

# *Kepler* Science Operations Center pipeline framework

Todd C. Klaus<sup>\*a</sup>, Sean McCauliff<sup>a</sup>, Miles T. Cote<sup>b</sup>, Forrest R. Girouard<sup>a</sup>, Bill Wohler<sup>a</sup>, Christopher Allen<sup>a</sup>, Christopher Middour<sup>a</sup>, Douglas A. Caldwell<sup>c</sup>, Jon M. Jenkins<sup>c</sup>

<sup>a</sup>Orbital Sciences Corp./NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035;

<sup>b</sup>NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035;

<sup>c</sup>SETI Institute/NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035

## ABSTRACT

The *Kepler Mission* is designed to continuously monitor up to 170,000 stars at a 30-minute cadence for 3.5 years searching for Earth-size planets. The data are processed at the Science Operations Center at NASA Ames Research Center. Because of the large volume of data and the memory needed, as well as the CPU-intensive nature of the analyses, significant computing hardware is required. We have developed generic pipeline framework software that is used to distribute and synchronize processing across a cluster of CPUs and provide data accountability for the resulting products. The framework is written in Java and is, therefore, platform-independent. The framework scales from a single, standalone workstation (for development and research on small data sets) to a full cluster of homogeneous or heterogeneous hardware with minimal configuration changes. A plug-in architecture provides customized, dynamic control of the unit of work without the need to modify the framework. Distributed transaction services provide for atomic storage of pipeline products for a unit of work across a relational database and the custom *Kepler DB*. Generic parameter management and data accountability services record parameter values, software versions, and other metadata used for each pipeline execution. A graphical user interface allows for configuration, execution, and monitoring of pipelines. The framework was developed for the *Kepler Mission* based on *Kepler* requirements, but the framework itself is generic and could be used for a variety of applications where these features are needed.

**Keywords:** *Kepler*, pipelines, distributed processing, cluster computing, photometry, framework

## 1. INTRODUCTION

The *Kepler* pipeline framework is a software platform designed for building applications that run as automated, distributed pipelines. The framework is designed to be reusable, and therefore contains no application-specific code. Application-specific components, such as pipeline modules, parameters used by the application, and the unit-of-work specification for each module, integrate with the framework via interfaces and abstract classes. The *Kepler* Science Operations Center (SOC)<sup>1</sup> uses this framework to build the pipelines that process the ~23 GiB of raw pixel data downlinked from the *Kepler* spacecraft every month on four computing clusters containing a total of 64 nodes, 512 CPU cores, 2.3 TiB of RAM, and 148 TiB of raw disk storage<sup>2</sup>.

The framework attempts to maximize the level of customization for applications by allowing them to dynamically specify how the unit of work is defined for each pipeline module while minimizing the amount of custom code that must be written. The application need only supply code (known as the unit-of-work generator) that generates unit-of-work descriptors for each module and the code for the modules. The framework invokes the generator just prior to executing each module in the pipeline and turns the resulting unit-of-work descriptors into pipeline jobs (one job per descriptor). It then executes and monitors those jobs and automates the transitions to the next modules, with support for multiple transition types (see Section 4). Module code need only process the data specified by the unit-of-work descriptor. The module code does not need to be aware of which module needs to be run next, or even where it sits relative to other

---

\* [Todd.C.Klaus@nasa.gov](mailto:Todd.C.Klaus@nasa.gov)

modules in the pipeline configuration. Modules read data to and write data from a common data store; they do not communicate directly with each other<sup>3</sup>.

While the framework is designed to enable distributed processing on clusters of many nodes, it also scales down to a single workstation for development and test scenarios. The framework builds on many existing open source components (JMS<sup>4</sup>, Hibernate<sup>5</sup>, Apache Commons<sup>6</sup>, JBoss JTA<sup>7</sup>).

## 2. PLUGGABLE ARCHITECTURE

Applications classes extend the framework at three main plug-in points: (1) The `UnitOfWorkGenerator` and `UnitOfWorkDescriptor` interfaces allow customization of the unit of work, (2) the `PipelineModule` abstract class allows for implementation of pipeline modules, and (3) the `Parameters` interface allows for the definition of the parameters used by the pipeline modules and unit-of-work generators. Figure 1 illustrates these framework extension points, using the *Kepler* Data Validation<sup>8</sup> module as an example.

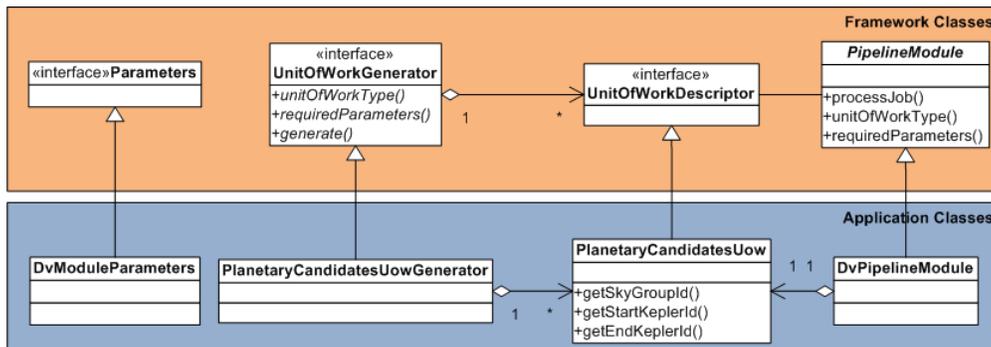


Figure 1: Framework extension points, using the *Kepler* Data Validation (DV) module as an example.

### 2.1 Unit-of-work generators

A key consideration when designing a distributed pipeline application is how to define the unit of work for each job processed on the cluster. For some applications, the unit of work remains constant throughout all modules that make up the pipeline. For example, in an image processing pipeline, the unit of work might be a single image. The image is modified as it passes through the various modules of the pipeline, but one image comes out of the end of the pipeline for every image that goes in. For this type of application, pipeline execution logic can be fairly simple. For each image to be processed, a single job is executed for each module in the pipeline.

For other applications, the nature of the unit of work changes from module to module. For example, in a photometric data reduction pipeline, a set of images flows into the pipeline, but a light curve for each star comes out the other end. Even in this case, there can be a single job for each module if the module that performs the aperture photometry performs this task for all stars in the image as a single job, preserving the simple model of one job per module. However, as the size of the data set grows, it may make more sense to partition the data to create multiple jobs per module so that they can run in parallel on the cluster. In addition, the way the data are partitioned may be different for each module and may need to be determined dynamically. The *Kepler* pipelines<sup>3</sup> fall into this category.

For example, the *Kepler* photometer<sup>9</sup> consists of an array of 42 charge-coupled devices (CCDs), each with two output amplifiers, for a total of 84 module/outputs. To keep its solar panels facing the sun, the spacecraft rotates 90 degrees about the optical axis every 90 days, changing the group of stars that fall on each module/output. For the front end of the *Kepler* pipeline (i.e., the Calibration<sup>10</sup> (CAL), Photometric Analysis<sup>11</sup> (PA), and Pre-search Data Conditioning<sup>12</sup> (PDC) modules), it makes sense to partition the data into 84 units of work (one per module/output), and then further partition these units temporally (same spacecraft orientation for a given unit of work). Meanwhile, for the back end of the pipeline (i.e., the Transiting Planet Search<sup>13</sup> (TPS) and Data Validation (DV) modules), each target can be processed independently and all available time samples from the beginning of the mission are needed in order to find long-period planets. In particular, DV is designed to run only on those targets that exceed the detection threshold in TPS, so the unit of work for DV cannot be predefined. These use cases imply that the framework needs to be flexible enough to support a different unit-of-work type for each pipeline module, while recognizing that the unit of work for one module may be dependent on the results of a previous module.

The application implements the `UnitOfWorkGenerator` and `UnitOfWorkDescriptor` interfaces to define custom unit-of-work types. The `generate` method of the `UnitOfWorkGenerator` is responsible for generating a collection of `UnitOfWorkDescriptor` objects. These descriptors contain fields that describe the unit of work. The framework creates a job to be executed on a cluster node for each descriptor. The `UnitOfWorkDescriptor` interface is simply a marker interface, as it contains no methods or fields. It serves as a common base class for all classes that implement the interface so that objects of this type can be handled and persisted generically by the framework. The concrete classes contain the actual specification for the unit of work. For example, the unit of work for the *Kepler* CAL module contains the CCD module number, the CCD output number, the start time, and the end time<sup>3</sup>.

When the operator assigns a generator to a pipeline module, the configuration graphical user interface (GUI) verifies that the unit-of-work type produced by the generator matches the type expected by the module. Generators can also use parameters from the parameter library to control their behavior. The `generate` method takes as arguments all of the parameter sets configured for the module by the operator.

## 2.2 Pipeline modules

The `PipelineModule` abstract class provides the entry point for the module. To implement a new module, the application creates a new class that extends the `PipelineModule` class and implements the abstract methods. When processing a job, the framework will create an instance of this class and invoke it via the `processJob` abstract method, passing it the unit-of-work descriptor and any parameters configured for the module. This class also contains abstract methods that the application implements to specify parameters required by the module and the expected unit-of-work type. These methods are used by the framework to enforce valid configurations in the configuration GUI.

## 3. DATA MODEL

The framework uses a relational database to store pipeline configurations and metadata created during pipeline execution, as shown in Figure 2.

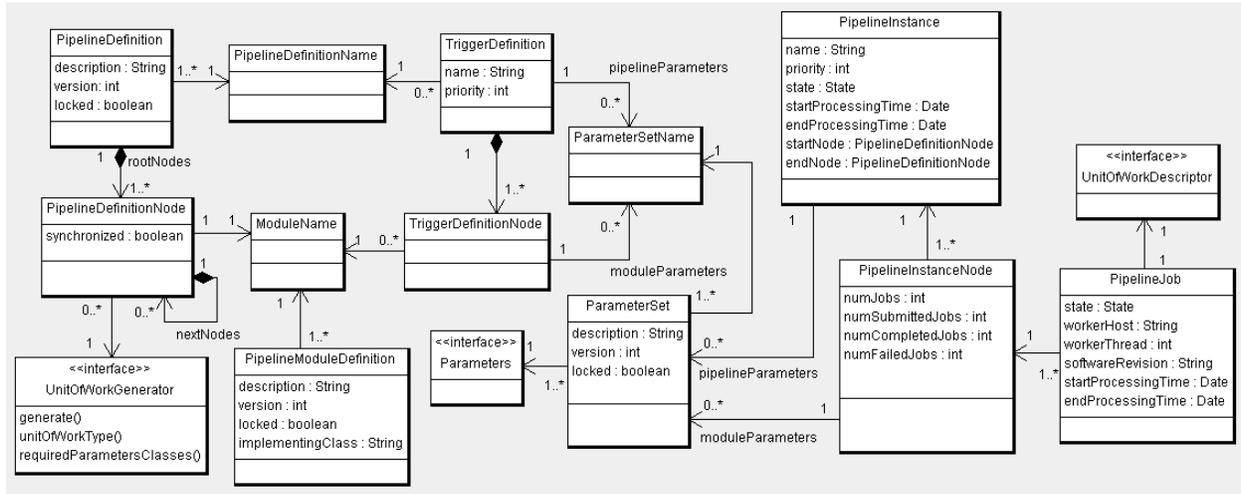


Figure 2: Framework data model.

The entities on the left represent the various elements of pipeline configurations, including pipeline definitions, the module library, and the parameter and trigger libraries. Configuration objects employ a versioning mechanism (described below) to preserve the state of the configuration for each pipeline instance for data accountability purposes.

The entities on the right (`PipelineInstance`, `PipelineInstanceNode`, `PipelineJob`, and `UnitOfWorkDescriptor`) store metadata created by the framework during pipeline execution. These objects include references to the specific versions of parameters used by jobs in the executing pipeline, unit-of-work descriptors, and other details about each job, such as which cluster node it ran on, the version of software used, and the current state. Metadata also include the current state of the data model registry. The data model registry is a framework entity that keeps track of the current versions of various data models used by the pipeline modules in a generic way (i.e., by name

and version only). Examples of data models tracked in the registry by the *Kepler* SOC include models that describe various characteristics of the *Kepler* photometer (e.g., electronic noise levels and optical vignetting), although any model with a unique name and version can be tracked in this fashion. Linking the state of the registry to the pipeline instance metadata makes it possible to determine the exact version of the models at the time any data product was generated.

The execution metadata objects, along with the versioned configuration objects, together form a complete data accountability record for every pipeline job. Each pipeline data product can be tagged with a single identifier (job ID) that leads back to the complete data accountability record for that product, including parameters used, software version used, versions of various models used, the unit of work to which it belongs, and the entire state of the pipeline configuration at the time the job executed.

Framework entities are stored in a relational database and accessed via an Object/Relational Mapping (ORM) layer implemented with Hibernate, an open source ORM framework. Because of the Hibernate layer, the underlying relational database can be changed without changes to the code. Use cases include Oracle for large cluster deployments and HSQLDB<sup>14</sup> for single-workstation deployments, although the choice of relational database is limited only by the list of databases supported by Hibernate. Entities are managed via the configuration GUI (shown in Figure 3), an XML import/export interface, or a programmatic interface.

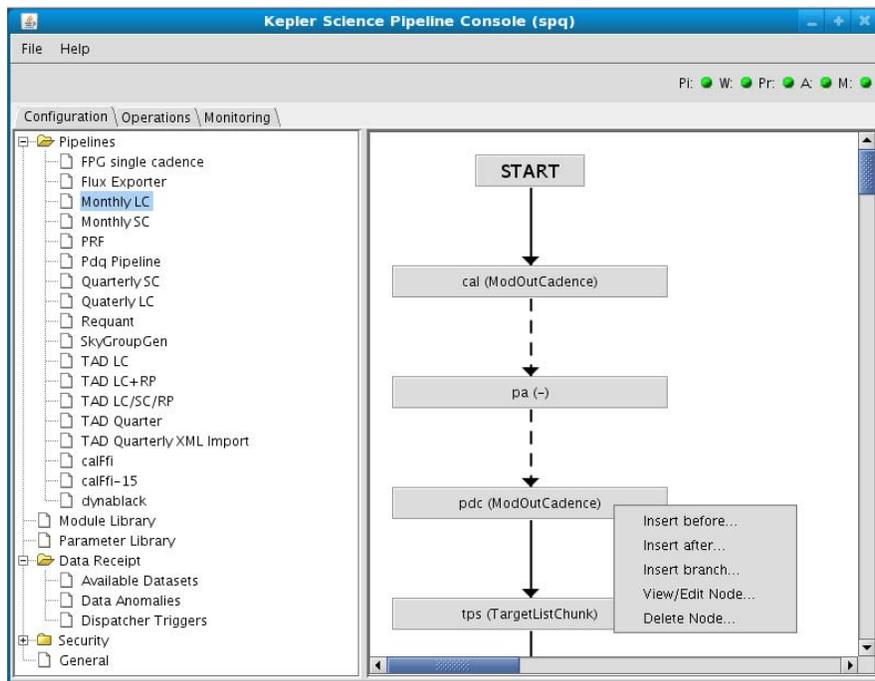
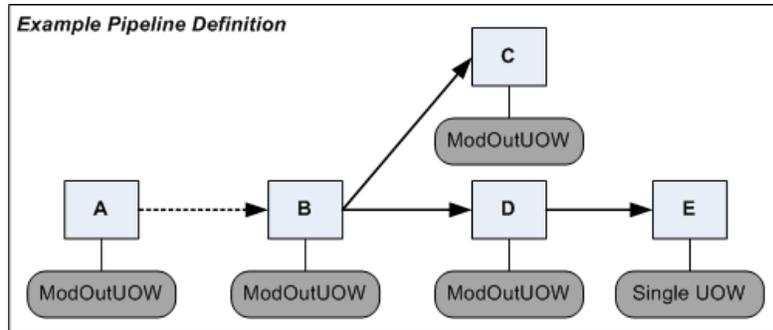


Figure 3: GUI: Pipeline configuration.

### 3.1 Pipeline definitions

Pipeline definitions specify the order of execution for modules, the unit-of-work generator for each module, and the transition type between modules. A pipeline definition is a tree data structure where each node in the tree is associated with a module library definition. While each node contains bidirectional references (parent and children), the tree can be considered a directed tree for execution purposes, since execution always progresses from the root node towards the leaf nodes.

Figure 4 shows an example PipelineDefinition with a tree of PipelineDefinitionNode objects, where A is the root node. Execution starts with the root node (A) and proceeds towards the child nodes. In cases where a node has multiple children, all of the children execute in parallel. In this example, the jobs for C and D execute in parallel following the completion of the jobs for B. Execution of a node consists of executing all of the jobs generated for the node, as described in Section 4.



**Figure 4:** Sample pipeline definition.

Each `PipelineDefinitionNode` must specify a unit-of-work generator and a transition type (Section 4.5). The generator is specified using the fully qualified name of the Java class that implements `UnitOfWorkGenerator`.

### 3.2 Module library

A module definition specifies the fully qualified name of the Java class that implements the module. This class must extend the abstract class `PipelineModule`. The module definition may also contain the name of an external executable (such as a MATLAB executable in the case of many of the *Kepler* pipeline modules) that is invoked as part of the job processing. Module definitions configured in the database are known collectively as the module library. Individual module definitions in the library are shared by the pipeline definitions that use them.

### 3.3 Parameter library

The parameter library is a collection of parameter sets. Each parameter set contains a collection of typed parameters represented by an application-provided Java class that implements the framework `Parameters` interface. The `Parameters` interface is simply a marker interface, so it contains no methods or fields. The operator can assign parameter sets to pipeline modules and unit-of-work generators as part of a trigger configuration (Section 3.4). The parameter library preserves all versions of the parameters and the framework keeps track of the specific versions used by each pipeline instance as part of the data accountability record (Section 3.6).

### 3.4 Trigger definitions

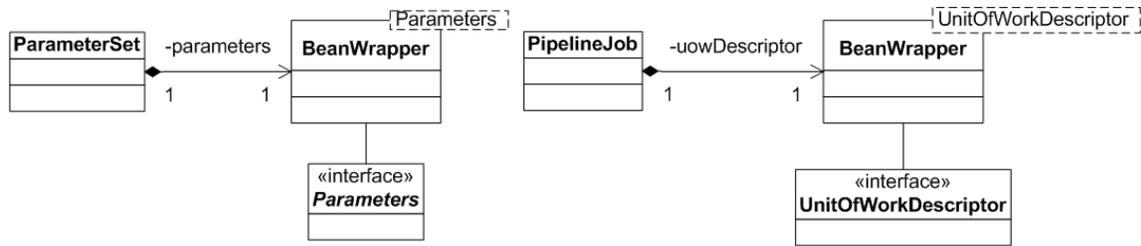
Pipeline triggers are used to launch new pipeline instances. The main purpose of a trigger is to maintain a mapping between the pipeline definition that will be used to create new pipeline instances and the parameter sets from the parameter library that will be used for those instances. The trigger also contains the priority that will be used for the new instances. Triggers can be fired by the operator using the GUI, or they may be fired programmatically, such as when a data ingest completes.

Each pipeline module implements the `requiredParameterTypes` abstract method of the `PipelineModule` to specify the parameter set types required at run time. The operations GUI verifies that all parameter needs are specified in the trigger before allowing an operator to fire the trigger.

### 3.5 Generic persistence of application objects

Application objects such as `Parameters` and `UnitOfWorkDescriptor` must be stored in the database as part of the pipeline configuration (in the case of `Parameters`) and job metadata (`UnitOfWorkDescriptor`). To avoid the need to customize the framework database schema for each application, these objects are persisted by the framework in a generic fashion, as shown in Figure 5.

The contents of each `Parameters` class and `UnitOfWorkDescriptor` are persisted to the database using the `BeanWrapper` class. `BeanWrapper` is a template class that can convert any class that conforms to the JavaBeans specification<sup>16</sup> to and from a collection of name/value (`String/String`) pairs. Because these classes conform to the JavaBeans specification, the framework can provide a generic GUI editor (a JavaBeans property sheet editor) and can persist them in a generic fashion. All primitive Java types, plus `String` (including arrays of these types), are supported.



**Figure 5:** Generic persistence Parameters class and UnitOfWorkDescriptor with BeanWrapper.

BeanWrapper utilizes the Apache Commons<sup>12</sup> BeanUtils package to convert from name/value pairs to the typed fields of the wrapped class and vice-versa. The BeanWrapper class is also a Hibernate class and contains the following fields, which are persisted to the database:

- **classname** : `String` – The fully qualified Java class name of the wrapped class
- **properties**: `Map<String, String>` – The names and values of the fields in the wrapped class in `String` form

Note that the wrapped class itself is not persisted except via the properties field indicated above. While parameter values are persisted as `String` regardless of type, their type information is retained in the wrapped class. This type information is used to validate parameter values entered by the operator via the configuration GUI or via XML import.

### 3.6 Locking and versioning of configuration entities

For data accountability purposes, the state of the pipeline configuration data model must be preserved each time a new pipeline instance is launched. Typically, launching new pipeline instances occurs more frequently than changing the configuration, so it would be inefficient to copy the configuration each time a pipeline instance is launched. Additionally, configuration changes typically involve only a small portion of the configuration, such as changing a handful of parameters, so it is equally inefficient to make a complete copy of the configuration each time the operator makes a change. For these reasons, a copy-on-write versioning and locking mechanism is implemented for each main entity type. `PipelineDefinition`, `PipelineModuleDefinition`, and `ParameterSet` are independently versionable. `PipelineDefinitions` are versioned as a unit. The nodes that make up the pipeline definition are not independently versionable.

Each of these versionable entities contains a version field and a locked field. When a new pipeline instance is launched, the latest versions of all entities referred to by the trigger used to launch the pipeline are marked as locked (if they are not already locked), and references are made between the pipeline instance and the specific versions of the entities that were used. When an operator attempts to modify a locked entity, a copy of the entity is created with an incremental version number and the framework sets the locked flag on the new version to false. Modifications are then made to the new version of the entity. This approach guarantees that once a particular version of an entity is bound to a pipeline instance, it will never change, thereby solidifying the data accountability record for that instance. These versioning and locking operations happen automatically and are transparent to the operator.

## 4. PIPELINE EXECUTION

Pipeline execution consists of executing the individual modules that make up the pipeline definition, starting with the first module and ending with the leaf modules. The start and end modules can also be overridden for a given pipeline instance, allowing execution of small segments of a pipeline. Execution of a module consists of executing all jobs associated with the module. There is one job per unit of work, so the number of jobs is determined by the unit-of-work generator configured for the node.

Jobs are processed by workers that run on the cluster nodes. There is one worker process per cluster node, but each worker process can be configured to run any number of threads, typically one thread per available CPU core. Jobs are distributed to workers via a Java Message Service (JMS) queue, using ActiveMQ<sup>15</sup> as the JMS provider. Jobs are not pre-assigned to specific workers; rather, jobs are placed on the queue where worker threads can claim them when they are ready to accept a new job. In this way, jobs are dynamically load-balanced across the cluster. Workers can be dynamically added or removed from the cluster without configuration changes; newly added workers will simply start processing pending jobs.

There is no centralized controller for pipeline execution. Instead, the pipeline transition logic is performed by workers in a distributed fashion. As each job completes on a particular worker machine, the worker executes transition logic that decides what to do next, generating jobs for the next module if necessary. Module transitions (see section 4.5) can be synchronized (i.e., all jobs for a given node must complete before the next node starts) or unsynchronized (i.e., jobs for a given unit of work can start for the next node as soon as the same unit of work for the current node completes). The framework component responsible for pipeline launching and transition logic is known as the pipeline executor.

## **4.1 Pipeline launching**

Pipeline launching consists of creating new `PipelineInstance` and `PipelineInstanceNode` objects, binding the latest versions of all parameter sets referenced by the trigger to these objects, locking configuration entities for the data accountability record, invoking the unit-of-work generator for the first module in the pipeline to generate `UnitOfWorkDescriptor` objects, creating new `PipelineJob` objects for those descriptors, and launching the jobs. These steps are described in more detail below.

### **4.1.1 Lock pipeline definition**

The first step in launching a new instance is to lock the associated pipeline definition to prevent subsequent modifications to the version of the definition used by the new instance. This preserves the state of the pipeline definition for purposes of maintaining the data accountability record for the new instance.

### **4.1.2. Bind parameters**

Parameter sets can be assigned to the trigger at the pipeline level or at the module level. Parameter sets that are assigned at the pipeline level are referred to as pipeline parameters, while parameter sets that are assigned at the module level are referred to as module parameters. Pipeline parameters are visible to all modules and unit-of-work generators in the pipeline, whereas module parameters are visible only to the module and the generator for the module. Pipeline parameters are useful in cases where the same parameter set types are needed by several modules and the operator wants to use the same parameter set instance for all of the modules in the pipeline definition. A given parameter set type may not be defined as both a pipeline parameter and a module parameter in the same trigger.

Triggers are mapped to parameter sets by name only, not to a specific version of the parameter sets. This is referred to as a *soft reference*. When a trigger is fired, a new pipeline instance is created and the instance is bound to the latest version of each parameter set. This is known as a *hard reference*. A hard reference between a pipeline instance and specific versions of parameter sets serves two purposes: First, it guarantees that all jobs that make up the pipeline instance will have a consistent view of the parameters, even if the operator changes the parameters while the pipeline is running. Secondly, it provides a data accountability record of the parameter values that were used for a given pipeline run. When parameter sets are bound to the new pipeline instance, the pipeline parameters are bound to the `PipelineInstance` object and module parameters are bound to the appropriate `PipelineInstanceNode` object. Figure 2 illustrates these associations.

### **4.1.3 Create instance nodes**

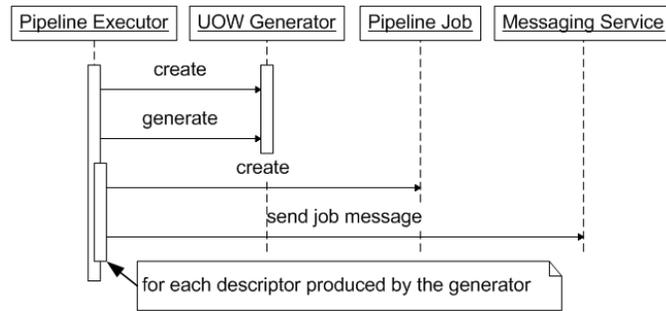
After parameters at the pipeline level are bound, instance nodes are created. There is one instance node per module in the pipeline definition. For each instance node, the associated module definition is locked and the parameters defined at the module level in the trigger are bound to the instance node.

### **4.1.4 Launch first module**

Finally, the first module of the pipeline is launched. The logic of launching a module is the same for the first module launched by a trigger as it is for later modules in the pipeline that are launched by a worker as part of the transition logic between modules.

## **4.2 Module launching and the unit of work**

Modules are launched by triggers when initiating a new pipeline instance or by worker machines when executing transition logic for synchronized transitions between modules. Figure 6 illustrates this process.



**Figure 6:** Launching a new pipeline instance node.

The decision of how to break up the work for a module is delegated to the unit-of-work generator. When the pipeline executor launches a module, it first instantiates a new instance of the unit-of-work generator using the name of the Java class defined in the pipeline definition for the module. Next, the executor combines the parameters that are bound to the pipeline instance with the parameters that are bound to the instance node and passes the combined set to the `generate` method of `UnitOfWorkGenerator`. This allows the parameters to control the behavior of the task generator. Passing the combined set allows an operator to set task generator parameters at the pipeline level or the module level.

The generator returns a list of objects that implement the `UnitOfWorkDescriptor` interface. This interface contains a single method, `displayString`, which returns a `String` containing a brief description of the unit of work for display in the console. The actual fields that describe the unit of work are contained only in the implementing class and are not used by the framework. They are simply stored and passed to the pipeline modules where the pipeline modules can use them to determine the dimensions of the data set to be retrieved from the data store.

The pipeline executor creates a `PipelineJob` object for each `UnitOfWorkDescriptor` object returned by the generator. The pipeline executor attaches the `UnitOfWorkDescriptor` object to the `PipelineJob` and persists it using the `BeanWrapper` mechanism described in Section 3.5.

Finally, the pipeline executor creates a `WorkerJobRequest` object that contains the pipeline job identifier, and this object is enqueued for the worker machines using JMS.

For unsynchronized transitions between modules, the pipeline executor will copy the unit-of-work descriptor from the previous module instead of calling the unit-of-work generator to generate new descriptors (see section 4.5).

### 4.3 Use of JMS for distributing jobs

Pipeline jobs are load-balanced across the pool of available workers using JMS. One key aspect of JMS is that it decouples senders from receivers. Instead of sending messages to a specific receiver, messages are sent to a named topic or queue. The JMS broker then forwards those messages to receivers that have registered an interest in those topics/queues. JMS supports two messaging paradigms: queues (point-to-point messaging) and topics (broadcast, or publish-subscribe messaging). A JMS queue is used to distribute pipeline jobs to workers. When a new pipeline instance is launched, a corresponding queue is created. This JMS queue has a name of the form `worker-queue-N`, where `N` is the pipeline instance identifier. A JMS message announcing the creation of the instance queue is then broadcast to all workers using the `pipeline-events` topic, to which all pipeline workers subscribe. The announcement tells the workers to start listening for new messages from the new queue.

Distributing jobs evenly to the available pool of workers is the responsibility of the JMS broker. The broker distributes the jobs to workers in a round-robin fashion, but jobs are allocated to a worker only when that worker signals that it is ready to start processing a new job. The worker does this by attempting to de-queue a message from the queue. Because jobs are not pre-allocated to workers, workers can be added to or removed from the cluster at any time and new worker machines will immediately start processing pending jobs in the queue.

Priority can be configured at the pipeline instance level, so multiple instances can be running at the same time, but workers will only process jobs for a given priority level if there are no available messages for higher priority levels.

## 4.4 Job execution

To execute a job, control must be passed to the custom code implemented for the pipeline module, and certain metadata managed by the framework must be made available to that code. These metadata include the unit-of-work descriptor associated with the job and the parameters configured by the operator.

### 4.4.1 Instantiating and invoking the PipelineModule object

Job processing is performed by creating a new instance of the implementing class for the pipeline module (specified by the `implementingClass` field in the `PipelineModuleDefinition` database object). This class must extend the `PipelineModule` abstract class. This class defines an abstract method called `processJob`, which is used to invoke the module. This method is defined as follows:

```
public abstract void processJob(PipelineInstance pipelineInstance, PipelineJob pipelineJob) throws PipelineException;
```

If the `processJob` method throws an exception, the framework considers the job to have failed and updates the `PipelineJob` metadata in the database accordingly. The framework does not execute the transition logic (see below) in this case. Access to the unit-of-work descriptor and the parameters is provided by the `PipelineJob` object via the following methods:

```
public <T extends Parameters> T getParameters(Class<T> parametersClass);
```

This method allows a pipeline module to retrieve the instance of the parameters for the specified parameter class. This method will retrieve parameters regardless of whether they were assigned at the module level or at the pipeline level in the trigger, so the application code does not need to be aware of the distinction.

```
public BeanWrapper<UnitOfWorkDescriptor> getUowDescriptor();
```

This method provides access to the unit-of-work descriptor for the job.

### 4.4.2 Distributed Transactions

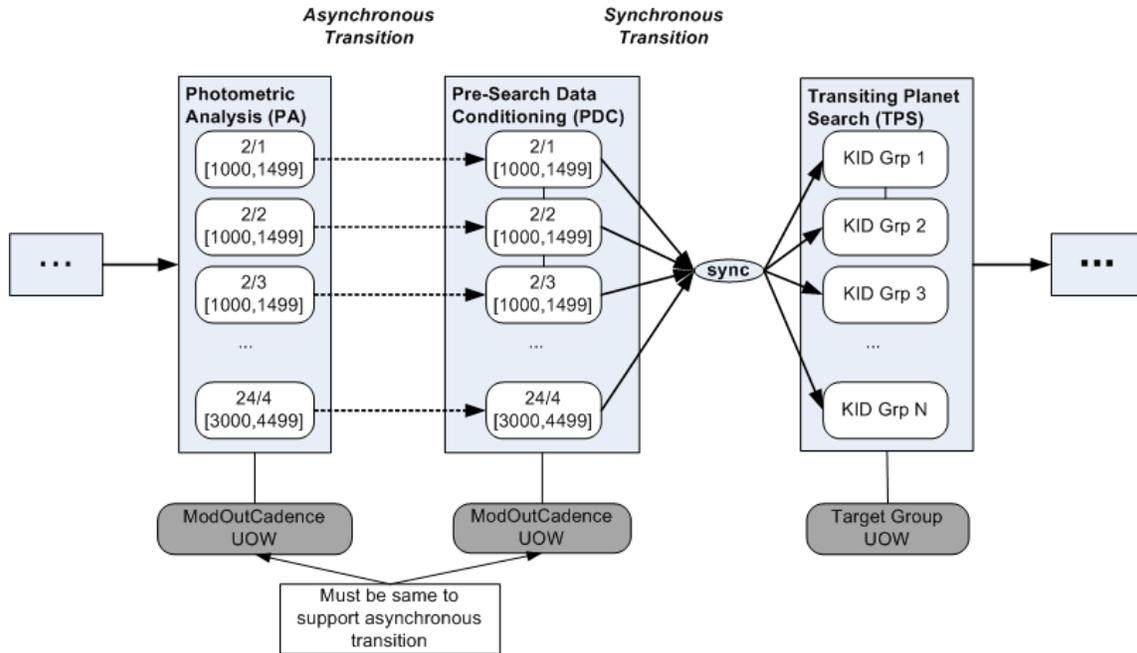
Each pipeline job is executed in the context of a single distributed transaction that includes both the relational database and the *Kepler DB*<sup>17</sup>. In order to guarantee ACID (atomicity, consistency, isolation, durability) properties for this transaction, each worker runs an embedded instance of the JBoss transaction manager. The transaction manager ensures that both the relational database and *Kepler DB* portions of the transaction will either succeed together, or fail together. If the job fails for any reason, all changes made as part of the transaction will be rolled back to their previous state in both databases.

### 4.4.3 Pipeline transition logic

When an operator launches a new pipeline instance, the pipeline executor only creates `PipelineJob` objects for the first module. Worker machines create jobs for subsequent modules as part of the transition logic execution that takes place immediately following successful job processing.

Transitions between pipeline modules can be synchronous or asynchronous. The operator specifies the type of transition in the configuration of the pipeline definition. For synchronized transitions, all jobs must be complete for the current module before jobs for the next module can be created. For unsynchronized transitions, as each job completes for the current module, a new job will be created for the next module using the same unit-of-work descriptor as the current job. This type of transition requires that both modules use the same unit-of-work type (i.e., they must be configured to use the same `UnitOfWorkGenerator`). Transitions between modules of different types must be synchronized.

Figure 7 shows an example of both types of transition. The *Kepler* PA and PDC modules both use the same unit-of-work generator and therefore support asynchronous transition. As each PA job completes, a PDC job with the same unit-of-work descriptor will be started, even if there are pending PA jobs for other units of work. The TPS module, however, uses a different unit-of-work type than PDC, so the transition must be synchronous. All PDC jobs must be complete before any TPS jobs start.



**Figure 7:** Synchronous vs. asynchronous transitions.

The pipeline instance node keeps track of the number of jobs associated with the module (grouped by job state). The first step in the transition logic is to update these metrics. Since workers running on different nodes may attempt to update these metrics at the same time, a locking mechanism is needed to serialize access to the critical block of code that retrieves the old values of the metrics, updates them, and stores the updated values back to the database. Java synchronization is insufficient since multiple workers running in different Java virtual machines on different hosts need to be synchronized. Instead, we lock the row in the database that contains the metrics using SQL ‘SELECT FOR UPDATE’ syntax. This approach locks the row until the transaction is committed and blocks other workers from reading the row until the lock is released, ensuring that the *read-increment-update* sequence is performed atomically. This lock is held for a very short period of time, so performance impact is negligible.

#### 4.5 Failure recovery

Since pipeline job failures are an inevitable part of pipeline operations, the framework provides a mechanism that allows an operator to restart failed jobs after the problem that caused the failure is corrected. An operator can rerun (via the operations GUI) all jobs in the `ERROR` state for a particular pipeline instance. Similarly, an operator can rerun individual failed jobs.

When a failed job reruns, the pipeline executor resets the state to `SUBMITTED` and updates the job state counts (decrement failed count, increment submitted count). Then, the pipeline executor places a new `WorkerJobRequest` object in the JMS queue for the pipeline instance using the messaging service.

There are certain scenarios where a job is interrupted, but the worker doesn’t get a chance to update the job state to `ERROR`. These include power failures, hardware failures, and software crashes. To allow recovery from these states, workers perform a check when they initialize. This check consists of changing the state to `ERROR` for all jobs in the `PROCESSING` state that are currently assigned to the worker doing the check. Since the worker is still initializing, these jobs cannot truly be `PROCESSING` and must have been previously interrupted. Putting the jobs in the `ERROR` state allows an operator to rerun them if desired. An operator can also manually force a job into the `ERROR` state (for example, if the worker machine goes permanently offline).

In order to preserve the integrity of the data accountability record, changes made to the pipeline configuration, parameter library, and model metadata library after the instance was launched will not be seen by the jobs of that instance, even if the changes were made before rerunning failed jobs. If fixing the problem that caused the error involves changing one of these elements, a new pipeline instance will need to be launched.

## 4.6 Pipeline monitoring

Running pipelines can be monitored using the monitoring GUI, organized by worker (Figure 8) or by pipeline instance (not shown). All pipeline processes broadcast a periodic heartbeat message using a JMS topic, and the monitoring GUI listens for these heartbeats. If the heartbeat for a process is not received within a configurable timeout, indicators in the GUI turn yellow, then red to alert the operator of the problem. The framework also provides an instrumentation API that is used to measure the execution time for various blocks of code (both in the framework and application code). These metrics are stored in the database and can be plotted in the monitoring GUI.

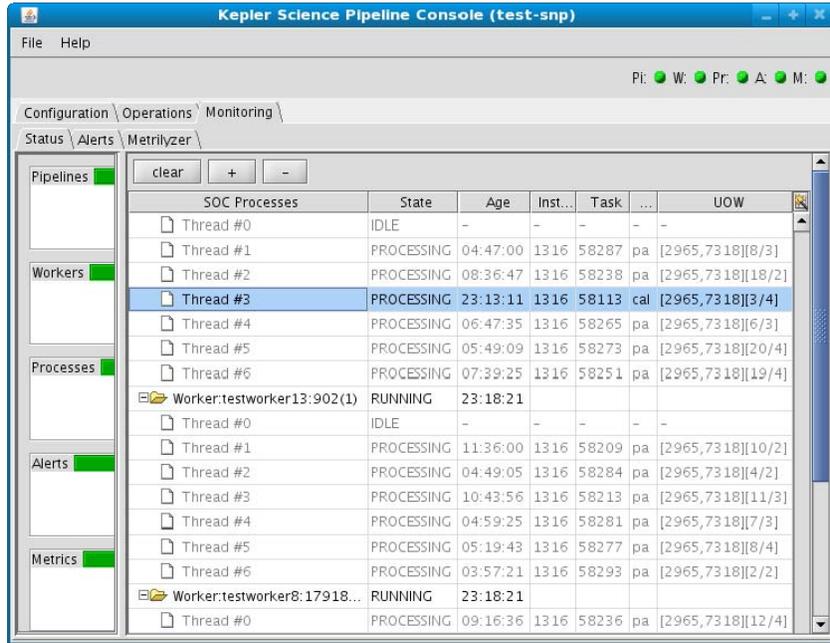


Figure 8: GUI: cluster status monitor.

## 4.7 Single-host deployment

In support of development and testing, pipeline developers can run all pipeline processes and third-party services on the same host, typically on a developer workstation. In this scenario, we use HSQLDB, an open source, pure Java relational database implementation. While HSQLDB does not support data volumes as large as those supported by Oracle, it is very easy to install, configure, and run, making it ideal for development. The *Kepler* SOC has developed test pipelines that include generation of simulated test data<sup>3</sup> for use in these scenarios.

## 5. SUMMARY AND CONCLUSIONS

The *Kepler* pipeline framework provides a scalable, generic infrastructure for distributed pipeline applications. Via a plug-in architecture, the framework allows applications to define a dynamic, customizable unit of work while leaving the details of job execution to the framework. The framework provides a comprehensive data accountability record, including parameter values used, at a minimal cost to the application developer. The framework includes a user-friendly GUI that provides configuration, execution, and monitoring features for the pipeline operator.

## ACKNOWLEDGMENTS

The authors would like to thank Patricia Carroll, Sue Blumenberg, and Greg Orzech for reviewing and editing this paper. We also wish to thank Bill Borucki and David Koch for their leadership of the *Kepler Mission*. Funding for the Kepler Mission has been provided by the NASA Science Mission Directorate (SMD).

## REFERENCES

- [1] Middour, C., et al., “*Kepler* Science Operations Center architecture,” *Proc. SPIE 7740*, in press (2010).
- [2] Hall, J., et al., “*Kepler* Science Operations Processes, Procedures, and Tools,” *Proc. SPIE 7737*, in press (2010).
- [3] Klaus, T. C., et al., “*Kepler* Science Operations Center pipeline framework extensions,” *Proc. SPIE 7740*, in press (2010).
- [4] Sun Developer Network, “Java Message Service Specification,” <http://java.sun.com/products/jms/>
- [5] Hibernate, “Hibernate,” <http://www.hibernate.org/>
- [6] Apache, “Apache Commons,” <http://commons.apache.org/>
- [7] JBoss, “JBoss Transactions,” <http://www.jboss.org/jbosstm>
- [8] Wu, H., et al., “Data validation in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [9] Koch, D. G., et al., “*Kepler* Mission design, realized photometric performance, and early science”, *ApJL*, **713(2)**, L79-L86 (2010).
- [10] Quintana, E. V., et al., “Pixel-level calibration in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [11] Twicken, J. D., et al., “Photometric analysis in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [12] Twicken, J. D., et al., “Presearch data conditioning in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [13] Jenkins, J. M., et al., “Transiting planet search in the *Kepler* pipeline,” *Proc. SPIE 7740*, in press (2010).
- [14] HSQLDB, “HyperSQL Database,” <http://hsqldb.org/>
- [15] Apache, “ActiveMQ,” <http://activemq.apache.org/>
- [16] Sun Developer Network, “JavaBeans Specification,” <http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html>
- [17] McCauliff, S., et al., “The *Kepler* DB, a database management system for arrays, sparse arrays and binary data,” *Proc. SPIE 7740*, in press (2010).