Implementing a Flexible, Fault Tolerant Job Management System for Science Gateways

Dimuthu Wannipurage
dwannipu@iu.edu
Science Gateways Research Center
Indiana University
Bloomington, IN

Suresh Marru
smarru@iu.edu
Science Gateway Research Center
Indiana University
Bloomington, IN

Marlon Piece
marpierc@iu.edu
Science Gateway Research Center
Indiana University
Bloomington, IN

Eroma Abeysinghe
eabeysin@iu.edu
Science Gateway Research Center
Indiana University
Bloomington, IN

Sudhakar Pamidighantam
pamidigs@iu.edu
Science Gateway Research Center
Indiana University
Bloomington, IN

Marcus Christie
machrist@iu.edu
Science Gateway Research Center
Indiana University
Bloomington, IN

Gourav Shenoy
Gourav.Ganesh.Shenoy@ibm.com
IBM Watson Health
Cambridge, MA

Ajinkya Dhamnaskar
adhamnaskar@paypal.com
Microsoft
Redmond, WA

Lahiru Jayathilaka
lahiruj.14@cse.mrt.ac.lk
University of Moratuwa
Colombo, Srilanka

ABSTRACT

This paper summarizes our experiences evaluating and deploying a new task execution management system within the open source Apache Airavata framework for science gateways. We base our choices on our operational requirements and experiences running Airavata software as a multi-tenanted production service for multiple gateway clients. Our considerations include integrating semi-independent components, making major upgrades to those components while retaining the system's overall functionality, and choosing between integrating third party and in-house developed components. While we focus on Apache Airavata as the platform for evaluation, our results should be of general interest. After considering the options of extensions to our previous, in-house job management system using Apache Kafka or replacing it with Kubernetes, we ultimately chose Apache Helix, primarily for its ability to execute multiple tasks coupled into directed acyclic graphs. We have integrated this approach into Apache Airavata and have tested extensively over several months with many thousands of jobs, both from our internal throughput testing and operational tests with early adopter science gateway clients. The new system has proven to be at least as reliable as the previous system with the advantages that we now have simplified maintenance, do not need to support an in-house system that required extensive developer training to modify, and can support more sophisticated job execution scenarios.

PEARC '19, July 28-August 1, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-7227-5/19/07…$15.00
https://doi.org/10.1145/3332186.3332233

CCS CONCEPTS

• **General and reference** → **Reliability**; **Evaluation**; **Design**; • **Computer systems organization** → **Distributed architectures**.

KEYWORDS

Science gateways, job management, cyberinfrastructure, Apache Airavata

ACM Reference Format:
Dimuthu Wannipurage, Suresh Marru, Marlon Piece, Eroma Abeysinghe, Sudhakar Pamidighantam, Marcus Christie, Gourav Shenoy, Ajinkya Dhamnaskar, and Lahiru Jayathilaka. 2019. Implementing a Flexible, Fault Tolerant Job Management System for Science Gateways. In *Practice and Experience in Advanced Research Computing (PEARC '19), July 28-August 1, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3332186.3332233

1 INTRODUCTION

Science gateways are a central piece of the research cyberinfrastructure ecosystem, enabling thousands of scientists and students to make efficient use of scientific computing infrastructure [13] A common mode of operation for many science gateways is to provide user environments for executing scientific applications while shielding users from details such as how to manage the application's inputs and outputs, how to work with specific computing resources, and how to stage data on and off of a specific computing resource [2] [12] [17] [18] [26]. Following the definitions that we use in Apache Airavata [24], we refer to a user's request as a computational experiment and all of steps needed to execute the particular request as a process. A process can include one or more executions of jobs submitted to, for instance, a queuing system. Enacting a process requires executing multiple units of atomic tasks. Thus a single user's request is translated into multiple checkpointtable atomic tasks. Tasks include staging data, such as inputs and outputs,

submitting jobs and monitoring progress. In this paper we focus on distributed management of these low level tasks.

While integration with batch schedulers on computing clusters has been the traditional approach for executing tasks, gateways need to also manage more interactive submissions that are possible on cloud computing resources. Mixed-mode submissions are also important: a gateway may have large, computationally demanding jobs coupled with smaller post-processing pipelines. Given the large investment in cloud-based scheduling and execution management by companies such as Google and LinkedIn, and the fact that these software systems in many cases has been released as open source, we need to evaluate available task execution management solutions from open source communities and weigh these against developing an in-house solution.

This paper summarizes our experiences evaluating, integrating, and deploying a new task management system within the open source Apache Airavata framework [14] for science gateways. We operate Apache Airavata as the multi-tenanted Science Gateways Platform as a service (SciGaP) [23], so our choices are driven by operational requirements. As with other systems such as messaging [16], identity management [8], and sharing [19], we must consider carefully how to integrate many semi-independent components, how to make major upgrades to those components while retaining the system's overall functionality, and how to choose between integrating third party and in-house developed components. While we focus on Apache Airavata as the platform for evaluation, the results should be of general interest. For instance, other science gateway and scientific workflow developers can review our implementation and our evaluation methods.

2 RELATED WORK

There is a substantial body of publications on science gateways that manage the executions of tasks on high performance computers. The special journal issues introduced in [10] [21] [27] provide snapshots of the field. Several gateway software systems are reviewed and compared in [7]. General requirements of the field based on extensive surveys are discussed in [13].

As we discuss in this paper, executing a gateway request on a computational resource can be broken down into a sequence of tasks. Within a gateway experiment, multiple such jobs may need to be executed; these are typically encoded into workflows. These workflows vary in complexity from a single job to multiple jobs, which also include parallel steps. There is a substantial body of literature on scientific workflows, with [9] providing a recent survey.

Many science gateways support workflows. Prominent examples include Galaxy [2], which allows users to create and share executable workflows that can be composed with graphical user interfaces, and the HUBzero platform, which provides the ability to execute workflows using Pegasus [17]. In the first case, the workflow management system is part of the gateway software system. In the second case, the gateway outsources the execution of the workflow to an external workflow engine.

3 REQUIREMENTS FOR SCIENCE GATEWAY JOB MANAGEMENT

Apache Airavata falls into a class of science gateways that use a "shared nothing" approach to integrate with computing and storage resources. Apache Airavata middleware can be run on servers that are separate from the target computing and storage resources. This allows Airavata to support science gateways that need to deliver many backend computing resources to their user communities. It also allows us to run Apache Airavata as a centralized, hosted service for many gateways (such as [22]).

Apache Airavata already includes an job management component (GFac) that has been developed and maintained by the project over several years by a succession of developers. GFac is extensible to handle job submission using multiple methods and providers, such as GRAM [15] and UNICORE [5], but all production gateways that we currently support use SSH for remote execution on the target host.

While our current system has served many hundreds of thousands of jobs, it has also become a burden to support and difficult to improve and extend with new features that clients request. With an increasing number of gateways using the SciGaP platform, we have seen that there is a consistent growth of traffic coming into the current system and sometimes, there are bursts of requests arriving in a short period of time resulting in unexpected task execution failures or significant drops in throughput. In those cases, a new system should be able to provide an increased throughput by parallely submitting jobs to compute resources. However, the job management system should not naively transfer that load to compute resources as there are various limitations. For example, some compute resources have a maximum limit of remote SSH connections that can be created and some compute resources have a tight limit for maximum jobs that can be queued at a time. A system should be able to match input load and output load gracefully while maximizing throughput without dropping input traffic and overloading compute resources.

Reliability is another fundamental requirement of the new system. Some jobs that are submitted to compute resources are very expensive from the computational overhead perspective. So the job submission system should make sure that the jobs are submitted properly. If the system detects a failure such as a timeout on submission, retries should be done after verifying that they are not actually running in compute resources (that is, we need a two-phase commit procedure). When transferring input and output data from storage resources to compute resources and vice versa, the system should make sure that they were transferred properly, and if not, it should retry those failed transfers.

We also require a fast and reliable monitoring mechanism inside the task management system. The GFac system uses email-based monitoring where the middleware updates its view of the job's internal state based on email messages generated by the remote scheduling system (executing, completed, failed, etc). Email is very reliable (messages will eventually get delivered) and does not place a burden on the target resource like polling, but it has highly variable delivery times. We have observed delays of several minutes to hours between a job's completion on the target resource and the delivery of the "completed" email. This drastically reduces the throughput

and responsiveness of the entire task execution management system. In the new system, along with existing, reliable email-based monitoring, we need additional monitoring mechanisms that can decrease the time to detect state changes.

We expect the new system to do more than simply support current operations that we perform in Apache Airavata where we run well-defined, static workflows on computing clusters with job schedulers and resource managers. We have requirements from gateway clients to support heterogeneous workflows, where large computations may be coupled to lighter-weight pre- and post-processing steps that can be executed on different resources. Our requirements also include supporting different execution patterns, such as data-parallel applications as well as heterogeneous pipelines. We need to be able to do this for different clients using the same underlying, scalable infrastructure for multiple tenants simultaneously.

Finally, the system must be able to integrate diverse resources, including national scale infrastructure such as XSEDE in the US as well as international resources, university resources, local resources managed by the gateway clients themselves, and commercial cloud resources. The infrastructure must be able to accommodate the diverse usage and security requirements of these different systems. Our current approach within Apache Airavata supports over forty different computing resources within SciGaP operations, so we have substantial baseline requirements in this area for any new system.

While supporting existing deployment, we expected new system to be container-friendly where it consists of a set of small microservices communicating with each other and is as stateless as possible. This will increase the scalability and availability of the entire platform. Currently we deploy the Apache Airavata platform in bare metal servers; [25] explored the use of containers for Airavata.

Finally, we consider some non-functional requirements. First, we want a streamlined workflow engine that could be embedded and easily integrated as a component within Apache Airavata's distributed component framework. Second, since Apache Airavata makes releases through the Apache Software Foundation, we prefer to integrate software that uses Apache Software License or a compatible license [4]; incompatible licensed software cannot be included in releases, although such software can be a configuration option. Finally, we wanted to choose a system that has reasonable documentation and active online developer support.

4 A SIMPLIFIED VIEW OF TASK MANAGEMENT IN SCIENCE GATEWAYS

We first review the features we expect from a typical task management system and identify the common features that we can bring out that are already solved by others. The functionalities of a science gateway spans across multiple computer science subject domains, including user management, security, job management, data storage and warehousing, data visualization, application management, and metadata management for the digital objects created by the gateway users and administrators [1]. When it comes to task management, the gateway must manage the full life cycle of a task that is submitted to a compute resource. This includes setting up the runtime environment for the job to be executed, transferring input data to compute resources before job submission, submitting the job to compute resource once all the prerequisites are met, monitoring job status, fetching any outputs after the job completes, completing the rest of the post processing of the data, and cleaning up the environments. The task management system should be intelligent enough to retry or give up in case of an issue occurred in any of the above mentioned steps.

If we go further into fine grained details of the steps mentioned above, we can infer the following steps.

(1) The system should set up the proper runtime environment on the compute resource before transferring input data and job submission.
(2) Before the job submission, both the environment setup and input data staging should be completed.
(3) The system should wait until the job is completed before transferring output files to the gateway storage.
(4) Gateway level post-processing tasks should be performed after the output files are properly staged to gateway storage.

In summary, even the simplest tasks are workflows where a set of steps are executed with some dependencies. These workflows may be user initiated workflows or event-based workflows that will be triggered based on an event like the change of the status of a job. We must also consider data driven workflows specifically targeted to parse output data into user friendly formats and some machine learning workflows [22].

In summary, task management in a science gateway can be interpreted as a combination of workflows and a set of job monitors that emit events upon the change of status of jobs submitted to compute resources. We needed a workflow execution engine that could execute defined sets of steps. Those steps can be executed in parallel or in a serialy ordered Directed Acyclic Graph (DAG). In the DAG execution, involvement from Apache Airavata should be minimum and the system should be able to automatically recover from failures. The workflow management engine that we integrate with the rest of Airavata middleware should be scalable to multiple machines and should be deployable in cloud infrastructure. Following our "shared nothing" approach, we also wanted to retain our approach to place no additional requirements on the target compute resources. These need only to have a way to execute jobs via secure remote request and a way to securely handle file transfers.

5 SOFTWARE EVALUATIONS

Based on the evaluation criteria, we shortlisted three approaches for further evaluation.

5.1 Refactoring GFac With a Better Design

Our first attempt was to refactor GFac and outsource most of its critical parts that require high availability and scalability into third party software. In GFac, all of the task implementations were bundled into one server and communication between these tasks were highly coupled, so we decoupled these tasks into independent microservices and introduced a Kafka [**?**] message bus for communication purposes. Even though this simplified the solution, we had concerns about handling edge cases, such as what happens when something goes wrong in workflow execution and coordination of ordered task execution DAGs. To address these concerns, we

needed to introduce more parts in to the system, which made the system much more complex to maintain.

5.2 Using Kubernetes to Launch Tasks as Containers

Kubernetes [6] is a well-known open source container orchestration platform that can be deployed on various cloud platforms and on-premise servers. Even though Kubernetes itself is not a workflow execution engine, it has many functional and non-functional features that align with our requirements. From the functional perspective, we can use containers to abstract out tasks, we can implement tasks in any language we choose, and Kubernetes will run those task containers. But the downside is that Kubernetes can not run DAGs of tasks where execution ordering is required. We would need to develop our own custom orchestrator that observes already executed tasks and instruct Kubernetes on how and when the task container should be executed to preserve the DAG ordering. This introduces a significant additional level of complexity for us to implement and maintain.

5.3 Using Apache Helix Distributed DAG Execution Framework

Apache Helix [11] [3] is a generic cluster management framework. Helix assumes a distributed system as a state machine with number of constraints and transitions associated with it. Using this concept, Helix has come up with different out-of-the-box recipes to solve different distributed system problems like distributed lock management, replicated file stores, service discovery, and distributed DAG execution. Helix's DAG execution capabilities met most of our requirements.

6 HELIX-AIRAVATA INTEGRATION

When designing the new system, we extensively used the distributed DAG execution recipe of Apache Helix with some customization's to make it fit for our use cases, which we discuss below.

6.1 Wrappers to Simplify Task Definitions

Although Helix's Task API satisfied most of the technical requirements of a typical Apache Airavata Task, we still felt that we can improve it more in order to make it more developer friendly. In a basic Helix task, there are no built-in task level properties or variables. If you need one, you have to add it to the Helix's Configuration Map as a key value pair, and it's up to the task to read them from the Map and initialize the variables. Further, those variables should be String types. This approach works fine when we run tasks with less number of simple variables but still we have to manually fetch those values from the map.

To improve this, we introduced a wrapper class called AbstractTask that implements the original Task class of Helix. The advantage of having such abstraction is that any class that extends the AbstractTask class can define its own variables with its own types. AbstractTask takes care of the initialization of those variables using serialization and reflection. This approach is useful for developers who are proficient Java programmers but who are unfamiliar with the Helix framework.

6.2 Complex Object Messaging System

Helix only supports string values to be passed into tasks. However in our use case, tasks can have various types of data other than strings such as lists, maps, and objects defined by Apache Airavata data models [24]. One way to handle this case is to serialize all the values we need to transfer into to string types and deserialize them manually at the task level. This is a tedious and inefficient way to handle data inside tasks, and the developer is responsible for doing all those steps. To rectify this issue, we implemented an object serialization/deserialization module inside AbstractTask to support basic data types like int, long, double, boolean and any complex data type that is serializable. Through this, developers can directly work with actual data types, with the module taking care of all the serialization and deserialization work.

6.3 Workflow Management Agents

Workflow Management Agents are the entry points within Apache Airavata that create and execute Airavata level workflows. The responsibility of a Workflow Manager is to gather details of the requested workflow that is going to be created, generate the Helix DAG definition with specified variables for each individual tasks and job level dependencies, and launch the workflow to Helix execution engine. We have developed different Workflow Managers for running in Airavata to execute different workflows. These include the Pre-Workflow Manager, Post-Workflow Manager and Parsing Workflow Manager discussed below. A new workflow manager can be added to the system by extending the WorkflowManager class and it will inherit all the workflow manipulation operations in the framework.

7 JOB MONITORING

Once a job is submitted to a compute resource, the task management system should monitor the status of the job periodically and act upon each status. The lifecycle of a job can be illustrated as a state graph (Figure 1), and each state transition should be monitored by the task management system. In order to do that, there should be a monitoring component inside the task execution system. There are

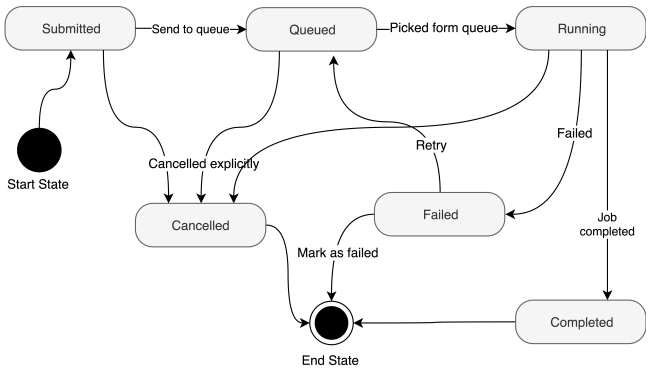

Figure 1: State Transition of a Job

two ways to implement monitoring: push and pull (or polling). In Airavata, we prefer push monitoring since it places less burden on the target computing resource, has no unnecessary communication, and can detect state changes more rapidly than polling. We consider two approaches to push messaging.

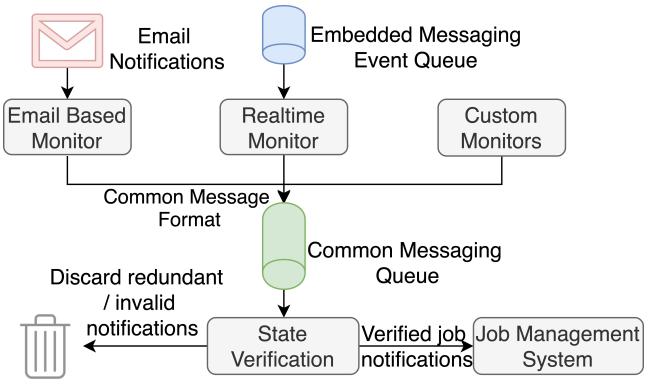

Figure 2: New Job Monitoring Framework

Job schedulers like SLURM and PBS on target compute resources send email updates to a designated email account upon the job state changes of each submitted job. One approach to monitoring is to utilize those email messages to feed the information to job monitoring system for further processing. This approach is more scalable than polling, reliable, and independent of the middleware's uptime; we have found it useful during upgrades and network failure recoveries. However, the downside of relying solely on email monitoring is that in some cases, email delivery becomes excessively slow. In extreme cases, we have observed email providers throttling bulk email deliveries, assuming that those are spam mails. Performance is a major concern in this mechanism

Another approach is to embed a signaling mechanism into the start and end of the job script that will trigger an external web hook and capture the events from that with near real time performance. The advantages of this approach is that it detects state changes much faster than email monitoring and requires no changes to the scientific applications themselves or to the scheduler/resource manager. The downsides are that we can only capture Running and Completed states this way and not error conditions, and we run the risk of missing messages.

If we can combine both approaches and come up with a hybrid solution, then we can reliably cover all the state transitions while keeping the improved performance for most scenarios. In the new system, we use this approach where jobs are monitored by two monitors. Since the two sources could potentially give conflicting states or introduce race conditions, we introduce a state verification phase that determines the correct state transitions based on the state machine mentioned above and last known states of the job; see Figure 2. This monitoring framework supports integrating with new monitoring agents seamlessly and all monitoring agents should follow the common messaging queue to publish their monitoring messages in a common message format.

8 DESIGN AND IMPLEMENTATION

The main design goals of the new system are to handover all of the task execution logic to Helix workflows and decouple job monitoring module from main workflows in order to support multiple job monitoring mechanisms.

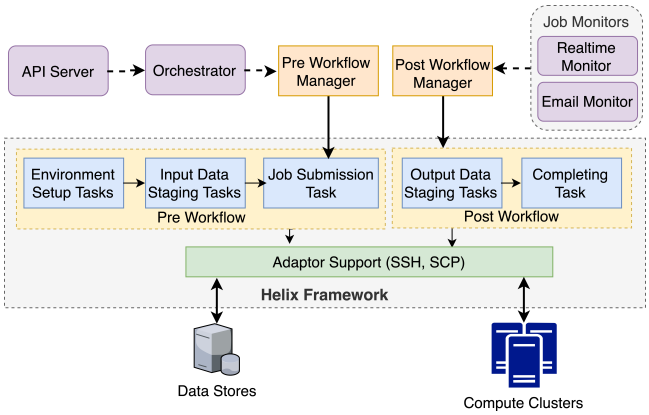

Figure 3: Architecture for workflow based job management. Dashed arrows represent inter-component communications through a messaging queue. Thick unidirectional arrows represent messages sent from Airavata into Helix Task Engine. Thick bidirectional arrows represent the SSH/SCP communication between Adaptors and Compute or Storage Resources.

Figure 3 summarizes our current approach. Rounded rectangles represent Apache Airavata components that we implement independently of Helix. Each component is an independent process that communicates over a network connection. Within the larger Helix Framework box, we implemented the Tasks in the figure using the AbstractTask extension discussed previously.

By analyzing the existing stack, we translated those tasks into two workflows called Pre-Workflow and Post-Workflow. Pre-Workflow contains tasks that should be executed until the job is submitted to the compute resource. Post-Workflow contains tasks that should be executed after Apache Airavata receives the job completion notification from the compute resource. Job monitors act in between these two workflows to monitor jobs that are submitted by Pre-Workflow in to compute resources and notify to Post-Workflow once they are completed, canceled, or exit with error conditions.

9 EVALUATION AND INITIAL OPERATION EXPERIENCES

To test the scalability and the accuracy of Apache Helix-based job system within Apache Airavata, we developed a load generating client that directly invokes Airavata's API endpoints to create and launch jobs. Using this client we could submit thousands of jobs per minute with different concurrency levels. During initial stress testing, from a total of 13,220 job submissions, 94.5% completed successfully, 2.0% were cancelled, 2.6% were failed and 0.9% were in created or executing state. During this testing, we noticed over the time a significant performance degradation on workflow execution.

Apache Helix uses Apache Zookeeper as its distributed coordinator and the storage for all workflow metadata. During debugging, we realized that Helix was not garbage collecting already completed workflows, and they were getting stacked up in Helix's internal Apache Zookeeper data paths, degrading performance. This issue resulted in the slowness of workflow execution as Helix performed unnecessary scanning iterations of already completed workflows. We contacted Apache Helix developers through the project's developer mailing list, and they instructed us to pass an expiry time for workflows when they are created in workflow managers. By introducing an expiry time, Helix automatically started to garbage collect already completed workflows which elapsed the expiry time, and the issue in performance was resolved.

9.1 Success to Failure Rates in a Real World Deployment

We were sufficiently confident in the system to begin testing with real loads from early-adopter SciGaP gateway clients. From 06/01/2018 to 02/01/2019, the system executed 12,088 experiments. 94.4% of this total were completed and 2.3% failed; the rest of the experiments were cancelled or in executing status. We compared above results with production results from the older GFac system for the same time period. For a total of 7,339 submitted experiments, 93.2% were completed, 3.9% failed, and 2.9% were cancelled. The majority of failures for both GFac and Helix-based jobs arose from three sources: a) too many jobs were submitted to a queue on a target resources, b) the resource scheduler was unresponsive within a timeout period, and c) we experienced network connection problems with the target resource. All of these errors can be reduced with further effort.

While we do not consider the increased success rate of the Helix-based system relative to GFac to be significant, we do have confidence that the new system is at least as reliable as the older system for our most basic use cases. More importantly, the Helix based system will enable new modes of operation and provide a more sustainable software base for future enhancements.

9.2 Performance

The capability of running multiple independent workflows in parallel is one of the key advantages of the new workflow framework. Based on the benchmarks we have performed, we achieved an average of 3000 job submissions per hour over different load conditions for the Echo application installed on a single node compute cluster with one Helix worker instance. As a percentage, it is a 83% improvement compared to our current job submission system; see Figure 4.

At the same time, due to the improvements in SSH connection pooling and a new adaptor framework, which simplified the connections to external resources from Apache Airavata, we were able to bring down the average time to submit a job by 40% (see Figure 5). The use of the new, streamlined framework simplified the implementation and testing of these improvements.

9.3 Job Monitoring Enhancements

We measured how new monitoring system reacts to the jobs submitted in to compute resources. Previously, all of the job monitoring

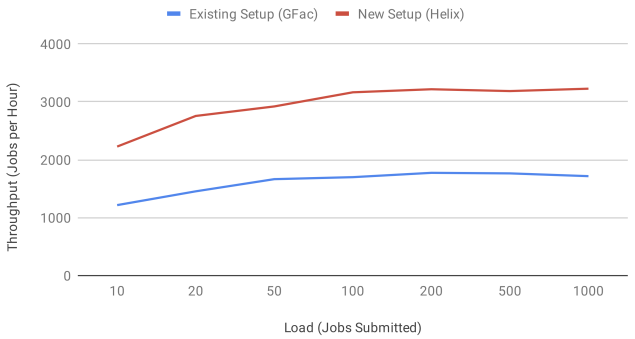

Figure 4: Job Submission throughput in different load conditions.

Figure 5: Comparison of average time to submit a job in different load conditions.

was handled by email-based monitoring. Now we added real-time monitors in to the new system.

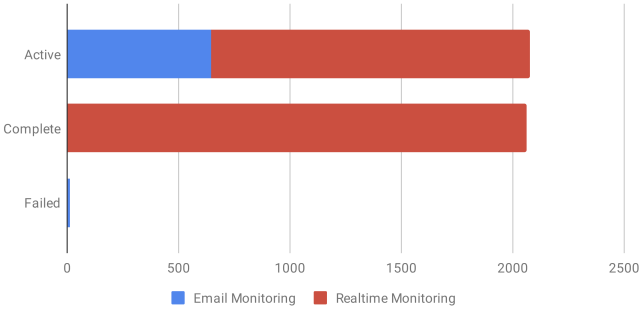

Figure 6: Contribution of each monitoring system with respect to notification type.

These are implemented as callbacks embedded in the cluster submission scripts that we generate. Callback messages are managed by an Apache Kafka broker that we operate as part of the SciGaP system. With the addition of a real-time monitoring capability, among a total of 2077 jobs submitted, 100% of the job completion notifications and more than 60% of job activation notifications were captured by real-time monitoring. Email-based monitoring captured

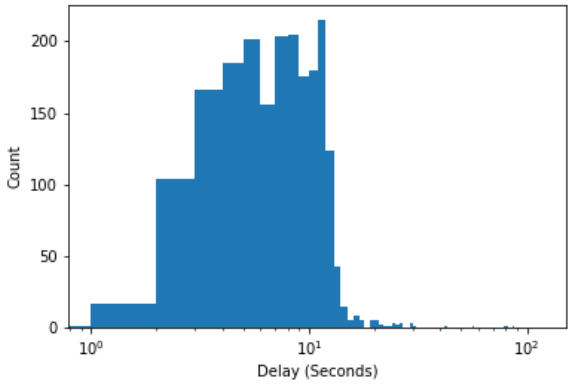

Figure 7: Time difference histogram of email monitoring and real-time monitoring for job completion notifications. Data set was clipped at 90th percentile to have a better visualization in common cases.

all of the failure notifications and 40% of job activation notifications; see Figure 6. This was consistent with our design goal to capture best-case scenarios where job statuses are getting updated as soon as possible while not losing edge cases like job failures. Further, we measured the time we saved by introducing real-time monitoring to capture job completion notifications (see Figure 7). In 90% of the cases, the delay for email-based events compared to real-time events was less than two minutes; 10% took longer than two minutes, with a maximum delay of 98 minutes. This may happened due to the mail delivery delays or intentional throttling performed by email providers to prevent DDoS attacks on their services.

10 ADVANCED USE CASES

Although we successfully reproduced Apache Airavata's task management system using the new workflow approach and introduced several improvements to the basic use case, our bigger goal is to extend its capabilities into much wider context.

As an example, we have client requirements to support post processing pipelines for the completed jobs in order to derive useful information from job outputs. These outputs are in application specific formats that can be text or binary. Many applications have third party parsing utilities available, which can be used to parse job outputs. We have already solved this issue by wrapping these parsing utilities as Docker containers and invoking these containers once a job is completed and then saving the output of each container execution in a MongoDB database for future usages. The SEAGrid Data Catalog [**?**] is currently benefiting from this approach to parse and visualize output data for Gaussian jobs submitted to SEAGrid Science Gateway.

We need to generalize this beyond the SEAGrid use case, integrate it with our core task management system, and apply it to other gateway use cases. Under the previous approach, if we want to integrate a new parser for new application and store the parsed data in a new data source or upload them in to the cloud, this had to be handled case by case. We solved this issue easily using the new

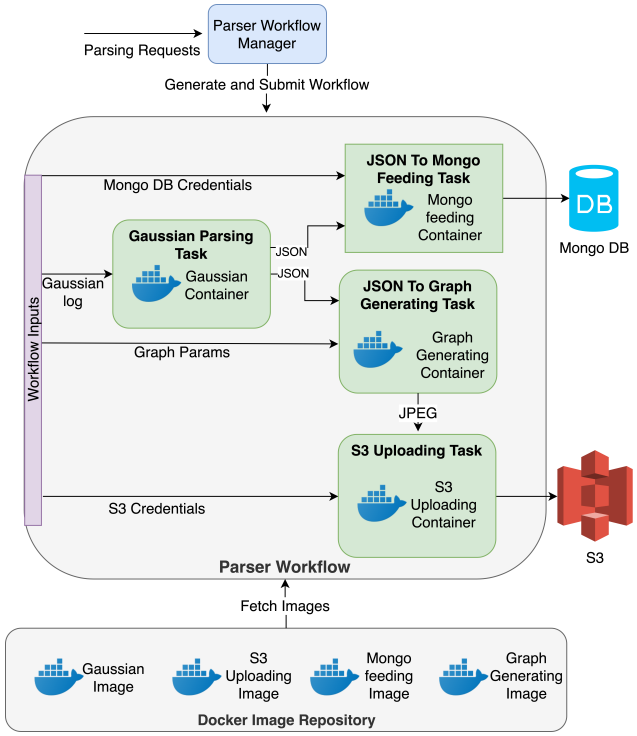

Figure 8: Workflow-based Data Parsing Framework.

workflow framework by introducing dynamic data parsing workflows; see Figure 8. In this approach, users can define data parsing docker images for target application and target storage mechanism. Each Helix Task encapsulates a Docker container execution. Then data parsing framework dynamically generates the workflow for that specific use case and executes each containerized application as a series of Helix Tasks. Another advantage we gained from this approach is that we can interleave multiple parsers into one workflow where we can easily perform data conversion at different stages.

11 CONCLUSIONS AND FUTURE WORK

Using a new workflow based approach, we were able to replace Apache Airavata's in-house developed task execution system with one base on Apache Helix, with some straightforward extensions. Under operational testing with real client gateway jobs, the new system performed at least as well as the old system for our basic use cases. With the streamlined framework, we were also able to make several improvements that would have been cumbersome to implement in our older approach. These include increased throughput (Figure 4), reduced overhead under load through SSH connection pooling (Figure 5), and decreased time to detect state change events in a remotely executing job (Figure 6).

More importantly, the use of Helix enables us to tackle more complicated task execution scenarios. We were able, for example, to re-implement the data parsing scheme introduced in [22] [20] using our new task execution system; previously it had been a standalone system developed for a single gateway.

The Helix-based task management system should make Apache Airavata more scalable under loads from multiple tenants. Helix is designed to scale out to multiple, cooperating instances that can balance loads of incoming tasks and that can withstand failures of individual instances. Kubernetes can be used to manage these containers, simplifying deployment.

Finally, we are experimenting with different approaches for extending this generic workflow capability into more user friendly and dynamic workflow defining methods. One immediate goal is to support application level workflow mechanisms where multiple scientific applications work together as a workflow by reusing one's outputs as inputs for others.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation award number 1339774.

REFERENCES

[1] 2018. *Towards a Science Gateway Reference Architecture.* http://ceur-ws.org/Vol-2357/paper6.pdf

[2] Enis Afgan, Dannon Baker, Bérénice Batut, Marius Van Den Beek, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Björn A Grüning, et al. 2018. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic acids research* 46, W1 (2018), W537–W544.

[3] Apache Software Foundation 2019. *Apache Helix Task Framework.* Apache Software Foundation. https://helix.apache.org//0.8.2-docs/tutorial_task_framework.html.

[4] Apache Software Foundation 2019. *ASF 3RD PARTY LICENSE POLICY.* Apache Software Foundation. https://www.apache.org/legal/resolved.html.

[5] Krzysztof Benedyczak, Bernd Schuller, Maria Petrova-El Sayed, Jedrzej Rybicki, and Richard Grunzke. 2016. UNICORE 7 a Middleware services for distributed and federated computing. In *2016 International Conference on High Performance Computing and Simulation (HPCS).* 613–620.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *ACM Queue* 59, 5 (2016), 50–57.

[7] Patrice Calegari, Marc Levrier, and PaweÃĚÂĆ BalczyÃĚÂĎski. 2019. Web Portals for High-performance Computing: A Survey. *ACM Transactions on The Web* 13, 1 (2019), 1–36.

[8] Marcus A Christie, Anuj Bhandar, Supun Nakandala, Suresh Marru, Eroma Abeysinghe, Sudhakar Pamidighantam, and Marlon E Pierce. 2017. Using Keycloak for Gateway Authentication and Authorization. (2017).

[9] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D. Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey S. Vetter. 2018. The future of scientific workflows. *International Journal of High Performance Computing Applications* 32, 1 (2018), 159–175.

[10] Sandra Gesing and Nancy Wilkins-Diehr. 2015. Science gateway workshops 2014 special issue conference publications. *Concurrency and Computation: Practice and Experience* 27, 16 (2015), 4247–4251.

[11] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, Adam Silberstein, Kapil Surlaker, Ramesh Subramonian, and Bob Schulman. 2012. Untangling cluster management with Helix. In *Proceedings of the Third ACM Symposium on Cloud Computing.* 19.

[12] Gerhard Klimeck, Michael McLennan, Sean P Brophy, George B Adams III, and Mark S Lundstrom. 2008. nanohub. org: Advancing education and research in nanotechnology. *Computing in Science & Engineering* 10, 5 (2008), 17.

[13] Katherine A Lawrence, Michael Zentner, Nancy Wilkins-Diehr, Julie A Wernert, Marlon Pierce, Suresh Marru, and Scott Michael. 2015. Science gateways today and tomorrow: positive perspectives of nearly 5000 members of the research community. *Concurrency and Computation: Practice and Experience* 27, 16 (2015), 4252–4268.

[14] Suresh Marru, Lahiru Gunathilake, Chathura Herath, Patanachai Tangchaisin, Marlon Pierce, Chris Mattmann, Raminder Singh, Thilina Gunarathne, Eran Chinthaka, Ross Gardler, et al. 2011. Apache airavata: a framework for distributed applications and computational workflows. In *Proceedings of the 2011 ACM workshop on Gateway computing environments.* ACM, 21–28.

[15] Suresh Marru, Srinath Perera, Martin Feller, and Stewart G. Martin. 2008. Reliable and Scalable Job Submission: LEAD Science Gateway's Testing and Experiences with WS GRAM on TeraGrid Resources. (2008).

[16] Suresh Marru, Marlon Pierce, Sudhakar Pamidighantam, and Chathuri Wimalasena. 2015. Apache airavata as a laboratory: architecture and case study for component-based gateway middleware. In *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models.* ACM, 19–26.

[17] Michael McLennan, Steven Clark, Ewa Deelman, Mats Rynge, Karan Vahi, Frank McKenna, Derrick Kearney, and Carol Song. 2015. HUBzero and Pegasus: integrating scientific workflows into science gateways. *Concurrency and Computation: Practice and Experience* 27, 2 (2015), 328–343.

[18] Mark A Miller, Wayne Pfeiffer, and Terri Schwartz. 2010. Creating the CIPRES Science Gateway for inference of large phylogenetic trees. In *2010 gateway computing environments workshop (GCE).* Ieee, 1–8.

[19] Supun Nakandala, Suresh Marru, Marlon Piece, Sudhakar Pamidighantam, Kenneth Yoshimoto, Terri Schwartz, Subhashini Sivagnanam, Amit Majumdar, and Mark A Miller. 2017. Apache Airavata Sharing Service: A Tool for Enabling User Collaboration in Science Gateways. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact.* ACM, 20.

[20] Supun Nakandala, Sudhakar Pamidighantam, Suresh Marru, and Marlon Pierce. [n. d.]. Better Data Discoverability in Science Gateways. ([n. d.]).

[21] Silvia Delgado Olabarriaga and Nancy Wilkins-Diehr. 2016. GCE15 Special Issue Conference Publications. *Concurrency and Computation: Practice and Experience* 28, 7 (2016), 1949–1951.

[22] Sudhakar Pamidighantam, Supun Nakandala, Eroma Abeysinghe, Chathuri Wimalasena, Shameera Rathnayaka Yodage, Suresh Marru, and Marlon Pierce. 2016. Community Science Exemplars in SEAGrid Science Gateway. *international conference on conceptual structures* 80 (2016), 1927–1939.

[23] Marlon Pierce, Suresh Marru, Eroma Abeysinghe, Sudhakar Pamidighantam, Marcus Christie, and Dimuthu Wannipurage. 2018. Supporting Science Gateways Using Apache Airavata and SciGaP Services. In *Proceedings of the Practice and Experience on Advanced Research Computing.* ACM, 99.

[24] Marlon Pierce, Suresh Marru, Borries Demeler, Raminderjeet Singh, and Gary Gorbet. 2014. The apache airavata application programming interface: overview and evaluation with the UltraScan science gateway. In *Proceedings of the 9th Gateway Computing Environments Workshop.* IEEE Press, 25–29.

[25] Pankaj Saha, Madhusudhan Govindaraju, Suresh Marru, and Marlon E. Pierce. 2016. Integrating Apache Airavata with Docker, Marathon, and Mesos. *Concurrency and Computation: Practice and Experience* 28, 7 (2016), 1952–1959.

[26] Subhashini Sivagnanam, Amit Majumdar, Kenneth Yoshimoto, Vadim Astakhov, Anita Bandrowski, Maryann E Martone, and Nicholas T Carnevale. 2013. Introducing the Neuroscience Gateway.. In *IWSG.*

[27] Nancy Wilkins-Diehr, Sandra Gesing, and Tamas Kiss. 2015. Science gateway workshops 2013 special issue conference publications. *Concurrency and Computation: Practice and Experience* 27, 2 (2015), 253–257.