A General Approach to Real-time Workflow Monitoring

Karan Vahi*, Ian Harvey†, Taghrid Samak‡, Daniel Gunter‡, Kieran Evans†, David Rogers†, Ian Taylor‡, Monte Goode‡, Fabio Silva§, Eddie Al-Shakarchi†, Gaurang Mehta*, Andrew Jones†and Ewa Deelman*
*USC Information Sciences Institute, Marina Del Rey, California
Email: vahi,gmehta,deelman@isi.edu
†School of Computer Science, Cardiff, UK
Email: i.c.harvey,k.evans,d.m.rogers,Ian.J.Taylor,e.alshakarchi,Andrew.C.Jones@cs.cardiff.ac.uk
‡ Lawrence Berkeley National Laboratory, Berkeley, CA
Email: tsamak,dkgunter,mmgoode@lbl.gov
§ University Of Southern California, Los Angeles, CA
Email: fabio.silva@usc.edu

***Abstract*—Scientific workflow systems support different workflow representations, operational modes and configurations. However, independent of the system used, end users need to track the status of their workflows in real time, be notified of execution anomalies and failures automatically, perform troubleshooting and automate the analysis of the workflow to help categorize and qualify the results. In this paper, we describe how the Stampede monitoring infrastructure, which was previously integrated in the Pegasus Workflow Management System, was employed in Triana in order to add generic real time monitoring and troubleshooting capabilities across both systems. Stampede is an infrastructure that attempts to address interoperable monitoring needs by providing a three-layer model: a common data model to describe workflow and job executions; high-performance tools to load workflow logs conforming to the data model into a data store, and a querying interface for extracting information from the data store in a standard fashion. The resulting integration demonstrates the generic nature of the Stampede monitoring infrastructure that has the potential to provide a common platform for monitoring across scientific workflow engines.**

I. Introduction

In many scientific applications, workflows are used to describe the relationships between individual computational components [14]. Scientific workflows enable the formalization of the computations, exposing individual computational steps, and data and control dependencies between them. They allow scientists to focus on the logical structure of the overall computation rather than on the low-level implementation details. Representing the computation as a workflow makes updating and maintaining an application easier than modifying a complex script that represents the same computation.

Different workflow systems address different needs. For example, workflow systems such as Triana [24] , Taverna [29] and Kepler [5] focus on presenting a graphical user interface for executing workflows locally or on a distributed set of resources. Other workflow systems such as Pegasus [12], focus more on large-scale workflows that are usually executed on a distributed set of resources and present a command line driven interface to users. However, for all workflow systems, users can reasonably expect to be able get real-time status information, troubleshoot their workflows and detect any anomalies that may occur during execution. They are also interested in a common set of performance metrics such as how long the workflow ran, how many jobs it had, the breakdown of jobs by various types, and compute resource usage.

Most workflow systems have addressed this need by building their own specific monitoring infrastructure [32] or putting logs into a database *post mortem* and then performing the analysis [9]. These approaches duplicate effort that could be more efficiently combined to build a common analysis infrastructure.

A system that attempts to address these concerns and provide a common data model for filtering monitoring information across runs is the NSF-funded Synthesized Tools for Archiving Monitoring Performance and Enhanced DEbugging (Stampede) [22], [37] project. In this project, the authors have developed a framework that uses a general data model to represent workflows and their execution. Stampede decouples the representation of data in the workflow and job log files from the process of populating this data to a central store. This decoupling provides the potential to use the Stampede framework for loading logs and analysis across workflow systems.

The contributions of the present paper are:

- *Final Stampede Data Model* - Stampede is a complete workflow data model for representing the performance characteristics of distributed workflows. In our initial implementations, we only modeled the output (executable) workflow. However, workflow systems may restructure the workflow for performance improvements at runtime, and further extensions to the STAMPEDE data model were required to accommodate this. Hence, our final model supports both the input and the output workflows enabling us to answer questions in reference to both the input tasks defined by the user in their workflows, and the runtime information in the output executable workflow.
- *Demonstrate the generic nature of the Stampede Ap-*

proach - The integration of Stampede and Pegasus has been described in our earlier work [22]. The present paper demonstrates the generic nature of the Stampede approach by describing the integration with the independently developed Triana workflow engine.
- *Overview of performance metrics and trouble shooting capabilities* that Stampede tools provide to the users of workflow systems, once the integration is done.

The paper provides an overview of related work and an introduction to the Triana Workflow System; it then describes the Stampede data model in Section IV. The Stampede and Triana integration is described in Section V. We conclude the paper by describing a music retrieval workflow modeled in Triana and demonstrate how Stampede tools facilitate monitoring, troubleshooting and mining of performance metrics.

II. Related Work

Workflows present compact operational models of complex domain processes, which can assist substantially in the understanding and learning of these processes. A workflow can be used to systematically break a complex problem into a tractable set of components that support the entire research lifecycle, which promotes a general methodology for disseminating good research practices and their reuse across institutions, problem domains and disciplines.

There are numerous workflow systems in use within the scientific community, for example, ASKALON [16], Kepler [5], Pegasus [12], MOTEUR [21], P-Grade [26], Taverna [29], Triana [24] and Trident [7]. However, there has not been much work in providing a common workflow monitoring tool within the eScience community. Instead, the community has focussed on achieving interoperability at the workflow representation level, which enables users to run the same workflow using different workflow engines or by embedding workflows as black box processes, e.g. SHIWA [38], WF4Ever [1].

Many workflow systems have some form of integrated monitoring capability. For example, Kepler, Triana, and Taverna support run-time monitoring through a graphical user interface. Some, like P-GRADE [26], support integrated performance analysis and debugging. Although there is a common set of analysis needs across these workflow systems, there is little interoperability between the monitoring capabilities. To date, interoperability of workflow state information has been focused on the *provenance* of computations, *e.g.*: the IETF Open Provenance Model [30], OPM-V [31], PREMIS [33], and SWAN [40]. Provenance, unlike performance data, is designed to extend in space and time beyond the system where the original computation was performed. Therefore, provenance systems are not well-suited for representation and access to system-specific details needed for performance analysis, such as CPU usage, job wait times, and failure codes.

Condor's Quill [25] provides well-defined interfaces to operational data from the scheduling substrate. However, this operational data is geared towards providing a global view of the computational infrastructure (i.e a Condor Pool). It is very difficult to correlate its statistics with the particular sequences of activities in a given workflow, sub-workflow, job, and so on. There is no way to directly associate jobs in a particular workflow with the data in the Quill database. Quill can be used to see jobs running by a particular user. However, it does not help in identifying what higher level workflow it belongs to. Quill serves the needs of an administrator of a computational resource, whereas we are concerned with the user experience.

III. Background

In this section, we describe the workflow systems integrated with Stampede and the underlying monitoring system.

A. Pegasus and Triana

The workflow systems being integrated, Pegasus and Triana, serve different requirements and user communities.

Pegasus is a workflow management system that bridges the scientific domain and the execution environment by automatically mapping high-level abstract workflow descriptions onto distributed resources. It automatically locates the necessary input data and computational resources necessary for workflow execution without the user needing to understand the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor [20], [19] , Globus [18], or Amazon EC2 [15]). Prior to the integration of Stampede in Pegasus [22], [37], the workflow and job logs were converted to Netlogger BP format and uploaded to a netlogger database separately after the workflows completed for analysis [9]. The database schema was a general schema [43] to load log messages and not specific to workflow systems.

Triana is a workflow and data analysis environment, providing an interactive GUI to enable the composition of scientific applications. It has been used in many Grid and standalone settings. Since Triana came from the gravitational wave physics application domain [41], the system contains a wide ranging palette of tools (around 400) for the analysis and manipulation of one-dimensional data, which are mostly written in Java (with some in C). Recently, other extensive toolkits have been added for audio analysis [42], image processing, text editing, for creating diabetic retinopathy decision-making workflows [3], [8], for data mining based on configurable Web Services [11], [4] and for distributed execution of multi-level workflows in clouds (see section V).

Pegasus and Triana differ in a number of significant ways. First, Pegasus is focused at the job level and utilizes Condor [20], [19] for the submission of its jobs. Triana, on the other hand, focuses more on services and components, where each component is a piece of Java code, which may or may not contain interfaces to distributed computing infrastructures. Therefore, whilst the typical mode of use for Pegasus would be to coordinate the running of thousands of jobs and manage their execution across multiple resources, Triana might focus on a more fine-grained mapping of the algorithms by decomposing them into sub-components or sub-workflows. In fact, the typical Triana model of distributed execution is quite recursive because it often involves a Triana pipeline of Java

components forming a job and being managed within a Triana meta-workflow to coordinate their distributed execution across the resources.

Another significant difference is that Pegasus executes workflows based on a Directed Acyclic Graph (DAG), i.e., a graph that cannot contain loops. Triana workflows, on the other hand, are free to contain loops and, for example, to be driven by the previous analysis of data. The two workflow systems also have different terminologies to describe the entities in their systems. Due to the nature of the representations, components or jobs being dealt with in the two systems, it follows that the monitoring information is very different between the two systems. The common data model exposed by Stampede [22], [37] allows both Pegasus and Triana entities to be mapped to this model. This provides the opportunity to interface with the two systems using the common terminology.

B. NetLogger

The monitoring framework underlying Stampede, called the NetLogger Toolkit [43], provides an integrated set of tools to collect, archive, and analyze time-series data. The format and structure of the data from NetLogger are defined in the Logging Best Practices (BP) document [2]. All monitoring data processed and archived by Stampede is first converted to BP log messages, giving the architecture flexibility to process the data in streams or load it into an alternative database.

IV. Stampede

The goal of Stampede [22], [37] is to enable real-time debugging and analysis of workflow performance. This is challenging because scientific workflows can involve many sub-workflows and millions of individual tasks [27]. Resource problems such as node and network failures, or application bugs, or both, can cause slowdowns and delays that are hard to detect and harder to debug. Users need automated analyses that can alert them to problems before resources and time are wasted, and then they need tools that can help them isolate the cause of the problems. These analyses in turn require access to a "live" repository with detailed events and recent history of the workflow and associated resource statistics. The repository must provide flexible mechanisms to navigate and sub select data sets that are too large to fit into memory.

Thus, Stampede must provide this repository based on information contained in thousands of log files [10], and then make the infrastructure truly useful by building new and advanced analyses on top of it. The types of views and analyses of the data that are enabled by Stampede include:

- *Global views* of the workflow synthesized from status and failure information from multiple components.
- *Real-time updates* of monitoring data, even for workflows with hundreds of thousands of tasks and high task throughput.
- *Real-time queries* of both detailed and summarized status.
- *Real-time troubleshooting* across all layers of the software and resource stack.
- *Performance prediction* of runtime and other resources, which are useful *e.g.* for provisioning on grids and clouds.
- *Anomaly detection* to distinguish actual failures from normal variation, which are particularly useful for large complex workflows like CyberShake [13].

A high level schematic of how Stampede interacts with Workflow Systems is shown in Figure 1. To achieve this, Stampede provides a three layer model for integration with different workflow systems.

1) *Common data model*. Stampede has a well defined data model to describe workflows and their executions. Workflow systems refer to this Data Model, to develop a workflow system-specific log normalizer that converts the workflow logs to Netlogger-formatted logs that are compatible with the model.
2) *High-performance loader*. The normalized logs are loaded into a data store using *nl_load* from the Netlogger Toolkit. *nl_load* has a *stampede_loader* module, that allows us to load large workflow logs in realtime into a relational archive.
3) *Query Interface*. There is a standard query interface for extracting the data from the relational archive. The Stampede troubleshooting, analysis and dashboard tools use this interface. Some of these tools are described in Section VII.

A. Data model

A lasting contribution of Stampede, upon which the analysis tools and other infrastructure depends, is a general data model for the performance characteristics of distributed workflows. Each workflow system has its own terminology for the data and processing components. In addition, some systems such as Pegasus distinguish between the abstract specification and its concrete mapping onto the target resources. Stampede defines a precise terminology to differentiate between the various components. The relationship between these terms is illustrated in Figure 2, and each is described below.

- *Workflow*: Container for an entire computation. Execution of a workflow is called a run.
- *Abstract workflow graph (AW)*: Input graph of tasks and dependencies, independent of a given run on specific resources. We assume AW to be a directed acyclic graph.
- *Executable workflow (EW)*: Result of mapping an AW to a specific set of resources. The cardinality of the AW task to EW job mapping is many-to-many.
- *Sub-workflow*: A workflow that is contained in another workflow.
- *Task*: Representation of a computation in the AW.
- *Job*: Node in the EW. A node in the EW can be associated with one or more tasks in the AW. It may also represent jobs added by the workflow system to manage the workflow that were not present in the AW, for example jobs added to stage-in data for the workflow.
- *Job instance*: Job scheduled or running by the workflow engine (e.g. DAGMan and Condor Schedd). Due to

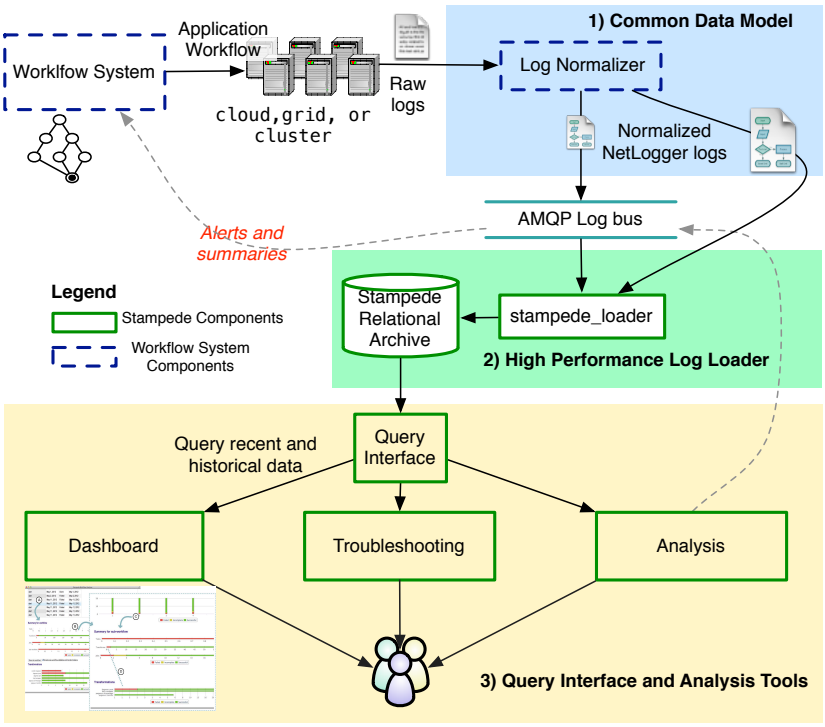

Fig. 1. Stampede data collection architecture.

retries, there may be multiple job instances per job.

- *Invocation*: When a job instance finishes, one or more invocations may be associated with that job instance. An invocation captures the actual invocation of an executable on a remote node. There can be multiple invocations if the corresponding job instance is associated with multiple tasks specified in the AW. A job present in the EW but not in AW also always has an invocation associated when the corresponding Job Instance terminates.

In relation to the computations (tasks) specified by the user in an AW, invocations are the instantiation of tasks, whereas jobs and job instances are an intermediate abstraction for use by the workflow planning and scheduling sub-systems.

B. Log Messages and Validation

In order for different workflow systems to utilize the same underlying monitoring infrastructure, it is important to formalize the format of the logs. In Stampede, we use the Netlogger [23] framework to format the log messages. The NetLogger events that make up the log messages are modeled using the YANG [45] schema language. YANG is a data modeling language originally developed to model configuration state data manipulated by the Network Configuration Protocol (NETCONF). This schema, available from [35], ensures that the events in the log messages conform to the Stampede data model described earlier. It captures required and optional attributes that need to be associated with different netlogger event messages. Modeling the log events in YANG allows us to use existing YANG validation tools like *pyang* [34] to validate log messages against the data model. These normalized logs can then be populated to Stampede data store using the loading infrastructure we have developed.

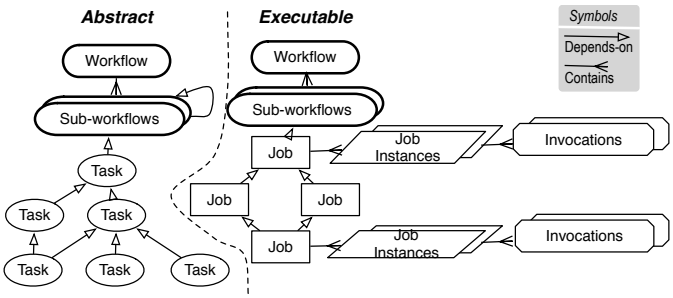

Fig. 2. Entities in Stampede Workflow Model.

An example NetLogger log event for Stampede, showing the start of a workflow, is shown below:

```
ts=2012-03-13T12:35:38.000000Z event=stampede.xwf.start
   level=Info xwf.id=ea17e8ac-02ac-4909-b5e3-16e367392556
   restart_count=0
```

Below is a snippet from the YANG schema that describes the *stampede.xwf.start* event and lists the attributes required. It also describes whether an attribute is mandatory or not.

```
container stampede.xwf.start {
     uses base-event;
     leaf restart_count {
             type uint32;
             mandatory "true";
             description    "Number of times workflow was
               restarted (due to failures)";
     }
}
```

The above description refers to a base event that identifies all the common attributes an event message conforming to schema should have. Below is a snippet describing the base event from the schema:

```
grouping base-event {
      description "Common components in all events";
      leaf ts {
        type nl_ts;
        mandatory "true";
        description
          "Timestamp, ISO8601 or  seconds since 1/1/1970";
      }
      ...
      leaf xwf.id {
        type uuid;
        description "Executable workflow id";
      }
    }  // grouping base-event
```

Having a formal, machine-processable, description for the log messages helps developers of workflow systems to write out the log messages conformant with the Stampede system.

C. Message bus

The NetLogger log events are placed on a message bus, which avoids blocking the producers and also easily accommodates many consumers for the same event. For this function, we chose to use RabbitMQ [36], a popular implementation of the standard Advanced Message Queueing Protocol (AMQP) [6]. AMQP defines an efficient and flexible publish/subscribe interface that is independent of the data model.

We use the hierarchical datatype of the NetLogger log message, called the *event* field, to route messages through an AMQP *topic queue*. Topic queues allow clients to subscribe to messages matching a prefix of the message type, *e.g.*, to receive all "stampede.job" messages or just the subset starting with "stampede.job.mainjob". This capability provides a great deal of flexibility in gluing together analysis components, while maintaining good performance and keeping implementations simple.

D. Relational archive

Currently, Stampede stores data collected from the workflow logs into an SQL database. To load into the database, we have developed a component called the *stampede_loader*. The loader can be accessed programmatically using its Python API, or workflow systems can write messages to the message bus and the *stampede_loader* can asynchronously insert them into a database. The stampede loader leverages the SQLAlchemy [39] object-relational mapping layer. This allows seamless support for a number of relational database products, including SQLite, MySQL and PostgreSQL.

The relational schema for workflow performance data is shown in Figure 3. Each workflow is an entry in the `workflow` table. Each workflow is associated with an Abstract Workflow (AW) described by the `task` and `task_edge` tables and an Executable Workflow (EW) described by the `job` and `job_edge` tables. Capturing the edges present in the AW and EW allows the tools to reconstruct the dependency graph of the jobs in the workflow. A job in the EW is associated with a job_instance that is recorded in the `job_instance` table. Each execution of a `job_instance` results in one or more invocation records that are recorded in the invocation table. The invocation record also links back to a task in the AW. Workflow state changes are recorded as rows in the `workflowstate` table. Like workflows, jobs are associated with any number of time-stamped and named states (`SUBMIT`, `EXECUTE`, `JOB_SUCCESS`, etc), which are stored in the `jobstate` table.

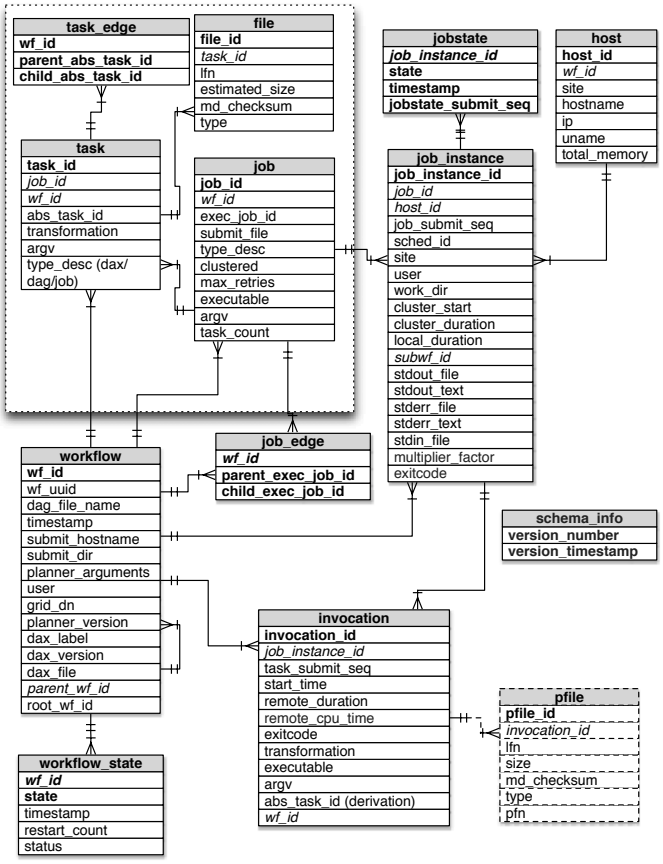

Fig. 3. Relational schema for Stampede Data Model.

E. Loader

The program that loads normalized log messages into the Stampede database, labeled `stampede_loader` in Figure 1, is `nl_load` a standard component of the NetLogger Toolkit (described in Section III-B). This program processes a stream of NetLogger BP log messages and inserts them into the Stampede database using the `stampede_loader` module that we developed for Stampede . The architecture is modular such that the loader can be invoked as a script from the command-line, or the Python modules can be imported and used as an API from within a Python program.

The loader can read its input from a file or an AMQP message queue, and it is able to save the data into a number of target databases and file formats. The architecture is, again, modular in that the loader can use an arbitrary Python

module for the target database. For example, to load logs from an AMQP message bus into the Stampede database, the `stampede_loader` module would be used:

```
nl_load --amqp-host=12.13.14.15  -A user=joe
    -A queue=stampede stampede_loader
     connString=mysql://16.17.18.19/mydb
```

On the first line of this invocation, the options describe how to connect to an AMQP message bus and retrieve events. The second line specifies the loader module and provides connection information for a MySQL database. Other databases supported by SQLAlchemy, such as PostgreSQL and SQLite, could also be used.

The loader has been shown to scale well for large workflows [37], for example the Cybershake workflows [28] that have O(10^6) tasks.

F. Analysis and Dashboard

Stampede tools enable users to troubleshoot workflows and extract meaningful performance statistics. Stampede also facilitates job and workflow level analysis, as described in our previous work [37] .Workflow-level analysis aims to predict workflow failures from basic aggregations on high-level statistics. Job-level analysis focuses on low-level detailed inspection of jobs to predict and isolate problem causes.

Users should not need to wait for a workflow to finish to see its status. To enable this, we have designed a very lightweight performance dashboard that enables easy monitoring and online exploration of workflows based on an embedded web server written entirely in Python. The performance dashboard is not described in this paper due to space constraints and will be described in a future publication.

V. Use of Stampede In Triana

In this section, we describe how Triana has been extended to use the Stampede Monitoring infrastructure. Unlike Pegasus, Triana is focused primarily around data flows or workflows of Java components and is therefore not focused directly on the execution of jobs and their dependencies, which require different semantics in the modeling. Each Java component can embed a job or interface with another system (e.g. Pegasus) to submit job-based workflows but it does not do this itself. At this level, therefore, Triana is not directly involved with mapping logical jobs and files across to distributed computing environments, such as Condor. Because of this, one of the key differences for the Triana Stampede integration is that there is no *planning* stage defined within the execution of a Triana workflow. Each task within a task graph is run locally (though that task may perform actions which include access or execution on remote locations). This means that there is a one-to-one mapping between a Stampede task and a Stampede job entity, unlike in Pegasus where multiple tasks may be clustered into a larger executable job during the planning stage.

This is illustrated in Figure 4, which shows the interaction of the various Triana components during a workflow execution. A task graph contains tasks, which may be another task graph (i.e. a sub-workflow, which can contain a sub-workflow, and so on). Each workflow consists of a collection of tasks (abstract Java class), which are implemented as a component within the Triana Java "Unit" class, and each unit class has a *process()* method that contains the code to be run within that unit.

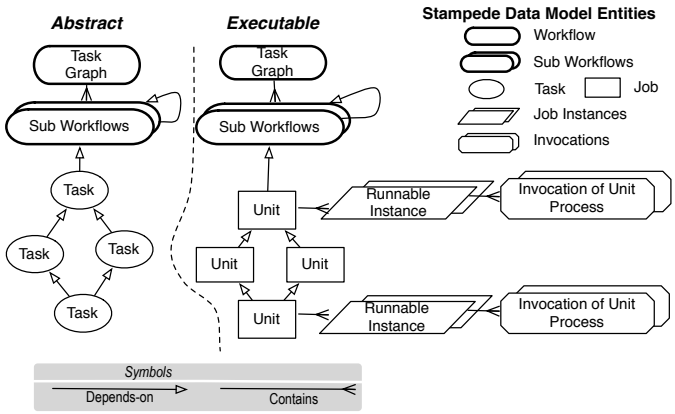

Fig. 4. Entities in Triana Workflow.

Figure 5 illustrates the Triana framework that generates Stampede data. *Runnable Instances* control the running of a task unit while the Scheduler controls the start/ stop/ reset/ events of a task graph lifecycle. The Scheduler also holds a *StampedeLog* object which listens for Triana *Execution Events* and converts them to *Stampede Events*. A Stampede event contains key value pairs which match the events in the Stampede schema. The StampedeLog also creates the events required for the schema compliance, but are not directly related to Triana events, such as mapping of tasks to units. These *Stampede Events* are then converted to the NetLogger format through the *Rabbit Appender* and recorded to either a file for later evaluation, or are posted directly to an AMQP queue for runtime processing using the Stampede tools.

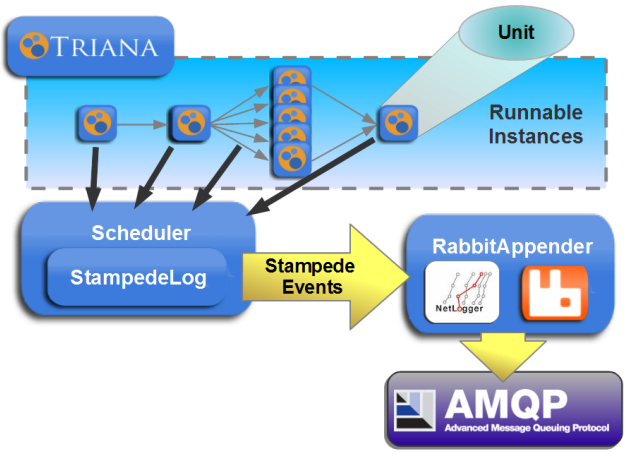

Fig. 5. Stampede logging in Triana.

A. Triana Execution Modes and the Stampede System

A major difference between Pegasus and Triana is the way in which Triana workflows are executed. Triana can be run in one of two modes: it can be run single step where each

component is scheduled to be executed once (like a DAG); or they can be run continuously, where a component continuously waits for data, until it is released through a local condition and stopped. In continuous mode, therefore, the logic of what runs and when a workflow finishes, is completely dependent on the data, its analysis and the condition that releases the components. For example, data can be analyzed until a certain threshold value is reached, within an iterative algorithm. Aside from conditional workflow exits, a workflow execution can be stopped interactively by the user within the GUI by pressing the *stop* button. This sends a message to the local task graph to pause the execution of each component in the workflow.

In Triana's continuous mode, all tasks are run almost simultaneously as separate Java threads. Each thread waits for all the input data to arrive before starting to process the data. To support this mode of operation, the Stampede job instance entity is used to model the *Runnable Instance*; that is, when the task is first set to a *running* state, the invocation entity is used to capture the processing of the task after all the inputs have been satisfied. This approach provides support for both Triana modes. However, for the use case described in this paper, Triana is run in *Single Step* mode, which is more compatible with a Pegasus run, allowing us to more easily compare a user's experience of using Stampede in both systems. In the future, we plan to devise a workflow experiment that executes a data driven workflow employing the continuous mode of operation of Triana.

B. Mapping with the Stampede Data Model

The events natively recognised within Triana by the workflow and tasks listener interfaces are: NOT_INITIALIZED, NOT_EXECUTABLE, SCHEDULED, RUNNING, PAUSED, COMPLETE, RESETTING, RESET, ERROR, SUSPENDED, UNKNOWN, and LOCK.

The states are reached via a transitioning *Execution Event*, which follows a broad "execution requested", "execution starting", "execution finished" and "execution reset" lifecycle. A task can reach each state for a variety of reasons, for example "execution finished" can occur with a complete, error, suspended or unknown state. An *Execution Event* manages state and can store the new state, as well as the previous state, giving some context as to the flow of the workflow.

As an example, converting the RUNNING state in Triana to a Stampede event, we would need to take into account whether the previous state was PAUSED or SCHEDULED, which would result in a *stampede.job_inst.held.end* or *stampede.job_inst.main.start* event, respectively. For other events, the mapping is more straightforward, such as PAUSED in Triana mapping directly to a *stampede.job_inst.held.start* event.

A large number of the Stampede events are populated in the logs at the beginning of a workflow's execution. Immediately before the scheduler sets the task graph's state to "RUNNING", the logging object records the workflow planning events, including the Task, Edge, and Job descriptions defined by Stampede. These simply describe the tasks and the connections (edges) between the tasks in the workflow. Further, when Triana receives a parent UUID from the incoming workflow bundle object, it is recognised that this is a sub-workflow being prepared, and the UUIDs of the parent and child are logged.

Once the task graph enters a "RUNNING" state, each task is "WOKEN", their Job Submit Start event is recorded and they wait for input data. Once data is received, the Task enters the "RUNNING" state and begins to process the data, triggering an Invocation Start event. The data is then processed, which results in the Invocation Terminate and Invocation End events being recorded. If a unit's process method throws an error instead of returning appropriately, the Terminate and End events have return codes of "-1".

When Triana tasks are set to *Run Continuously* mode, the process method of a Task's Unit is able to run more than once. This allows a streaming function, where chunks of data from previous tasks can be processed in succession, utilizing a queuing function at both the input and output cables of the task. At each unit's execution time step, Invocation Start is recorded, allowing a job to have multiple invocations during each execution of the workflow. The Invocation End event is fired each time the process method completes.

The execution of a single workflow on a local machine results in the logging information produced by Triana being collected locally in a log file. If the workflow is re-run, this is considered to be a new workflow, rather than a re-execution of a previous one. The Host Information is also logged, including the *localhost* hostname.

C. Log Collection

Triana utilizes standard java logging mechanisms like *LOG4J* to do the logging. In order to incorporate Stampede logging to an AMQP queue, we integrated a Stampede RabbitMQ appender, and to record the Stampede events consistent with the YANG [35] schema, we created a mapping of Triana events to the corresponding events in the YANG schema.

The RabbitMQ appender, written for Triana, is discovered using the standard *LOG4J* system, and allows logging details to be sent as they are produced. In this way the events are received on the AMQP queue in real time, and can be listened for via any connected consumers. The nl_load tool can then be set up to listen for logging events on an AMQP channel, and direct them all to a database file, as follows:

```
nl_load -a  s-vmc.cs.cf.ac.uk -p 7000
    -A user=username -A pw=password -A queue=Stampede
    -A durable=true -A auto_delete=false
    stampede_loader connString=sqlite:///test.db -v
```

D. Distributed Execution of Triana - Meta Workflows

Triana has the ability to modify its workflow at runtime. This can mean anything from the changing of input parameters and premature removal of tasks, to the addition of new tasks within the workflow. This latter ability is used in the creation of a sub-workflow, which in this sense refers to the creation and execution of a workflow during the run of a parent workflow. The sub-workflow can further be responsible for

the creation of a series of other sub-workflows at runtime, and the execution may follow immediately.

This function is used in the distributed execution of Triana for running a workflow and spawning sub-workflows on multiple nodes within a cloud environment. Specifically, a meta-workflow processes multiple tasks and dynamically produces a number of smaller sub-workflows ready for distribution. It is possible at this stage to design a sub-workflow to use input parameters to concretize specific executions, allowing for more accurate spreading of load across available resources. For example, partial analysis could be conducted on input data to prepare it before the more processor-intensive tasks are farmed out to remote execution environments. Each of these sub-workflows can then be sent to a remote cloud node and executed within a Triana environment running on each node. In this way, input variables or command line arguments can be defined in advance of distribution, which is desirable in the case where they cannot be generated easily by the workflow itself.

Each of these workflows running on a distributed node can be seen as a sub-workflow of the initial meta-workflow running on the user's desktop machine. In case of Meta Workflows in Triana, the sub-workflow tasks in the root workflow are generated by the root workflow at runtime before the submission of the sub-workflow. However, the way the *stampede-loader* is implemented it requires all the events related to the mapping between the tasks in the abstract workflow, and the jobs in the executable workflow to be generated before execution can proceed. This meant that for the experiment described in this paper, we had the number of sub-workflows in the meta workflow pre-defined. It is important to note that this is not a deficiency of the Stampede data model. Instead it is a limitation of the *stampede-loader*, which will be addressed in future work. This behavior was implemented to improve the performance of Pegasus workflows logging by batching similar inserts together.

VI. Scientific Experiment

In order to demonstrate that the analysis and debugging infrastructure developed as part of Stampede can be applied to Triana, we chose to model DART workflows [42] in Triana. The DART workflow uses the DART Music Information Retrieval (MIR) research platform to perform a parameter sweep experiment in order to discover the optimal parameter settings for the Sub-Harmonic Summation (SHS) pitch detection algorithm.

The DART application was originally designed and created in Triana, using the graphical workflow-design environment as a development test bed for the distributed algorithms. The algorithm was converted into a standalone JAR (multi-platform) executable and is distributed (along with any required input data) to workers at runtime. DART parameters can be modified from the DART Command Line Interface and a parameter sweep experiment can be carried out by creating a script to generate the required execution parameters, which can be run sequentially.

The parent workflow designed for Triana uses a single file as its input. This file was created using a separate Python script, and defines a list of 306 strings, separated by the newline character. These strings are executable via a terminal's command line, and as such are able to run anywhere where the DART jar and the audio input files present in the execution directory.

The function of the Triana workflow is to split the input file into individual lines, and edit a pre-defined aggregate file (using SHIWA bundles [38]). A Triana task creates a series of new workflows using a subset of the input lines, with each workflow containing approximately 16 executing tasks. This set of workflow files is added to an existing bundle file, resulting in as many executable bundles as there were sub-workflows created. A final task in the workflow sends each of these bundles to the TrianaCloud Broker via an HTTP POST. The Broker is then responsible for each sub-workflow's execution.

The DART workflow was run on the TrianaCloud infrastructure and used 8 cloud nodes, each running Ubuntu 12.04, with 2GB RAM, 20GB disk space, 1 core per instance.

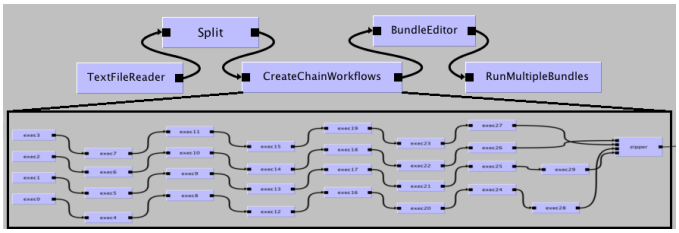

Fig. 6. DART Workflow - Triana

A. Triana workflow within the GUI

Figure 6 shows the workflows that execute in Triana. The parent workflow (top) is designed to run on the user's local machine, typically a desktop, within the Triana GUI. This workflow creates the child workflows (bottom) at runtime, and executes them as sub-workflows on the distributed TrianaCloud environment. These sub-workflows have 16 executable tasks, which run 4 at a time on the compute node. After all the executable tasks have all completed, a *Zipper* task collates all the outputs produced in the *results* folder.

VII. Analysis of Results

In this section, we show how Triana users can use the Stampede tools to debug their workflows and extract meaningful statistics and troubleshooting information from their workflow runs. We give an overview of the statistics that Stampede provides and then present sample statistics from a Triana workflow run.

Users often execute workflows on a variety of cyberinfrastructures like nationally distributed computational grids (e.g. XSEDE [44]) and increasingly on commercial and scientific clouds [15] [17]. To execute on such resources, users put in allocation requests or provision resources beforehand. One way for a user to determine the amount of resources required

is to do a baseline run and use that to extrapolate accordingly. In order to do this, a user needs detailed metrics about their workflow runs.

Both Pegasus and Triana provide for distributed execution of jobs in a particular workflow. In a distributed environment, there are multiple points where delays can be introduced for a job. For example, a job can experience delays in a remote queue of a cluster. It is useful for the users to be able to determine the various delays a job encounters. For example, if a user notices that there are long scheduling delays, they may choose to restructure their workflows so that each job does a larger unit of work.

Stampede provides a tool called *stampede_statistics* that can provide such statistics for a particular workflow. The tool provides statistics both at workflow level and at job level. Below is a summary of some statistics provided by Stampede .

Workflow Level Statistics

- *Workflow wall time*: The wall time from the start until the end of the workflow execution, as reported by the workflow engine.
- *Workflow cumulative job wall time*: Sum of actual run times for all the jobs in the workflow. This helps in estimating the resources a workflow requires in a perfect system without delays.
- *Breakdown of jobs by count*: Breakdown of count by job type. For each job types, lists the total number of jobs, the number succeeded and number failed.
- *Breakdown of jobs by runtime*: This captures the total, minimum, maximum and mean runtimes for each different job type in the workflow.
- *Breakdown of tasks and jobs over time on hosts*: A single workflow can be executed over a number of hosts. This captures the number of jobs and total runtime of these jobs executed by each host over time.

Job Level Statistics

For each job, stampede_statistics also displays statistics like

- the name of the job
- runtime of the job (as measured by the workflow engine)
- remote delay encountered
- actual CPU time used (if captured)
- the host on which the job ran.

A. Triana Execution Statistics

A run of 306 executions of the DART experiment was performed on the TrianaCloud environment. These 306 tasks were split into sub-workflows for distribution on the available nodes within the cloud. This required the construction of a root workflow to run on the user's machine, which contained one submission task for each sub-workflow. For ease of use, this root-workflow was generated by a meta-workflow, which allows modification of the number of tasks running in each sub-workflow. Choosing to have 16 tasks per sub-workflow generated 20 workflow bundles for execution within the pool. Progress of these workflow bundles can be seen in Figure 7, with wall-clock time on the X axis and the cumulative runtime of each bundle on the Y axis.

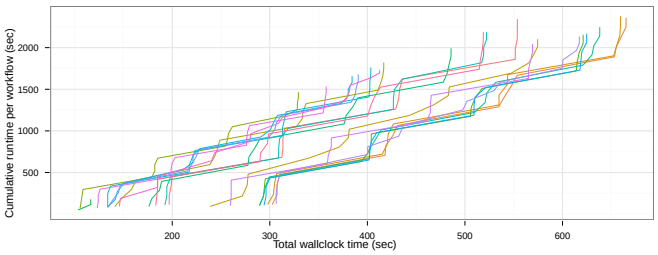

Fig. 7. Progress to completion of DART workflow "bundles" of 16 tasks per sub-workflow.

Each of the sub-workflows has additional tasks to prepare outputs, while the root workflow has tasks for monitoring the running of the sub-workflows. In total, there were 367 tasks. The Stampede logs were collected via the AMPQ queue and `nl_load` was used to load these events into a database during runtime. The log events were also retained in their original (plain-text NetLogger Best Practices) format. The loaded data was then analyzed by the stampede-statistics tool, to retrieve the metrics and provide a human-readable output, shown in Table I.

Type	Succeeded	Failed	Incomplete	Total	Retries	Total
Tasks	367	0	0	367	0	367
Jobs	367	0	0	367	0	367
Sub WF	20	0	0	20	0	20

Workflow wall time : 11 mins, 1 sec, (661 seconds).
Workflow cumulative job wall time : 11 hrs, 10 mins, (40224 seconds).

TABLE I
SUMMARY OUTPUT FROM STAMPEDE-STATISTICS FOR DART WORKFLOW

The *stampede-statistics* tool also produces a number of output files, which further detail the execution time of individual jobs. Below are some examples of the available information.

Table II shows a section of *breakdown.txt*, defining the times of each task in the workflow. This section shows a sub-workflow of 4 jobs executing the DART experiment. The Stampede tools can also generate aggregated statistics for a meta workflow that include all the sub-workflows.

Type	Count	Success	Failed	Min	Max	Mean	Total
115-119	1	1	0	1.0	1.0	1.0	1.0
Output_0	1	1	0	1.0	1.0	1.0	1.0
exec0	1	1	0	74.0	74.0	74.0	74.0
exec1	1	1	0	75.0	75.0	75.0	75.0
exec2	1	1	0	74.0	74.0	74.0	74.0
exec3	1	1	0	75.0	75.0	75.0	75.0
exec4	1	1	0	36.0	36.0	36.0	36.0
zipper	1	1	0	1.0	1.0	1.0	1.0

TABLE II
BREAKDOWN.TXT DESCRIBING THE TASKS IN A SUB-WORKFLOW

Tables III and IV show the entry for a sub-workflow in *jobs.txt*, showing the location where the job ran and the times

as seen from different perspectives. The invocation duration captures the duration of the task on the remote host. The queue time is the time spent by the job in the remote queue before it started executing.

Job	Try	Site	Invocation Duration
unit:304-305	1	trianaworker6	1.0
exec1	1	trianaworker6	64.0
file.Output_0	1	trianaworker6	1.0
file.zipper	1	trianaworker6	1.0
processing.exec0	1	trianaworker6	51.0

TABLE III
SECTION OF JOBS.TXT FOR A SINGLE SUB WORKFLOW

Job	Queue Time	Runtime	Exit	Host
unit:304-305	0.06	1.0	0	None
exec1	0.04	64.0	0	trianaworker6
file.Output_0	0.0	1.0	0	trianaworker6
file.zipper	0.0	1.0	0	trianaworker6
processing.exec0	0.07	51.0	0	trianaworker6

TABLE IV
SECTION OF JOBS.TXT FOR A SINGLE SUB WORKFLOW

B. Troubleshooting

Stampede allows users to debug their workflows using a tool called *stampede_analyzer*, employing the Stampede query interface. It is a command-line utility that connects to the Stampede Data Store and queries it for a given workflow. Its output contains a brief summary section, showing how many jobs have succeeded and how many have failed. For each failed job, stampede_analyzer will print information showing its last known state, along with the location of its job description, output, and error files. It will also display any application stdout and stderr that was captured by the workflow system and stored in the Data Store.

In the Stampede Data Model, a workflow can consist of tasks that refer to other sub-workflows modeled. Large workflows can be modeled in Pegasus and Triana conceptually as layered hierarchal workflows. The stampede_analyzer is an interactive debugging tool, that allows a user to debug each level of the hierarchy. It first identifies for users the failures at the top level workflow and then allows them to drill down the hierarchy to debug the individual failed workflows. The output is not included because of space constraints.

VIII. Conclusion and Future Work

Stampede is a distributed monitoring infrastructure for scientific workflows that employs a three-layer model for integration with different workflow systems. It was initially incorporated into the Pegasus Workflow Management System. In this paper, we have described how the Stampede data model and infrastructure was applied to the Triana Workflow System. The two workflow systems have different specification languages, execution engines and use cases. We gave a detailed overview of the Stampede approach and described the system-specific steps involved in integrating the Stampede data model into Triana.

In order to demonstrate the usefulness of integrating Triana and Stampede, we modeled and deployed a music information retrieval workflow experiment in Triana. The workflow was executed on a cloud deployment in Cardiff University and workflow logs were loaded in realtime into the Stampede data store. We then provided a walkthrough of how the standard stampede tools can help Triana users to troubleshoot and mine performance metrics for their workflows.

In our earlier work, we have investigated the performance of the Stampede data model for loading large workflows that are executed through PegasusWMS [37]. Since both the workflow systems use the same Stampede component (the *nl_load*) to load the logs, we do not expect any performance penalty when running large workflows through Triana. In future, we plan to test this hypothesis further by doing detailed experiments that involve running workflows of varying sizes through Triana and evaluation of the loading performance.

Stampede also provides analysis components that give insight into the workflow execution to enable performance prediction and fault diagnosis. The results of this analysis have been applied to Pegasus workflow runs in our previous work [22]. In future, we plan to do similar analysis on larger corpus of workflow runs.

Acknowledgment

The Stampede work was supported by National Science Foundation grant OCI-0943705 and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract DE-AC02-05CH11231.

For Triana, we would like to thank our sponsors, PPARC (GridOneD and Geo 600) for the development of Triana, UK STFC TRIACS project ST/F002033/1 for the Triacs work, Wellcome Trust for the Sintero work and the EU for the Gridlab project to help the development of the distributed computing capabilities and SHIWA for the development of the SHIWA bundles that provide the cloud-based distributed mechanisms, described in the Triana sections of this paper. We would also like to thank Andrew Harrison for his insight and for helping recreate Triana in its present form.

References

[1] The EU Wf4Ever Project. http://www.wf4ever-project.org/.
[2] Grid logging: Best practices guide, 2008.
[3] David R. Owens 4. Adina Riposan, Ian J. Taylor, Omer Rana and Edward C. Conley. TRIACS Workflows Platform For Distributed Decision Support Processes. In *In CBMS 2009*, Albuquerque, 2009.
[4] A.S. Ali, O.F. Rana, and I.J. Taylor. Web Services Composition for Distributed Data Mining. In *ICPP 2005 Workshops, International Conference Workshops on Parallel Processing*, pages 11–18. IEEE, New York, 2005.
[5] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 423–424. IEEE Computer Society, New York, 2004.
[6] Advanced message queuing protocol. Web: http://www.amqp.org.

[7] Roger Barga, Jared Jackson, Nelson Araujo, Dean Guo, Nitin Gautam, and Yogesh Simmhan. The trident scientific workflow workbench. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 317–318, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Taylor I. Benson T, Conley EC, Harrison AB. Sintero Server Simplifying interoperability for distributed collaborative health care. In *IHIC 2011 Conference, Orlando, May*, 2011.

[9] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher X. Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, David Okaya, and Thomas Jordan. Scaling up workflow-based applications. *J. Comput. Syst. Sci.*, 76(6):428–446, 2010.

[10] Scott Callaghan, Philip Maechling, Patrick Small, Kevin Milner, Gideon Juve, Thomas Jordan, Ewa Deelman, Gaurang Mehta, Karan Vahi, Dan Gunter, Keith Beattie, and Christopher X. Brooks. Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *IJHPCA*, 25(3):274–285, 2011.

[11] Data Mining Tools and Services for Grid Computing Environments (DataMiningGrid). http://www.datamininggrid.org/.

[12] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D.S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[13] Ewa Deelman, Scott Callaghan, Edward Field, Hunter Francoeur, Robert Graves, Nitin Gupta, Vipin Gupta, Thomas H. Jordan, Carl Kesselman, Philip Maechling, John Mehringer, Gaurang Mehta, David Okaya, Karan Vahi, and Li Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '06, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Gener. Comput. Syst.*, 25:528–540, May 2009.

[15] Amazon Elastic Cloud. Web:http://aws.amazon.com/ec2.

[16] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing*, pages 122–131. IEEE Computer Society Press, New York, 2005.

[17] FutureGrid. Web:https://portal.futuregrid.org/.

[18] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[19] J. Frey. Condor DAGMan: Handling inter-job dependencies, 2002.

[20] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPCD-'01)*. IEEE Computer Society, New York, 2001.

[21] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. *IJHPCA*, 22(3):347–360, 2008.

[22] Dan Gunter, Ewa Deelman, Taghrid Samak, Christopher X. Brooks, Monte Goode, Gideon Juve, Gaurang Mehta, Priscilla Moraes, Fabio Silva, D. Martin Swany, and Karan Vahi. Online workflow management and performance analysis with stampede. In *CNSM*, pages 1–10. IEEE, 2011.

[23] Dan Gunter and Brian Tierney. Netlogger: A toolkit for distributed system performance tuning and debugging. In *Integrated Network Management, IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003)*, volume 246 of *IFIP Conference Proceedings*, pages 97–100. Kluwer, 2003.

[24] Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. WS-RF Workflow in Triana. *International Journal of High Performance Computing Applications*, 22(3):268–283, August 2008.

[25] J. Huang, A. Kini, E. Paulson, C. Reilly, E. Robinson, S. Shankar, L. Shrinivas, D. DeWitt, and J. Naughton. An overview of Quill: A passive operational data logging system for Condor. *Computer Sciences Technical Report, University of Wisconsin*, 2007.

[26] Peter Kacsuk. P-grade portal family for grid infrastructures. *Concurr. Comput. : Pract. Exper.*, 23:235–245, March 2011.

[27] Daniel S. Katz, Joseph C. Jacob, G. Bruce Berriman, John Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, and Thomas A. Prince. A comparison of two methods for building astronomical image mosaics on a grid. In *ICPP Workshops*, pages 85–94. IEEE Computer Society, 2005.

[28] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan, and E. Field. Scec cybershake workflows – automating probabilistic seismic hazard analysis calculations. In Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shield, editors, *Worflows for e-Sciences*. Springer, 2006.

[29] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[30] The Open Provenance Model (OPM). http://openprovenance.org/.

[31] The Open Provenance Model Vocabulary Specification (OPM-V). http://open-biomed.sourceforge.net/opmv/ns.html.

[32] Simon Ostermann, Kassian Plankensteiner, Radu Prodan, Thomas Fahringer, and Alexandru Iosup. Workflow monitoring and analysis tool for askalon. In Ramin Yahyapour, Domenico Talia, and Norbert Meyer, editors, *CoreGRID Workshop on Grid Middleware*, pages 1–14, 2008.

[33] PREservation Metadata Implementation Strategies (PREMIS). http://www.loc.gov/standards/premis/v2/premis-2-0.pdf.

[34] PYANG - An extensible YANG validator and converter in python. Web:http://www.yang-central.org/twiki/pub/Main/YangTools/pyang.1.html.

[35] Yang Schema for Stampede Log Messages. Web:http://acs.lbl.gov/projects/stampede/4.0/stampede-schema.html.

[36] RabbitMQ. Web: http://www.rabbitmq.com.

[37] Taghrid Samak, Dan Gunter, Monte Goode, Ewa Deelman, Gideon Juve, Gaurang Mehta, Fabio Silva, and Karan Vahi. Online fault and anomaly detection for large-scale scientific workflows. In Parimala Thulasiraman, Laurence Tianruo Yang, Qiwen Pan, Xingang Liu, Yaw-Chung Chen, Yo-Ping Huang, Lin huang Chang, Che-Lun Hung, Che-Rung Lee, Justin Y. Shi, and Ying Zhang, editors, *HPCC*, pages 373–381. IEEE, 2011.

[38] The SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs Project . http://www.shiwa-workflow.eu/.

[39] SQLAlchemy. Web: http://www.sqlalchemy.org.

[40] Semantic Web Applications in Neuromedicine (SWAN). http://swan.mindinformatics.org/spec/1.2/pav.html.

[41] Ian Taylor. Triana Generations. In *Scientific Workflows and Business workflow standards in e-Science in conjunction with Second IEEE International Conference on e-Science*, Amsterdam, Netherlands, December 2-4 2006.

[42] Ian Taylor, Eddie Al-Shakarchi, and Stephen David Beck. Distributed Audio Retrieval using Triana (DART). In *International Computer Music Conference (ICMC) 2006, November 6-11, at Tulane University, USA.*, pages 716–722, 2006.

[43] B. Tierney and D. Gunter. NetLogger: A toolkit for distributed system performance, tuning and debugging. In *Proceedings of the IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003), Vol. 246 of IFIP Conference Proceedings*, pages 97–100. Kluwer, 2003.

[44] Extreme science and engineering discovery environment. https://www.xsede.org/.

[45] YANG - A Data Modeling Language for the Network Configuration Protocol. Web:http://tools.ietf.org/html/rfc6020.