

The following database models are involved in usage enforcement:

- The User model contains, among other user details, a Linux UID number unique to that user on the cluster, used to identify whom a job is being submitted by.
- The Project model contains its unique, identifying name, as well as the number of service units the project has been allocated for user consumption.
- The ActiveUserProjectLink model contains a pair of foreign keys, the first to a User instance and the second to a Project instance, marking the two as related. Also contained is the number of service units available to the user under the project.
- The ProjectUsage model contains the usage of a particular project; that is, it stores the number of service units used by that project thus far.
- The UserProjectUsage model serves the same role as the ProjectUsage model, but for a user-project pair.
- The Job model contains job details, including its Slurm ID, which user and project it is associated with, and its cost, expected or actual.

The web API assumes that users, projects, the associations between them, and other data are already loaded into the database. In Savio's case, various Google Sheets contain this information. A Django management command tailored to Savio's spreadsheets uses the Sheets API to extract rows of data from these spreadsheets, convert them into Django model objects, and load them into the database [1]. A second script extracts additional fields from the /etc/passwd file on the cluster.

4.2.2 Endpoints. The API is based on the Django REST framework, which was chosen for its ability to serialize Django models into HTTP-friendly JSON objects (and vice versa); its built-in support

for the HTTP CRUD operations (GET, POST, PUT, PATCH, and DELETE); and its browsable API view.

The API provides three endpoints for usage enforcement, each of which is called by a Slurm plugin:

- GET /can_submit_job/{job_cost}/{user_id}/{account_id} The job submit plugin makes an HTTP request to this endpoint. It returns whether or not the estimated cost of the job satisfies two criteria: adding it to the account's usage would not exceed the account's allocation, and adding it to the user's usage on the account would not exceed that user's allocation on the account.
- POST /jobs/ The spank plugin makes an HTTP request to this endpoint. It creates a new Job object with the given jobslurmid in the database, increases the submitting user's usage amount on the submitting account, increases the account's total usage amount, and returns the created Job.
- PUT /jobs/ The job completion plugin makes an HTTP request to this endpoint. It updates an existing Job object in the database, or, if none exists, creates a new one. If a matching object already exists, it calculates the difference between the job's expected cost, already stored in the object, and the job's actual cost, provided in the request. Because the expected is an upper bound on the actual, the difference is removed from the user's usage on the account. Finally, it returns the updated Job.

Each of these operations is performed within a database transaction to prevent other requests from modifying usage values mid-operation. In addition, if any part of an operation fails, all changes performed so far are also rolled back.

These endpoints require token authentication to prevent any non-administrator user from creating jobs, and by extension, modifying project and user usage balances. Each token is associated with a user, and when passed in the header of a request, identifies the request as originating from that user. Each token can be given an expiration time so that its use, and potential for misuse, is limited.

4.2.3 Drawbacks. Because a job is not assigned an ID by Slurm until run time, there is no way to create a Job in the database until it is already running. However, in order to block jobs, can_submit_job must be called by the job submit plugin at submission time. As a result, a project's usage might change after the GET request to can_submit_job but before the POST request to create the job. The decision was made to allow POST and PUT requests even if they would cause allocations to be overdrawn, since, by the time the requests are made, the jobs are already running and expending resources.

4.2.4 API Client Generation. The API conforms to the OpenAPI Specification, version 2.0. This was chosen both for documentation and for automatic generation of client libraries.

The following command generates a Rust client library from a JSON version of the OpenAPI Specification, which can be retrieved from the API, but is also available in the Slurm banking plugins repository.

```
openapitools/openapi-generator-cli generate \
-i /local/spec/swagger.json \
-g rust \
```



```
-o /local/rest_client \
--library=reqwest
```

Some work has also been done on using OpenAPI documentation to automatically generate web front-end code, which may be useful for future applications [9].

4.3 Usage Monitoring

The API gives us the ability to extend usage monitoring and administrative functionalities, which was one of our primary design goals.

For users, we developed a command-line tool that allows cluster users to check how many service units they have utilized so far and how many they have left. This instant feedback can enable them to better understand their limits and plan their future requirements and resource usage.

Cluster administrators can view all user-submitted jobs, usages for users and accounts, as well as any other database objects in Django's built-in admin interface.

```
$[~] ./check_usage.py -A ac_scsquest -s 2019-01-01
Usage for ACCOUNT ac_scsquest [2019-01-01,2020-02-11T17:18:00]: 1418 jobs, 78934.52 CPUHrs, 76893.25 SUs.
```

Figure 3: Cluster users can view job counts, CPU usage, user and account usages, and account allocations for a given time frame using the `check_usage` tool. This command shows usage details for the account `ac_scsquest` since January 1st, 2019.

With the existing as well as any future API endpoints acting as building blocks, more robust monitoring tools can be developed, such as an allocations dashboard that can detach cluster users from their dependence on cluster administrators.

4.4 Plugin Configuration and Installation

The job submit and job completion plugins need to be stored in `/usr/lib64/slurm`. The spank plugin can be placed in any location accessible to Slurm.

Slurm supports multiple job submission plugins, which can be added to `/etc/slurm/slurm.conf`:

```
JobSubmitPlugins=job_submit/slurm_banking
```

It supports only one completion plugin:

```
JobCompType=jobcomp/slurm_banking
```

And there may be multiple spank plugins. The spank plugin is enabled by the following line:

```
optional /etc/slurm/spank/spank_slurm_banking.so
```

The service unit rate for each partition (in service units per CPU-hour) can be configured in a config file:

```
[PartitionPrice]
# All partitions must be listed here.
# If a job requests to use a partition that is not listed
# here, it will be rejected.
# These numbers are the cost of one CPU-hour on the
# partition.
normal = 1
```

The screenshot shows a web form for viewing job details. The fields are as follows:

- JobslurmId:** 5503008
- Submitdate:** Date: 2020-02-05, Time: 18:03:57
- Startdate:** Date: 2020-02-05, Time: 18:03:57
- Enddate:** Date: (empty), Time: (empty)
- Userid:** sample@berkeley.edu
- Accountid:** co_sample
- Amount:** 2073.60
- Jobstatus:** RUNNING
- Partition:** savio2_bigmem
- Qos:** sample_bigmem2_normal

Figure 4: Cluster administrators can view jobs, usages for users and projects, and other database models in a web interface provided by Django. This view lists the details of a running job.

```
debug = 0.5
savio = 0.75
savio_bigmem = 1.67
savio2 = 1.00
savio2_bigmem = 1.20
savio2_gpu = 2.67
savio2_1080ti = 1.67
savio2_htc = 1.20
savio2_knl = 0.40
savio3 = 1.00
savio3_bigmem = 2.67
savio3_xlmem = 1.00
savio3_gpu = 1.00
```

4.5 Error Handling

The plugins and API are directly involved in accounting, so it is crucial that error incidence is minimized.

The primary potential cause of error is if the API service is unresponsive. We designed the plugins to be maximally permissive in this scenario, to avoid interrupting or blocking user jobs in production. The following cases describe what occurs should the API be down at each point in the job's life cycle:

- **Submission Time:** The job submit plugin attempts to connect to the API with a timeout of five seconds. If unable, it allows the job regardless of its cost. From the user's perspective, the worst case entails a five second delay. From the cluster's perspective, users with overdrawn allocations are temporarily able to submit jobs; in our view, this is less disruptive than blocking all users from submitting any job.
- **Run Time:** The job begins running, and its details are stored in slurmdbd, whether or not the API responds.
- **Completion Time:** The job completes, and its updated details are stored in slurmdbd, whether or not the API responds.

Below are some noteworthy cases:

- If the API is unavailable at run time but available at completion time, then the most up-to-date details of the job are sent to the API. No further correction is required.
- If the API is available at run time but unavailable at completion time, then the API contains only the job's estimated cost, which is an upper bound on its actual cost, and lacks some details, including its end time. These updated details need to be sent to the API.

Because job details are stored in slurmdbd regardless of the API's availability, discrepancies can always be reconciled after the fact. We developed a script, available in the plugins repository, that takes a file containing job entries, perhaps generated from `sacct`, and sends them in PUT requests to the API. As aforementioned, if the job does not already exist in the database, it is created; otherwise, the API computes the difference between the existing and new costs and updates the corresponding user and account usage values. This script can be placed in a cron job so that discrepancies exist for at most the periodicity of the cron job (e.g., one day).

5 RESULTS AND CONCLUSIONS

All three Slurm plugins have been developed and tested on a small test cluster running CentOS 7 and Slurm 18.08.7.

In production, we have deployed the `spank` plugin to UC Berkeley's Savio supercluster, which feeds job information to our API's database at the beginning of each job.

The web API is hosted on a virtual machine running Red Hat Enterprise Linux 7 in Lawrence Berkeley Laboratory's Science VM (SVM) infrastructure. It has received data for over 200,000 Savio jobs, where each entry includes the job's ID, submission time, start time, username, account name, partition type, and list of used nodes.

During a cluster maintenance period, we tested the job submit and job completion plugins successfully, and intend to deploy these in production, as well.

This work has given the administrators of Berkeley Research Computing's Savio cluster the ability to automatically and easily enforce our cluster usage policy. Cluster users also now have an easy way to monitor and keep track of their project usage. The approach we took of developing our own web API has been advantageous, as it has been and will continue to be extended to support future applications.

6 FUTURE WORK

While this system, driven by the interaction between Slurm plugins and a web API, already provides immediate value to our organization, there is much that can be improved and extended upon in future work.

The estimated job of a cost is added to a project's usage by the `spank` plugin, rather than by the job submit plugin. In other words, the project's balance is not updated until the job begins running. As a result, a user could potentially submit a large number of small jobs in a short duration. Each job would pass the checks performed by the job submit plugin, getting added to the queue. But the sum of their costs could overdraw either the user's allocation or the project's. We are investigating ways to ameliorate this issue at various locations in our stack.

When the job completion plugin runs, the job's status in Slurm's database is set to "COMPLETING", rather than "COMPLETED". This slightly-incorrect value is then propagated to the API's database, so it may be necessary to make a second pass over completed jobs to update their statuses.

If a user does not specify either `--ntasks` or `--cpus-per-task`, the job submit plugin's estimate for the job cost is zero. The job would be allowed to run as long as the corresponding project's and user's remaining allocation balances are non-negative. A better estimate can be computed if the requested partition is exclusive, meaning that entire nodes within the partition are allocated for the job, regardless of the number of tasks and CPUs requested. In this case, we can perform a lookup for the number of CPU cores each node has and use that in the estimate.

We described our capacity to correct discrepancies after the fact if the API is down during a job's life cycle, but we can implement more sophisticated validations. For example, if a job in the database is missing an end time or has an incorrect status, we can deduce that the API was down when the job completion plugin was invoked, and flag it for correction. Additionally, we are investigating more proactive approaches. Currently, the plugins merely log if the API is down; an alert would be preferable, so that it can be brought back up with minimal downtime.

Existing users and accounts are currently loaded into the database using a Django management command, which retrieves data from Savio's Google Sheets. In the future, new users and accounts can be stored directly in the API's database via a new set of REST-like HTTP endpoints which expose users, projects, and their associations. Specifically, we aim to use these and other endpoints to develop a user portal for Berkeley Research Computing.

In our current system, sub-allocations to individual users within a project are not dynamic. The simplest scheme for dividing a project's allocation would be to split it evenly among its users, but this is complicated if users are added to or removed from the project. Currently, a project user has access to all of the service units available to the project; once that amount is exceeded, no user in that project can submit any job. As a result, we are in the process of implementing an allocations dashboard that uses the web API as its back end. Using this dashboard, PIs will be able to manage their project allocations themselves, without having to depend on cluster administrators. In particular, they will be able to sub-allocate service units to individual users and view past jobs

and statistics. Finally, we are investigating visualization tools like Grafana to provide various data views to users and administrators [6].

ACKNOWLEDGMENTS

To Jacqueline Scoggins (Lawrence Berkeley National Laboratory HPCS Team), for participating in plugin testing. And to Amy Neeser (University of California, Berkeley) for feedback on technical writing.

To Berkeley Research Computing and Lawrence Berkeley Laboratory High Performance Computing Services teams for the initial design and motivation.

REFERENCES

- [1] Google Sheets API. [n.d.]. *Sheets API | Google Developers*. Retrieved Feb 3, 2020 from <https://developers.google.com/sheets/api>
- [2] ColdFront. [n.d.]. *ubccr/coldfront: HPC Resource Allocation System*. Retrieved Feb 3, 2020 from <https://github.com/ubccr/coldfront>
- [3] Django. [n.d.]. *The Web framework for perfectionists with deadlines | Django*. Retrieved Feb 3, 2020 from <https://www.djangoproject.com>
- [4] Slurm Documentation. [n.d.]. *Slurm Workload Manager - Documentation*. Retrieved Feb 3, 2020 from <https://slurm.schedmd.com/documentation.html>
- [5] Django REST Framework. [n.d.]. *Home - Django REST framework*. Retrieved Feb 3, 2020 from <https://www.django-rest-framework.org>
- [6] Grafana. [n.d.]. *Grafana: The open observability platform | Grafana Labs*. Retrieved Feb 3, 2020 from <https://grafana.com>
- [7] Douglas Jacobsen, James Botts, and Yun He. 2016. SLURM. Our Way. *Proceedings of the 2016 Cray User Group* (2016).
- [8] Morris Jette and Mark Grondona. 2003. Slurm: Simple Linux Utility Resource Management. In *Proceedings of ClusterWorld Conference and Expo*. San Jose, CA. Retrieved Feb 3, 2020 from https://slurm.schedmd.com/slurm_design.pdf
- [9] István Koren and Ralf Klamma. 2018. The Exploitation of OpenAPI Documentation for the Generation of Web Frontends. In *Proceedings of The Web Conference 2018*. Lyon, France. <https://doi.org/10.1145/3184558.3188740>
- [10] PostgreSQL. [n.d.]. *PostgreSQL: The world's most advanced open source database*. Retrieved Feb 3, 2020 from <https://www.postgresql.org>
- [11] OpenAPI Specification. [n.d.]. *OpenAPI Specification | Swagger*. Retrieved Feb 3, 2020 from <https://swagger.io/specification>
- [12] Jimmy Tang and Paddy Doyle. [n.d.]. *slurm-bank*. Retrieved Feb 3, 2020 from <http://jcftang.github.io/slurm-bank>
- [13] Jyun-Yan You. [n.d.]. *rust-bindgen: Automatically generates Rust FFI bindings to C (and some C++) libraries*. Retrieved Feb 3, 2020 from <https://github.com/rust-lang/rust-bindgen>