An Edge Service for Managing HPC Workflows

J. Taylor Childers
Argonne National Laboratory
Lemont, Illinois
jchilders@anl.gov

Thomas D. Uram
Argonne National Laboratory
Lemont, Illinois
turam@anl.gov

Doug Benjamin
Duke University
Durham, North Carolina
benjamin@phy.duke.edu

Thomas J. LeCompte
Argonne National Laboratory
Lemont, Illinois
lecompte@anl.gov

Michael E. Papka
Argonne National Laboratory
Lemont, Illinois
papka@anl.gov

ABSTRACT

Large experimental collaborations, such as those at the Large Hadron Collider at CERN, have developed large job management systems running hundreds of thousands of jobs across worldwide computing grids. HPC facilities are becoming more important to these data-intensive workflows and integrating them into the experiment job management system is non-trivial due to increased security and heterogeneous computing environments. The following article describes a common edge service developed and deployed on DOE supercomputers for both small users and large collaborations. This edge service provides a uniform interaction across many different supercomputers. Example users are described with the related performance.

CCS CONCEPTS

• **Computer systems organization** → *Interconnection architectures*; *Grid computing*;

KEYWORDS

Edge Service, HPC, supercomputer, HEP, workflow

ACM Reference Format:
J. Taylor Childers, Thomas D. Uram, Doug Benjamin, Thomas J. LeCompte, and Michael E. Papka. 2017. An Edge Service for Managing HPC Workflows. In *HUST'17: HUST'17: HPC User Support Tools, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3152493.3152557

1 INTRODUCTION

Many of the national science user facilities operated by the Department of Energy (DOE) Office of Science (SC) have significant data and computing needs that are set to exceed their current computing resources, a trend that will worsen in the coming years [5]. As an example, the Large Hadron Collider (LHC) at CERN is set to be upgraded between 2023 and 2025. The projected computing needs for the High Luminosity phase of the LHC (HL-LHC) is of order 100 times the current capabilities of the Worldwide LHC Computing Grid (WLCG or "the Grid"). To keep pace with this increase in computing, High Energy Physics (HEP) experiments are seeking computing resources beyond the Grid; leveraging High Performance Computing (HPC) is a key part of the strategy.

HPC centers offer a combined total of 15 billion core-hours per year: Mira at Argonne, a 48k node BlueGeneQ PowerPC system, delivers 6.8 billion core-hours per year; Cori at the National Energy Research Scientific Computing Center (NERSC), a 9,304 node Intel Xeon Phi (Knights Landing) and 1600 node Intel Haswell system delivers 5.5 billion core-hours annually; and Theta at Argonne, a 3,624 node Intel Xeon Phi (Knights Landing) system delivers two billion core-hours per year. LHC experiments currently use approximately three billion core-hours annually on the Grid. With DOE deploying Summit and Aurora at Oak Ridge National Laboratory and Argonne National Laboratory, respectively, a 20x increase in computing power, these resources are being targeted to accelerate HEP research goals.

The systems in place for managing compute jobs in large experiments exhibit significant diversity. Leadership computing facilities are comparably diverse user facilities. Integrating these systems requires solving several technical challenges, including authentication, interfacing with schedulers to submit jobs, managing input and output data transfers, and monitoring running jobs. This paper describes a service designed to ease the integration of supercomputers with experimental and observational science project computing infrastructure, by providing a uniform interface to HPC resources for automated job management systems.

2 THE HPC EDGE SERVICE

The HPC Edge Service is designed as a simple, adaptable interface between job management systems operated by experimental and observational science projects. It grew out of previous work to provide an interface for the production systems of large LHC collaborations [6–8]. The edge service is an open source project.

The edge service is primarily composed of two pieces, the Argo and the Balsam services. These services are Python-based and employ the Django [2] framework to utilize its database interface (for record keeping and robustness) and web authoring capabilities (for monitoring and user interface). Argo and Balsam are implemented as Django applications within a single Django project. A Balsam service running on a compute resource will be referred to as a Balsam site. A single Argo service coordinates sending work to many Balsam sites. In the examples used below, the RabbitMQ [4] message queue server is used for inter-service communication and a GridFTP [3] server is used for data movement, however, both of these functionalities are abstracted to make for easy replacement.

Both services receive work via a message queue server. Large production systems can submit jobs directly to the message queue whereas the underlying Django framework makes it natural to build a web-based interface for the individual user to define jobs. Administrators control access using the authentication tools of the communication service or Django web interface. RabbitMQ provides the ability to use SSL certificates to authenticate with the message queue which was used in the examples below.

Figure 1 shows an example workflow used by the ATLAS collaboration that will help to elaborate the components of the Argo and Balsam services and how they interoperate. The figure is numbered in order of these steps:

(1) The ATLAS production service stages input data to the data transfer server.
(2) The ATLAS production service sends a job message to the Argo input job queue.
(3) Argo receives the job message, which is composed of a two subjobs. The first subjob uses a small HEP HTCondor cluster, and the second uses the Mira supercomputer. In some cases, ATLAS would run the second subjob on Edison instead of Mira depending on availability.
(4) Argo sends the first subjob message to the Balsam running on the HTCondor cluster.
(5) Balsam stages in input data if needed.
(6) After the job completes, Balsam stages out data.
(7) Balsam sends a message to Argo that the job completed.
(8) Argo submits the second subjob to the Balsam running on the Mira supercomputer.
(9) Balsam stages in input data if needed.
(10) After the job completes, Balsam stages out data.
(11) Balsam sends a message to Argo that the job completed.
(12) Argo posts a message to ATLAS that the job has completed. In addition, Argo has been sending updates to ATLAS through this same message queue as the job changed state.
(13) ATLAS receives the completed message.
(14) ATLAS retrieves the output data from the server.

The following sections cover the Argo and Balsam services in more detail. The usage of the edge service by the ATLAS experiment is discussed further in Section 3.1.

2.1 Balsam

The Balsam service runs on the computing resource where jobs will be run and acts as a job database and an interface to the local batch scheduler. Balsam employs a modular design to allow adaptation to the particular technologies used at each compute site, as shown in Figure 2. The Balsam service is a multi-process Django application which runs as a background service, listening to the designated message queue for incoming jobs, submitting jobs according to user-defined limits, monitoring running jobs and sending status updates to the designated external message queues, and managing data transfers. The user can configure many parts of the Balsam service using standard Django configuration files.

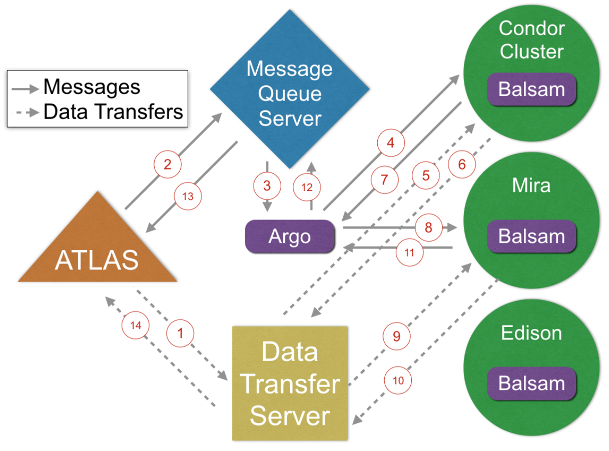


Figure 1: An example diagram in the case of the ATLAS collaboration event generation workflow. The numbers correspond to steps in the workflow which is described in the text.

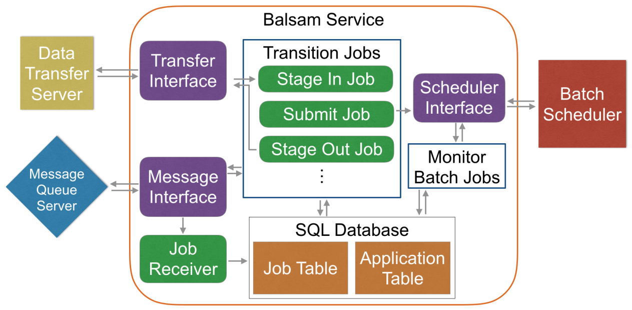


Figure 2: Diagram of the Balsam service. The Transfer, Message, and Scheduler interfaces are abstract class objects that can be replaced with the user's favorite tool. The Transition Jobs and Job Receiver are run as subprocesses in order to allow Balsam to be responsive and robust.

2.1.1 *Balsam Jobs.*

Balsam receives jobs via the message queue service and submits them to the local batch queue. Balsam manages an SQL database with two tables: a job table and an application table. Each job message is translated into a row of the job table. Jobs are represented using a Django model object which manages the database transactions. When Balsam receives a job message it converts the message text to a `BalsamJob` object. The job messages are defined by creating a Python dictionary containing specific key-value pairs, the dictionary is converted to text format following JSON formatting rules, and sent to the Balsam job queue. The possible keys are below.

- `name`: A name for the job given by the user.
- `description`: A description for the job given by the user.
- `site`: The name of the site where this job should run. This setting is used by Argo to determine which Balsam site to send the subjob and is ignored by Balsam.

- `argo_job_id`: The id of the Argo job to which this job belongs. This can be omitted if Argo is not being used. It is used by Balsam in sending job updates to Argo.
- `queue`: The name of the local batch queue to which to submit the job.
- `project`: The project name to charge against when submitting to a batch system.
- `wall_time_minutes`: The wall clock limit in minutes.
- `num_nodes`: The number of nodes to submit for this job.
- `processes_per_node`: The number of processes per node for this job.
- `scheduler_config`: An optional configuration file used to configure the batch submission command.
- `application`: The name of the application to run. This name must exactly match the `name` field of an application in the application table described below.
- `config_file`: The application configuration file which is passed as input to the `config_script` of an application as described in the next section.
- `input_url`: This is the URL from which to stage in data before running the job. When using Argo it sets this value.
- `output_url`: This is the URL to which to stage out data after running the job. When using Argo it sets this value.

`BalsamJob` objects have a state; when a job is first received it is stored in the database in the `CREATED` state. The Balsam service periodically checks for jobs in the database that are ready to transition to the next state. The service launches a subprocess that transitions the job to the next state by running the corresponding function in the `models.py` file. If a job fails at some point during its execution, it is marked with the corresponding failure state; for example, `SUBMITTED` fails to `SUBMITTED_FAILED`. Balsam uses the combination of the failure state and various job characteristics to determine whether to try to rerun the failed step (for example, if a transfer server is temporarily down), or whether the intermediate results should be retained (the output may still be useful and be transferred manually, even if the automated transfer is temporarily failing). The order of state progression and a description of each state is as follows:

- `CREATED`: The job has been stored in the database.
- `STAGED_IN`: If an input URL was defined, the folder has been copied into local storage.
- `PREPROCESSED`: If the application being executed on the batch system specifies a preprocessing step it has completed.
- `SUBMITTED`: The job has been submitted to the batch system.
- `QUEUED`: The job is currently queued on the batch system.
- `RUNNING`: The job is currently running on the batch system.
- `EXECUTION_FINISHED`: The job has finished running on the batch system.
- `POSTPROCESSING`: If the application being executed on the batch system specifies a post-processing step it has completed.
- `STAGED_OUT`: If an output URL was defined, the output from the job has been copied to this location.
- `JOB_FINISHED`: The job has completed successfully.

As jobs change state Balsam sends updates to a message queue for other processes, such as Argo, to monitor the state of jobs.

In addition to the programmatic submission and management interface, Balsam implements a command-line interface that is useful for interacting with the job, which can be run using the Django command `./manage.py <command>`. The command line interface includes the following commands:

- `balsam_ls_jobs`: This lists all the jobs in the job table.
- `balsam_rm_jobs`: This removes jobs from the job table.
- `balsam_alter_job`: This allows changes to existing jobs.
- `balsam_add_job`: This adds a job to the table.
- `balsam_rm_all_jobs`: This deletes all the entries in the job table.

The user can use the standard `-h` after the command name to get help messages on how to use these commands.

2.1.2 *Balsam Applications.*

The Balsam service has a second database table, the application table. This table defines the applications that Balsam can run, and includes necessary configuration for setting up and running the applications. The database is interfaced using a Django model class object like the Balsam jobs described above and is defined in the same Python module. When defining an application, the following parameters must be specified:

- `name`: The name of an application that can be run locally. It should be code friendly as it is specified in the job definition as described above.
- `description`: A description of the application. This can be human readable text.
- `executable`: The executable and path to run this application on the local system.
- `config_script`: The script which digests the input configuration file included in the job definition and can construct the command line for the application or perform other configuration needs.
- `preprocess`: A script that is run in a job working directory prior to submitting the job to the queue.
- `postprocess`: A script that is run in a job working directory after the job has completed.

Applications can be added to the database using the Django application command `./manage.py balsam_add_app`. Balsam will only execute jobs using applications in this table; because Balsam executes jobs defined by the external message queue, this "inclusive" model is used to constrain the possible jobs to those explicitly defined by the user. For example, if Balsam is configured to run only high energy physics event generation jobs, it will reject all other job requests. Another security precaution is the use of the `config_script`, which takes a configuration file from the user and uses the settings inside to determine the command line options passed to the application. Nothing provided by the user is directly executed on the command line to preclude command injection.

The Balsam command-line interface includes several commands that are useful to interact with the application table contents, which can be run using the command `./manage.py <command>`. The possible commands are:

- `balsam_ls_apps`: This lists all the applications in the application table.
- `balsam_rm_apps`: This removes applications from the application table.

Table 1: An example Balsam job.

`name`	= `'alpgenGeneration'`
`description`	= `'W-boson to 4 jet events at 13TeV using Alpgen'`
`site`	= `'mira'`
`queue`	= `'prod-short'`
`project`	= `'EnergyFEC'`
`wall_time_minutes`	= `30`
`num_nodes`	= `16384`
`processes_per_node`	= `64`
`application`	= `'alpgen_wjets'`
`config_file`	= `'alpgen.ini'`

Table 2: An example Balsam application.

`name`	= `'alpgen_wjets'`
`description`	= `'W-boson production with jets using Alpgen'`
`executable`	= `'/path/to/alpgen/wjet'`
`config_script`	= `'/path/to/alpgen/alpgen_cmdline.py'`
`preprocess`	= `'/path/to/alpgen/alpgen_presubmit.sh'`
`poseprocess`	= `'/path/to/alpgen/alpgen_postsubmit.sh'`

- `balsam_alter_app`: This allows changes to existing applications.
- `balsam_add_app`: This adds an application to the table.

2.1.3 *Example job and application.*

Tables 1 and 2 show examples of a job definition and an application description for a job run by the ATLAS collaboration on the Mira supercomputer at the Argonne Leadership Computing Facility (ALCF). The application defines where the executable is located and all the related scripts. The user provided `config_file` is passed to the `config_script` of the application which produces the command that will be executed in the batch job.

2.1.4 *Balsam Housekeeping.*

The Balsam service also performs housekeeping operations such as cleaning up old job directories and ensuring that subprocesses finish successfully or are restarted when they exit unexpectedly.

2.2 Argo

The Argo service manages a database of jobs to be run and oversees their execution, which it does by subscribing to a message queue to retrieve jobs, communicating job descriptions to Balsam sites for execution and monitoring, and managing transfer of input and output data. The architecture of Argo is shown in Figure 3. Argo accepts jobs from message queues to be run at specified Balsam sites. Whereas Balsam jobs contain only a single application, Argo jobs can be composed of multiple applications run in series on Balsam sites. The user can configure many parts of the Argo service using standard Django configuration files.

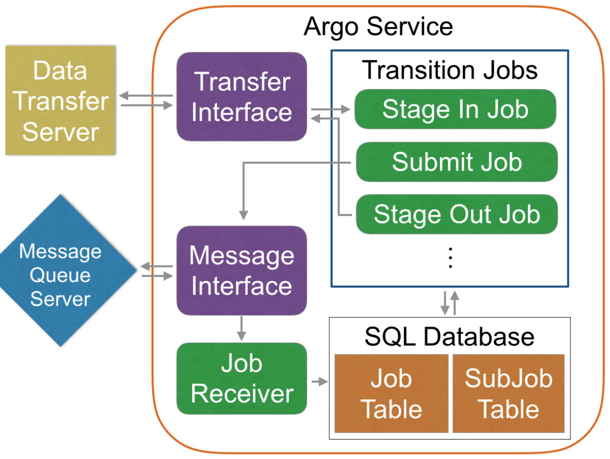


Figure 3: Diagram of the Argo service. The Transfer and Message interfaces are abstract class objects that can be replaced with the user's favorite tool. The Transition Jobs and Job Receiver are run as subprocesses in order to allow Argo to be responsive and robust.

2.2.1 *Argo Jobs.*

Argo receives jobs via the message queue service and manages their submission to Balsam sites, including the related data transfers. Argo uses an SQL database with two tables to record jobs. Argo jobs are represented by a Django model object, `ArgoJob`, which handles the database interaction similarly to Balsam. There is also an Argo subjob database table, represented as a model object. However the `ArgoSubJob` object is simply a copy of the `BalsamJob` object described above. Argo job messages are defined similarly to Balsam job messages, i.e., they are JSON-formatted Python dictionaries. The keys of the ArgoJob message are below.

- `user_id`: This is an identifier the user can define to help with tracking.
- `name`: This is a user defined job name.
- `description`: This is a user defined job description.
- `group_identifier`: This is a user defined value to help identify jobs that should be grouped together. It is mainly helpful for searching the jobs table.
- `username`: This is the username of the person who submitted the job.
- `email`: This is a comma separated list of email addresses which will be notified about the job status.
- `input_url`: This is the URL from where to stage in data before submitting the job to the first Balsam site.
- `output_url`: This is the URL to which to stage out data after the last job on the last Balsam site completes.
- `job_status_routing_key`: This is the message queue to which job status messages are sent. This can be used by users or production systems to keep track of Argo jobs.

- subjobs: A python list of Balsam job dictionaries as defined in Section 2.1.1.

Similar to Balsam, Argo jobs have states which Argo uses to track progress. The UserJobReceiver is a subprocess object launched by Argo to block on messages from the user submission message queue. Upon receipt, a job and its subjobs are saved to the database in the CREATED state. Argo jobs have a more complex state flow compared to Balsam jobs as depicted in Figure 4. There is a state loop for subjobs such that each subjob is processed through this loop and when all subjobs are completed the state SUBJOBS_COMPLETED is reached. A description of all states is below:

- CREATED: Jobs are initially saved to the database with this state.
- STAGED_IN: If an input URL was defined, the folder has been copied into local storage.
- SUBJOB_SUBMITTED: A subjob has been submitted to the destination Balsam site.
- SUBJOB_IN_PREPROCESSED: Argo received an update message from Balsam that a subjob is in the PREPROCESSED state.
- SUBJOB_QUEUED: Argo received an update message from Balsam that a subjob is in the QUEUED state.
- SUBJOB_RUNNING: Argo received an update message from Balsam that a subjob is in the RUNNING state.
- SUBJOB_RUN_FINISHED: Argo received an update message from Balsam that a subjob is in the EXECUTION_FINISHED state.
- SUBJOB_IN_POSTPROCESSED: Argo received an update message from Balsam that a subjob is in the POSTPROCESSED state.
- SUBJOB_COMPLETED: Argo received an update message from Balsam that a subjob is in the JOB_FINISHED state.
- SUBJOB_INCREMENTED: When a job reaches the SUBJOB_COMPLETED state, Argo checks if there are more subjobs. If yes their submission is triggered and the Argo job returns to the SUBJOB_SUBMITTED state. If no the job is moved to the SUBJOBS_COMPLETED state.
- SUBJOBS_COMPLETED: This state is reached when all subjobs have been completed.
- STAGED_OUT: If an output URL was defined, the output from the job has been copied to this location.
- HISTORY: This Argo job is completed.

2.2.2 Example Argo job. Table 3 gives an example of a real job run by the ATLAS collaboration on the Mira supercomputer at the ALCF. The primary piece of the job definition is the list of subjobs containing a site name which determines the message queue on which Argo places the message to route it to the correct Balsam site.

2.3 Data Transfer

The edge service supports data transfer plugins. Currently there are plugins for GridFTP, direct copy using cp, or Secure Copy using scp. The URL protocol determines which plugin is used: gsiftp:// for GridFTP, scp:// for scp, and local:// for cp. This module can be extended to include additional protocols. Credentials used in data transfer are specified, as needed, in the configuration file for Argo and Balsam.

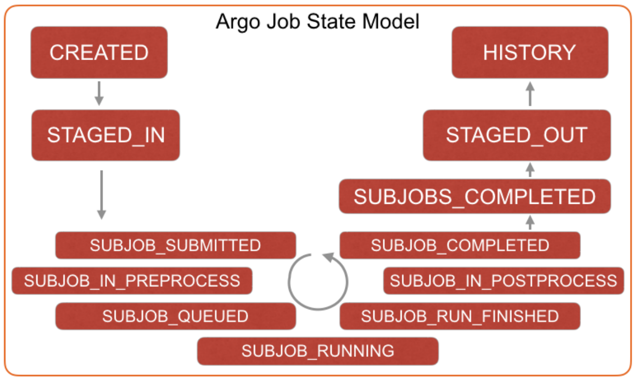


Figure 4: Argo job state flow. The circle of SUBJOB states will be executed for each subjob. Each state has a corresponding failure state which can be set if that state transition fails.

Table 3: An example Argo job.

name	= 'alpgenGeneration'
description	= 'W-boson to 4 jet events at 13TeV using Alpgen'
user_id	= 'alpgen.wjet.enu.5jets.13TeV.prodset001.job001'
group_identifier	= 'alpgen.wjet.enu.5jets.13TeV.prodset001'
username	= 'johndoe'
email	= 'johndoe@doe.gov'
input_url	= 'gsiftp://path/to/input'
output_url	= 'gsiftp://path/to/output'
job_status_routing_key	= 'atlas_job_monitor'
subjobs	= [{'name':'alpgenGenerator', 'description':'W-boson to 4 jet events at 13TeV using Alpgen', 'site':'mira', 'queue':'prod-short', 'project':'EnergyFEC', 'wall_time_minutes':30, 'num_nodes':16384, 'processes_per_node':64, 'application':'alpgen_wjets', 'config_file': 'alpgen.ini' }, { ... }]

2.4 Schedulers

Balsam can be configured to use different plugins for the batch scheduler interactions. The plugin implementations currently include HTCondor, Cobalt, Slurm, and Torque. Additional plugins can be implemented as needed.

2.5 Web Interfaces

The edge service is built in the Django framework to enable easy web interface construction for user communities. The creation of web pages which monitor and interact with jobs in the database is straightforward and well documented on the Django website. Figure 5 shows an example webpage constructed for browsing jobs submitted by the ATLAS experiment. This shows the benefit of the `group_identifier` as the user can locate jobs belonging to similar production configurations. Figure 6 shows an example of a second page displaying a single job description. In this example, the users wanted the ability to change the job state or even delete the current job from the database.

2.6 Security

As described above, Argo and Balsam together marshall jobs from external job databases to be executed on HPC resources. While providing this capability, we must also consider how it might be misused: access to the system should be strongly controlled, as should the applications that can be executed. In the ATLAS example, job descriptions are placed on a message queue, from which Argo retrieves them and passes them to Balsam, starting a process of execution, data transfer and communication of job states that finishes with a concluding message to the message queue. Communication between these components is secured with SSL using certificates issued by the CERN grid certificate authority. The message broker is configured with a whitelist containing only the certificate designated for this communication.

Balsam implements two safeguards to ensure that only approved applications are executed on target systems. The first is an application whitelist, included in the Balsam configuration; job descriptions designate applications by name, and Balsam dereferences the name according to the application whitelist to determine the executable. For job descriptions that specify applications not on the whitelist, Balsam simply logs the failure; the job is not run. The second safeguard avoids command injection with an abstraction of the command line, so users specify application options to Balsam, which then maps the options to command-line options. This approach prevents user-specified text from appearing explicitly on the command line.

3 APPLICATIONS

3.1 ATLAS Event Generation

The ATLAS Collaboration utilized the HPC Edge Service to run Alpgen event generation which was too costly to run on the LHC Grid computing resources. Their Argo jobs were described in Figure 1 and the accompanying text of that Section. It consisted of two Balsam jobs: the first was a serial processing step that ran on a five-node cluster running the HTCondor scheduler, while the second was a highly parallel step that ran on the Mira supercomputer at the ALCF or the Edison supercomputer at NERSC, depending on availability. This exhibits the capabilities of the service to run jobs across heterogeneous resources of varying sizes and architectures.

The HPC Edge Service has been deployed at Argonne since April 2015 to manage ATLAS event generation jobs on Mira and Edison. The usage from April 2015 to August 2016 is shown in Figure 7. Over the course of a year, the experiment ran 122M core-hours worth of work on Mira amounting to 2000 individual Argo jobs.

3.2 Fusion Experiments at DIII-D

Fusion energy experiments undertaken at the DOE DIII-D National Fusion Facility involve periodic (15-20 minutes) pulses of confined plasma, between which a collection of analyses are undertaken to shape magnetic coil configuration to better contain the next pulse. These analyses are typically executed on a local computer cluster at the fusion facility, with runtimes longer than the pulse period. The experimenters sought to reduce the analysis time to obtain results in time to plan for the next pulse, while also increasing the complexity and accuracy of the analysis. To do so, Balsam was deployed on ALCF resources to trigger jobs immediately upon availability of pulse data in the experiment database. The modular design of Balsam allowed the easy introduction of an alternative job source–in this case, the experiment database at DIII-D. By leveraging Balsam to trigger remote jobs (originating at DIII-D in California and running at Argonne in Illinois) on ALCF's compute resources, the scientists have more accurate results available to them timely enough that they can influence the configuration for the next pulse [10].

For this use case, Balsam communicates only with the experiment database; neither Argo nor a message queue is needed. Balsam subscribes to database updates, retrieves a pulse identifier when a pulse has finished, embeds the identifier in a job description and saves it to the Balsam jobs database. The Balsam service then submits the job to the scheduler. The underlying analysis application retrieves additional pulse data from the experiment database, runs the analysis, and transfers the results back to DIII-D. The relatively small data transfers are handled directly by the application, so Balsam is not involved in transferring data.

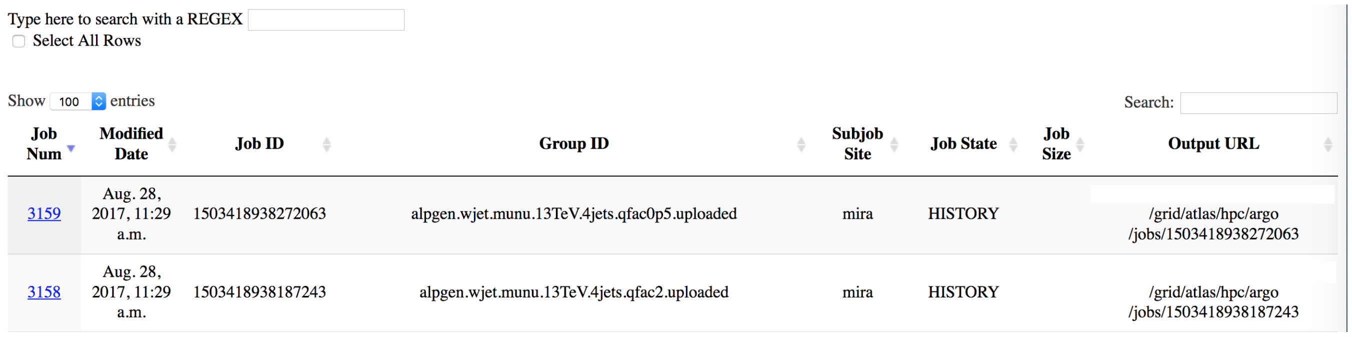
3.3 Related Work

The idea of providing a uniform interface to multiple computing sites with their disparate implementations of job schedulers, data transfer protocols, and monitoring capabilities is not new. The Globus Resource Allocation Manager (GRAM) established an API for job submission and data transfer and was widely deployed. The HPC Edge Service is a modern implementation of many of these same ideas in Python, which attempts to give users more flexibility and control over their interface to schedulers.

The ability of the HPC Edge Service to manage execution of multiple consecutive jobs on multiple compute resources is very much akin to that of workflow software, such as Pegasus [9], Taverna [11], or Airflow [1]. While the current edge service implementation supports only linear workflows, more complex applications will require support for complex job graphs, at which time a comparison with workflow software will be more apt.

4 SUMMARY

This proceedings describes the HPC Edge Service open source project. This project has successfully been deployed at the ALCF and NERSC facility for use by researchers in the HEP and Fusion sciences. The service performed well over the course of a year

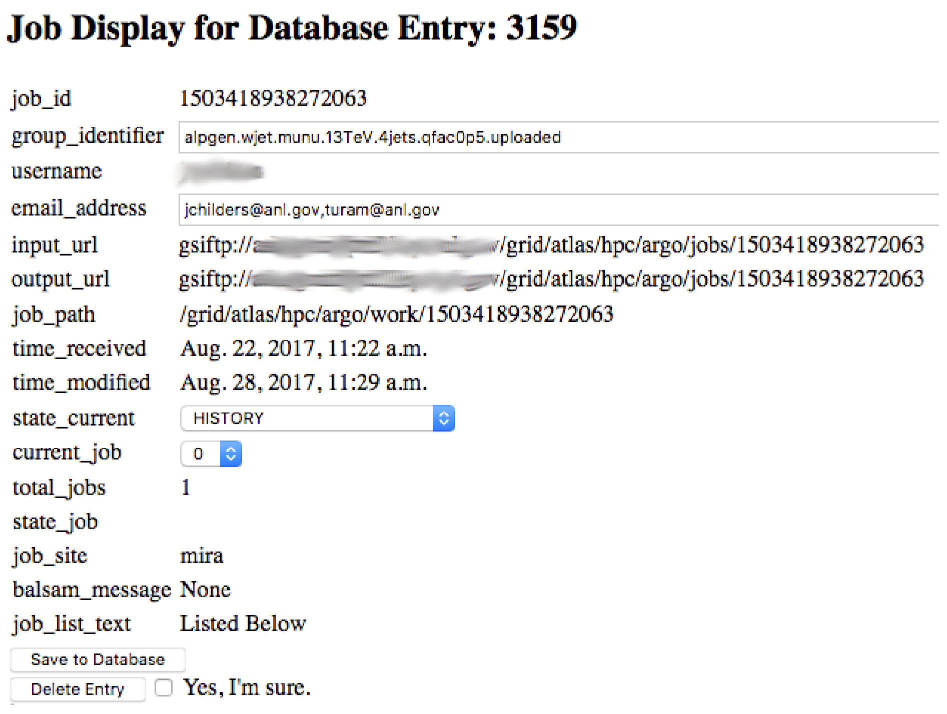


Type here to search with a REGEX
Select All Rows

Show 100 entries Search:

Job Num	Modified Date	Job ID	Group ID	Subjob Site	Job State	Job Size	Output URL
3159	Aug. 28, 2017, 11:29 a.m.	1503418938272063	alpgen.wjet.munu.13TeV.4jets.qfac0p5.uploaded	mira	HISTORY		/grid/atlas/hpc/argo /jobs/1503418938272063
3158	Aug. 28, 2017, 11:29 a.m.	1503418938187243	alpgen.wjet.munu.13TeV.4jets.qfac2.uploaded	mira	HISTORY		/grid/atlas/hpc/argo /jobs/1503418938187243

Figure 5: The Argo job table browser used by the ATLAS collaboration for managing jobs on Mira.



Job Display for Database Entry: 3159

job_id	1503418938272063
group_identifier	alpgen.wjet.munu.13TeV.4jets.qfac0p5.uploaded
username	[redacted]
email_address	jchilders@anl.gov,turam@anl.gov
input_url	gsiftp://[redacted]/grid/atlas/hpc/argo/jobs/1503418938272063
output_url	gsiftp://[redacted]/grid/atlas/hpc/argo/jobs/1503418938272063
job_path	/grid/atlas/hpc/argo/work/1503418938272063
time_received	Aug. 22, 2017, 11:22 a.m.
time_modified	Aug. 28, 2017, 11:29 a.m.
state_current	HISTORY
current_job	0
total_jobs	1
state_job	
job_site	mira
balsam_message	None
job_list_text	Listed Below

Save to Database
Delete Entry ☐ Yes, I'm sure.

Figure 6: An Argo job page for an ATLAS event generation job.

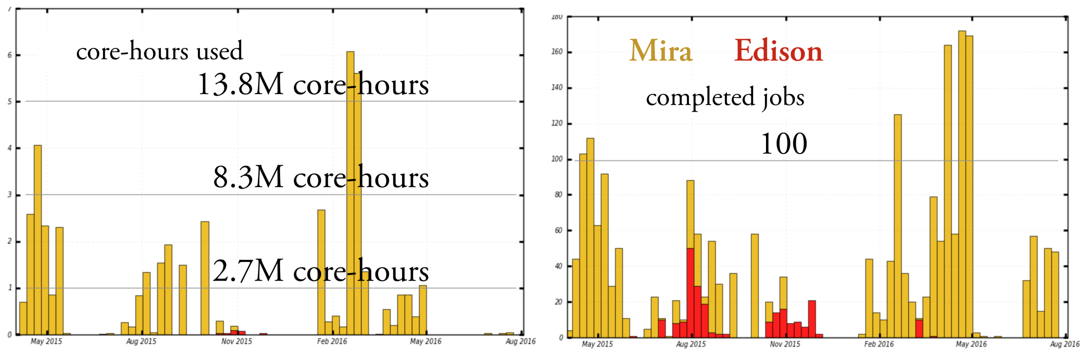


Figure 7: Left: Number of core-hours used on Mira and Edison. Right: Number of jobs run on Mira and Edison.

of running 2000 ATLAS event generation jobs using over 120M core-hours of supercomputer processing.

ACKNOWLEDGMENTS

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research was supported by an award of allocation time through the DOE ASCR Leadership Computing Challenge.

REFERENCES

[1] 2017. Airflow [software]. https://github.com/apache/incubator-airflow. (2017).
[2] 2017. Django [software]. www.djangoproject.com. (2017).
[3] 2017. GridFTP [software]. toolkit.globus.org/toolkit/docs/latest-stable/gridftp. (2017).
[4] 2017. RabbitMQ [software]. www.rabbitmq.com. (2017).
[5] E. Wes Bethel and Martin Greenwald (eds.). 2016. *Report of the DOE Workshop on Management, Analysis, and Visualization of Experimental and Observational data – The Convergence of Data and Computing.* Technical Report. Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 94720. LBNL-1005155.
[6] J.T. Childers, T.D. Uram, T.J. LeCompte, M. Papka, and D.P. Benjamin. 2015. Achieving production-level use of HEP software at the Argonne Leadership Computing Facility. *JOP: Conf. Series* 664, 6 (2015). https://doi.org/doi:10.1088/1742-6596/664/6/062063
[7] J.T. Childers, T.D. Uram, T.J. LeCompte, Michael Papka, and D.P. Benjamin. 2015. Adapting the serial Alpgen event generator to simulate LHC collisions on millions of parallel threads. 210 (11 2015). https://doi.org/doi:10.1088/1742-6596/664/9/092006
[8] J.T. Childers, T.D. Uram, T.J. LeCompte, M. Papka, and D.P. Benjamin. 2015. Simulation of LHC events on a millions threads. *JOP: Conf. Series* 664, 9 (2015). https://doi.org/doi:10.1088/1742-6596/664/9/092006
[9] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.
[10] M. Kostuk, T. D. Uram, T. Evans, D. Orlov, M. E. Papka, and D. Schissel. 2017. Automatic between-pulse analysis of DIII-D experimental data performed remotely on a supercomputer at Argonne Leadership Computing Facility. *Fusion Science and Technology, Special Issue on the Second IAEA Technical Meeting on Fusion Data Processing, Validation and Analysis* (2017).
[11] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, et al. 2004. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20, 17 (2004), 3045–3054.