

The *Kepler* Science Operations Center pipeline framework extensions

Todd C. Klaus^{*a}, Miles T. Cote^b, Sean McCauliff^a, Forrest R. Girouard^a, Bill Wohler^a, Christopher Allen^a, Hema Chandrasekaran^c, Stephen T. Bryson^b, Christopher Middour^a, Douglas A. Caldwell^c,
Jon M. Jenkins^c

^aOrbital Sciences Corp./NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035

^bNASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035

^cSETI Institute/NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035

ABSTRACT

The *Kepler* Science Operations Center (SOC) is responsible for several aspects of the *Kepler Mission*, including managing targets, generating onboard data compression tables, monitoring photometer health and status, processing science data, and exporting Kepler Science Processing Pipeline products to the Multi-mission Archive at Space Telescope [Science Institute] (MAST). We describe how the pipeline framework software developed for the *Kepler Mission* is used to achieve these goals, including development of pipeline configurations for processing science data and performing other support roles, and development of custom unit-of-work generators for controlling how *Kepler* data are partitioned and distributed across the computing cluster. We describe the interface between the Java software that manages data retrieval and storage for a given unit of work and the MATLAB algorithms that process the data. The data for each unit of work are packaged into a single file that contains everything needed by the science algorithms, allowing the files to be used to debug and evolve the algorithms offline.

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

1. INTRODUCTION

The *Kepler* Science Operations Center (SOC)¹ is primarily responsible for analyzing raw pixel data from the *Kepler* spacecraft for up to 170,000 stars with the goal of identifying stars with possible transiting exoplanets for further follow-up research². To perform the analyses, the SOC developed a number of distributed software pipelines using the *Kepler* pipeline framework³. The framework is a generic platform designed for building applications that run as distributed pipelines with customizable units of work. Some of the *Kepler* pipeline modules perform data processing and analyses, while others produce products in support of the analyses or data collection; for example, generating onboard target and compression tables. Additional pipeline modules support automated testing of the code base, including generation of simulated test data.

Kepler pipeline modules are implemented using a combination of Java and MATLAB. Java classes store and retrieve data to and from a commercial, off-the-shelf relational database and a custom time series database known as the *Kepler DB*⁴, while algorithms that process the data are written in MATLAB. (Section 3 describes the Java/MATLAB interface in more detail.) Algorithm inputs and outputs pass between Java and MATLAB using a custom binary file format. These files are completely self-contained and self-describing. This allows scientists and developers to execute the science algorithms outside the pipeline without accessing the cluster databases.

2. KEPLER PIPELINES AND UNITS OF WORK

Before discussing the individual pipelines, it is important to understand how the unit of work is defined for each pipeline module. Raw *Kepler* data consist primarily of 30-minute (long cadence) pixel samples for up to 170,000 stellar targets

* Todd.C.Klaus@nasa.gov

collected from 84 charge-coupled device (CCD) output amplifiers (each one referred to as a CCD channel or a module/output). In addition, the SOC collects 1-minute (short cadence) samples for up to 512 targets. These raw data sets and the intermediate data sets produced by the pipeline modules must be partitioned into smaller units of work for distributed processing across the cluster. However, pipeline module algorithms have data dependencies that impose certain constraints on how data sets can be partitioned. For example, the Pre-search Data Conditioning (PDC) module⁵ needs data from all stars for a given module/output in order to create ensembles of targets for purposes of detecting and removing systematic errors, although each month of data can be processed independently. In contrast, the Transiting Planet Search (TPS) module⁶ can operate on a single target at a time, but needs as many time samples as possible (from the beginning of the mission) in order to maximize the detection of periodic transits.

Resource utilization imposes another set of constraints on the partitioning of data sets. For example, memory typically sets an upper bound on the dimensions of a single unit of work, while the number of CPU cores sets an upper bound on the number of units of work (data set partitions) that can be generated. In addition, small units of work result in more efficient load-balancing across the cluster than do large units of work. However, some algorithms produce better results with a larger data set, so often a compromise must be struck between improving the results and staying within the hardware constraints. For example, outlier detection will work better with more samples.

To this end, the pipeline framework allows the unit-of-work type to be customized for each pipeline module. The framework allows the application (in this case, processing *Kepler* science data) to customize the unit of work by creating what we call “unit-of-work generators”³. Unit-of-work generators are responsible for generating a set of “unit-of-work descriptors”—i.e., application objects that describe the dimensions of the data set to be processed by each pipeline job. For example, a unit-of-work descriptor might contain the module/output to be processed and the time range. The framework creates a pipeline job for each unit-of-work descriptor created by the generator. Generators can query the data store for products produced by the previous module. This allows generators to partition data sets differently depending on the previous module’s results. For example, the units of work for the Data Validation (DV) module⁷ include only the targets for which the Transiting Planet Search (TPS) module produced a positive result. This approach allows the unit-of-work descriptors to be dynamic and data-dependent.

2.1 *Kepler* unit-of-work generators

The framework allows each module in the pipeline to use a different unit-of-work generator, but multiple modules can share the same generator. Several of the generators developed for *Kepler* are described below.

2.1.1 Time Index Range generator

This generator subdivides a specified interval of absolute time index numbers (referenced to the start of the mission) into subintervals of a configurable size. For example, if a 90-day interval was specified and the maximum size per subinterval was set to 30 days, then three descriptors would be generated. The generator can optionally be configured to further subdivide these descriptors such that no unit-of-work descriptor straddles a quarter boundary (required by some algorithms). Attributes include the start and end absolute time index numbers.

2.1.2 Module/Output and Time Index Range generator

This generator creates one unit-of-work descriptor for each of the 84 CCD module/outputs. It can be configured to include or exclude specific module/outputs. Attributes of the descriptor include the module number, the output number, and the start and end absolute-time index numbers. The generator first creates a descriptor for each time-index interval and then subdivides each descriptor into a descriptor for each module/output. For example, if a 90-day interval with 30-day subintervals is specified, a total of 252 descriptors ($3 * 84$) would be generated.

2.1.3 Observed Targets generator

This generator queries the database for the complete list of all stellar targets observed by *Kepler* to date, sorts them by target ID, then partitions this set into a configurable number of chunks (at least as many chunks as the number of available CPUs on the cluster). Because they are sorted, attributes only need to include a start and an end target ID, rather than a complete list. When processing a job, the module code performs a similar query of all observed targets, except that the query is constrained by the start and end target IDs supplied in the unit-of-work descriptor.

2.1.4 Planetary Candidates generator

This generator queries the database for the unique set of stellar targets where a threshold crossing transit event was detected by the most recent run of the Transiting Planet Search module. These targets are partitioned first by sky group (sets of stars that fall on the same module/output during any given quarter), then into chunks of a configurable size (again based on the number of available CPUs). Attributes include a start and end target ID and the sky group ID.

2.1.5 Single Job generator

This generator always generates a single descriptor with no attributes and is provided by the framework. This generator is used when algorithm constraints dictate that no partitioning be done on the data set.

2.2 Kepler Science Processing Pipelines

There are three main science processing pipelines: one for long cadence (30-minute) data, one for short cadence (1-minute) data, and one to perform the transit search and subsequent validation of results.

Long and short cadence pipelines run monthly, following each spacecraft downlink. The SOC and *Kepler* Science Office analyze monthly results and use the results to fine-tune algorithms and parameters prior to quarterly reprocessing².

The TPS pipeline runs on all long cadence mission data collected to date to identify targets with possible planetary candidates. This pipeline runs quarterly, after the quarterly long cadence pipeline completes. It may also be run as needed—e.g., after a software or parameter change is made to TPS and/or DV.

The unit-of-work generator may vary from module to module, depending on the nature of the science algorithms. Figure 1 shows the sequence of modules for each pipeline, along with the unit-of-work generator configured for each module. The arrows between the modules indicate the type of transition: solid for a synchronous transition and dashed for an asynchronous transition³.

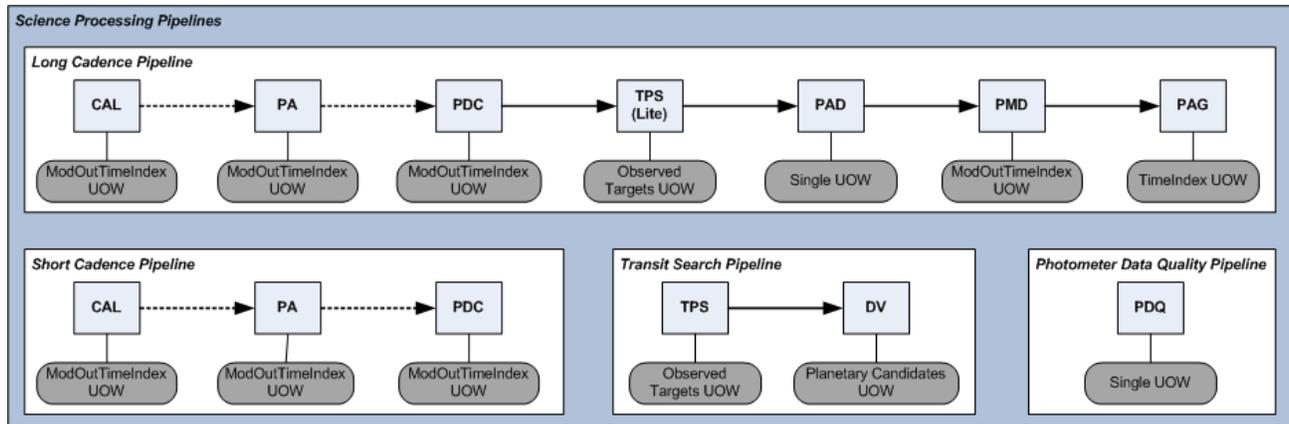


Figure 1: Science processing pipelines. Solid arrows indicate a synchronous transition. Dashed arrows indicate an asynchronous transition.

The following pipeline modules perform the primary analysis of the raw data downlinked from the *Kepler* spacecraft.

- **Pixel-level Calibration (CAL)**⁸: Corrects pixel data for instrument effects, such as electronics noise, smear (due to the lack of a shutter), and optical vignetting. The unit-of-work generator for CAL is Module/Output and Time Index Range. CAL needs all pixels from a full module/output at the same time because of the nature of the collateral data used for calibration, and to enable calculation of certain metrics at the module/output level. The time index range is typically no more than a month because of the memory requirements of the algorithm.
- **Photometric Analysis (PA)**⁹: Estimates and removes background and computes flux and centroid time series for all targets. Like CAL, the unit-of-work generator for PA is Module/Output and Time Index Range. PA needs a full module/output because of the way the background is estimated (with a polynomial that covers the entire module/output), and in order to calculate the pixel-to-sky coordinate mapping for each time index, which requires an ensemble of stars.
- **PDC**: Removes systematic errors from the flux time series for all targets. The unit-of-work generator for PDC is also Module/Output and Time Index Range. PDC needs an ensemble of stars from the same module/output in order to detect and remove systematic errors.
- **TPS**: Searches for periodic transit-like signatures in the corrected flux time series for targets of interest and estimates precision of the flux time series on transit time scales. The unit-of-work generator for TPS is Observed Targets. TPS can operate on a single target at a time, so the unit-of-work generator is typically configured to evenly divide the targets across the available workers in the cluster. TPS can run in “lite” mode,

where it only computes photometric precision metrics for the targets, or in “full” mode, where it searches for transits.

- **Data Validation (DV):** Performs a series of statistical confidence tests on the candidates identified by TPS in support of follow-up observations. Like TPS, DV can operate on a single target at a time, but DV is the most time-consuming module in the pipeline, so the number of targets per job is kept small (< 10) to improve the efficiency of the cluster and to prevent one bad target from stopping the entire unit of work from being processed. The unit-of-work generator is Planetary Candidates.
- **Photometer Attitude Determination (PAD)¹⁰:** Uses centroids and catalog coordinates to determine the pointing of the spacecraft over time. The unit-of-work generator is “Single” since PAD needs an ensemble of stars across the entire field of view.
- **Photometer Metrics Determination (PMD)¹⁰:** Calculates various metrics on the data to assist scientists in assessing the performance and health of the instrument. The unit-of-work generator is Module/Output and Time Index Range, since the metrics are computed on a module/output basis.
- **Photometer Metrics Aggregator (PAG)¹⁰:** Aggregates metrics computed by PMD into a single report covering the entire field of view. The unit-of-work generator is Time Index Range, since the goal of PAG is to generate a single report that covers all module/outputs for a specified time index interval.
- **Photometer Data Quality (PDQ)¹¹:** Performs analysis on a small subset of data downlinked every four days (vs. once per month for the full set) for the purpose of assessing the near-term performance and health of the instrument. The unit-of-work generator is “Single” because PDQ generates a report for the entire field of view.

2.3 Development and test pipelines

The SOC also developed several pipelines for automated testing and generation of simulated data. These pipelines contain modules that perform many tasks typically performed manually by SOC operators due to the various checks and balances that make up operations procedures. In a controlled test environment, however, these steps can be automated by the pipeline. This approach allows the SOC to fully automate end-to-end processing for testing purposes. Tests begin with an empty data store and consist of the following steps:

- Step 1: Initialize the database schema and clean out the data store.
- Step 2: Generate simulated test data with the end-to-end model of the *Kepler* photometer¹².
- Step 3: Ingest simulated test data.
- Step 4: Process simulated test data through the science pipelines.
- Step 5: Export pipeline products to FITS and XML files.
- Step 6: Validate exported products to verify that data flowed correctly through the pipeline.

The SOC uses the test pipelines to run a small data set through the science pipelines on a nightly basis as a regression test. This allows developers to detect problems introduced by changes to the code or parameters as soon as possible in each development cycle. Developers and scientists also use the test pipelines to troubleshoot problems or to experiment with changes to the algorithms prior to releasing the changes for processing flight data on the full cluster.

3. ANATOMY OF A PIPELINE MODULE

Data management software elements of the *Kepler* pipeline modules are written in Java, while the science algorithms are written in MATLAB. Java classes that wrap these algorithms and manage their inputs and outputs are referred to as module interfaces. Module interface code is responsible for retrieving inputs for a given unit-of-work descriptor from the databases (relational database and the *Kepler DB*⁴), passing inputs to the algorithm, and storing resulting outputs in the databases after the algorithm completes. This section describes pipeline modules that include a MATLAB algorithm component, but not all pipeline modules follow this model. Pure Java modules integrate with the pipeline framework the same way that MATLAB modules do. However, they do not invoke the serialization and execution services of the framework.

Figure 2 below shows the main steps involved in executing a pipeline job, starting with the invocation of the module interface code by the framework in the worker process (*Step 2*).

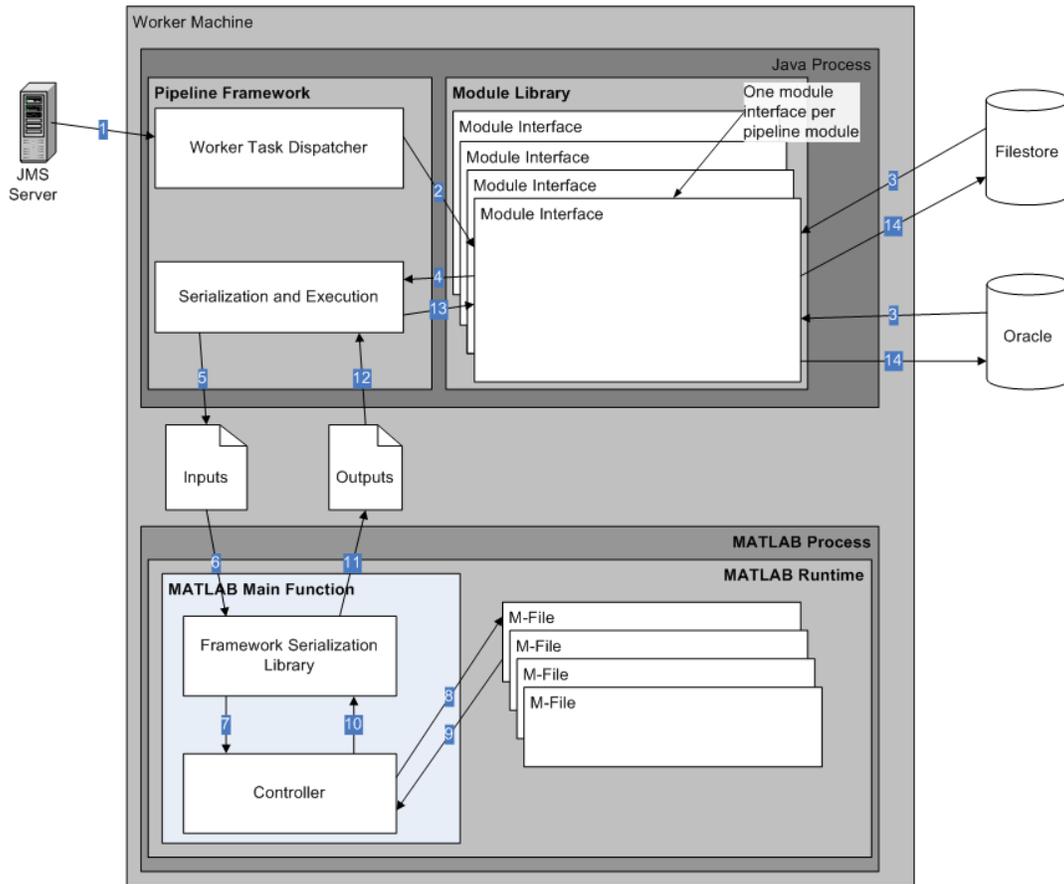


Figure 2: Module interface data flow.

The module interface class first retrieves inputs required by the module (constrained by the unit-of-work descriptor) from the databases (*Step 3*). It then aggregates these inputs into an object tree and passes the tree to the serialization and execution components of the pipeline framework (*Step 4*).

The framework includes a serialization component that can stream the object tree containing the algorithm inputs to a temporary file (*Step 5*). The execution component of the framework then launches the MATLAB process, passing the path to the temporary file (*Step 6*). (For details, see Section 3.3 and Section 3.4.)

The main function (entry point) of the MATLAB process invokes the framework serialization component on the MATLAB side to reconstitute the temporary file created in *Step 5* into a MATLAB struct (*Step 6*), then invokes the MATLAB controller for the module (*Step 7*). The controller then invokes the various algorithms needed to process the job (*Steps 8 and 9*), then returns control to the main function (*Step 10*) along with a MATLAB struct containing the outputs of the module. The main function then serializes these outputs to a temporary file (*Step 11*) and the MATLAB process terminates.

Once the MATLAB process completes, the serialization component of the framework reconstitutes the outputs into a Java object tree (*Step 12*) and returns it to the module interface class (*Step 13*), where the outputs are stored in the databases (*Step 14*).

Because all access to the databases is done from the Java process, the serialized inputs file contains everything the algorithm needs to process the unit of work. This means that the algorithm has no external dependencies at run time other than the inputs file, so it can be reprocessed on a different machine (such as a developer or scientist workstation) with only the inputs file. This allows activities such as debugging a problem found during pipeline execution,

experimenting with algorithm or parameter changes, or data analysis to be performed without accessing resources on the cluster where the inputs file was originally created.

The following sections discuss the components that support these steps in more detail.

3.1 Module interface classes

This section describes the main classes involved in the execution of a pipeline job, including classes that are part of the framework and application-specific classes written by the module developer. Figure 3 shows the relationships between these classes using the TPS pipeline module as an example. The classes in the top two boxes are implemented in Java, and the classes in the bottom two boxes are implemented in MATLAB.

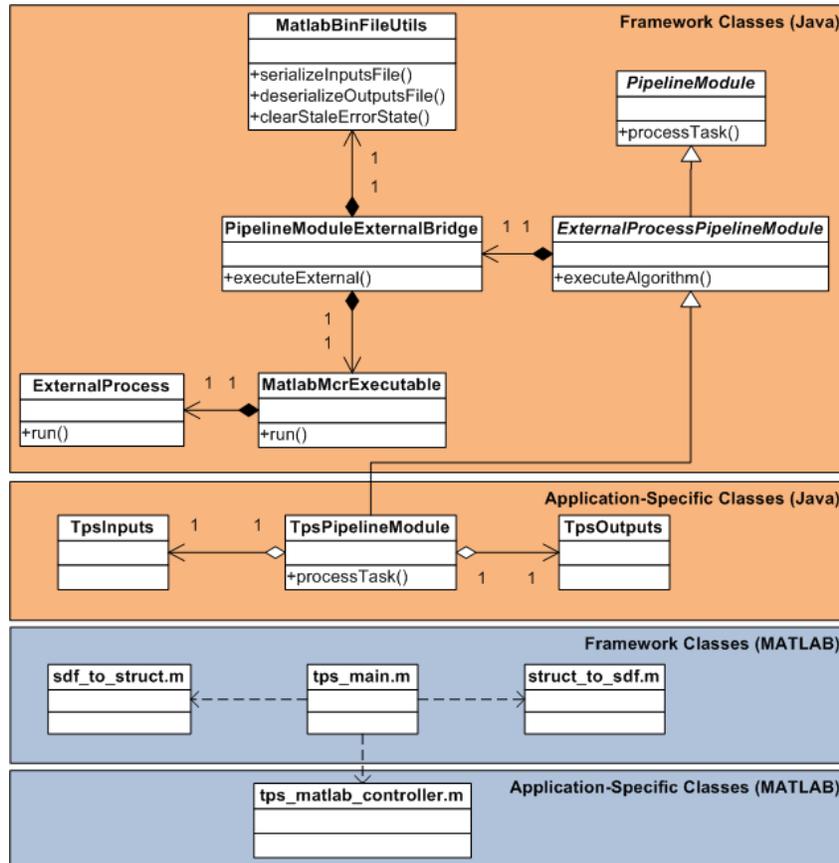


Figure 3: Module interface classes.

3.1.1 Application-specific Java classes

The `PipelineModule` abstract class is the main interface between the pipeline framework and the module interface code. All pipeline module classes must extend this class or a sub-class. Typically, modules that include a MATLAB algorithm extend the `ExternalProcessPipelineModule` abstract class (which extends `PipelineModule`). `ExternalProcessPipelineModule` provides easy access to the serialization and execution services of the pipeline framework via its `executeExternal` method. Pipeline modules that do not include a MATLAB component (Java-only pipeline modules) typically extend `PipelineModule` directly.

The main role of this class is to act as an entry point for the framework and to fetch the algorithm inputs from the databases and store the algorithm outputs to the databases.

3.1.2 Input and output classes

The input and output classes for the module interface contain the inputs and outputs of the MATLAB algorithm that are serialized by the framework for passing between the Java and MATLAB processes. Although Figure 3 only shows the `TpsInputs` and `TpsOutputs` classes, input and output classes typically contain many other nested classes.

3.2 Serialization library

As mentioned above, the framework uses files to pass MATLAB algorithm inputs and outputs between Java and MATLAB processes. These files are created and accessed using serialization classes that are part of the framework. In order to simplify MATLAB code that reads and writes these files, a simple serialization method was chosen over existing serialization methods, such as Java object serialization. Because data files can be large even for a single unit of work, we chose a binary format over a text format like XML.

The serialization library supports classes that contain only Java primitives, *Strings*, *Enums*, *Dates*, the standard Java collection classes *List*, *Set*, and *Map* (containing supported types), and other classes that contain fields of these types. These restrictions ensure that the contents of the input and output object trees can be properly represented in MATLAB structs using built-in MATLAB types.

The serialization file is known as an SDF file, for “self-describing format.” SDF files contain a header with a data dictionary that describes the classes of all of the objects contained in the file and a body that contains the serialized objects. The endianness of the platform is used, which means that the binary files are not portable between hardware with different endianness. Figure 4 shows the set of classes and MATLAB functions that read and write SDF files.

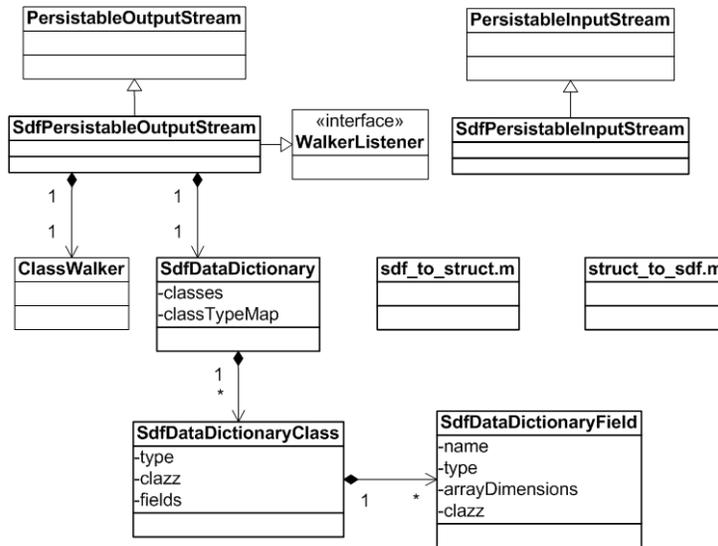


Figure 4: SDF Classes.

3.2.1 SDF data dictionary

The data dictionary of the SDF file is created with the help of the *ClassWalker* class, which provides a static analysis of a class tree. *ClassWalker* walks a class tree and feeds class and field descriptors to a registered listener that implements the *WalkerListener* interface (see Figure 5).

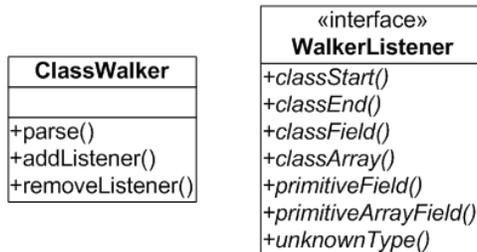


Figure 5: ClassWalker classes.

The *ClassWalker* class uses the Java Reflection API to identify all of the fields of a specified class. Each time a previously unseen class is encountered, *ClassWalker* fires a *classStart* event, followed by events for all of the fields

of that class, and then a `classEnd` event. The `classStart/classEnd` event pairs will be nested to match the static structure of the classes. Classes are only described once, even if they are referenced multiple times in the class structure.

The `SdfDataDictionary`, `SdfDataDictionaryClass`, and `SdfDataDictionaryField` classes use this information to build the data dictionary portion of the SDF header.

3.2.2 SDF body

The body of the SDF file is created in Java code using the `PersistableOutputStream` and `SdfPersistableOutputStream` classes. The `PersistableOutputStream` class uses the Java reflection API to determine the fields that make up the objects to be serialized and their types. This process starts with the root object to be serialized and proceeds recursively through all of the contained objects. For each field, `PersistableOutputStream` calls `SdfPersistableOutputStream` to write the field value to the SDF file via the abstract methods shown in Figure 6. MATLAB code uses the `struct_to_sdf.m` function to serialize MATLAB structs to SDF files.

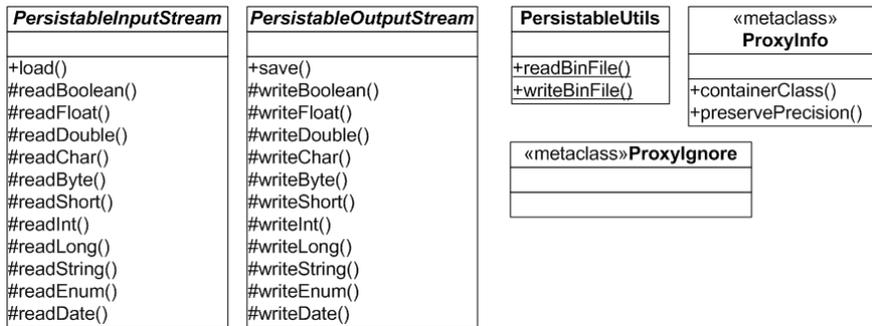


Figure 6: Framework abstract serialization classes.

Deserialization is handled in a similar way. In Java code, the `PersistableInputStream` class uses `SdfPersistableInputStream` to read the individual fields that make up the reconstituted objects from the SDF file. MATLAB code uses `sdf_to_struct.m` to deserialize an SDF file to a MATLAB struct.

This design allows changes to the external file format by simply using different implementation classes to read and write the fields.

`PersistableUtils` provides convenience functions to read and write SDF files. The `ProxyIgnore` annotation class allows the developer to control the behavior of the serializer on a field-by-field basis. Fields that are tagged with the `ProxyIgnore` annotation are ignored by the serialization library.

3.3 Execution library

The framework provides several classes on the Java side to manage the lifecycle of the external MATLAB process, including passing inputs and outputs (see Figure 7).

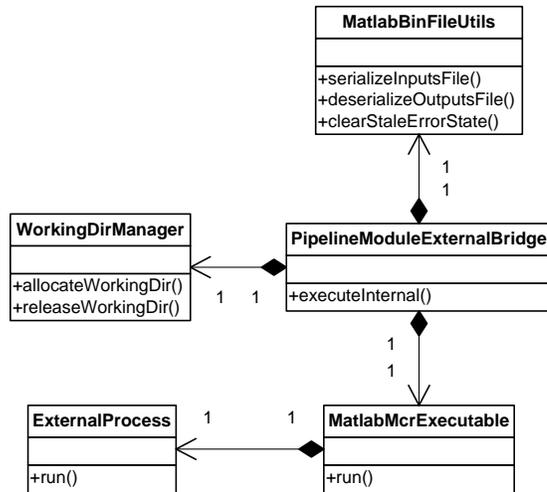


Figure 7: Framework External Process Management Classes.

On the Java side, the `PipelineModuleExternalBridge` class acts as the coordinator for the entire process of serializing the inputs to an SDF file, launching the external MATLAB process and waiting for it to complete, and deserializing the outputs from an SDF file produced by the MATLAB process. Figure 8 shows the main steps of this process. Note that `MatlabBinFilesUtils` uses the serialization library described in section 3.2 (not shown in Figure 8).

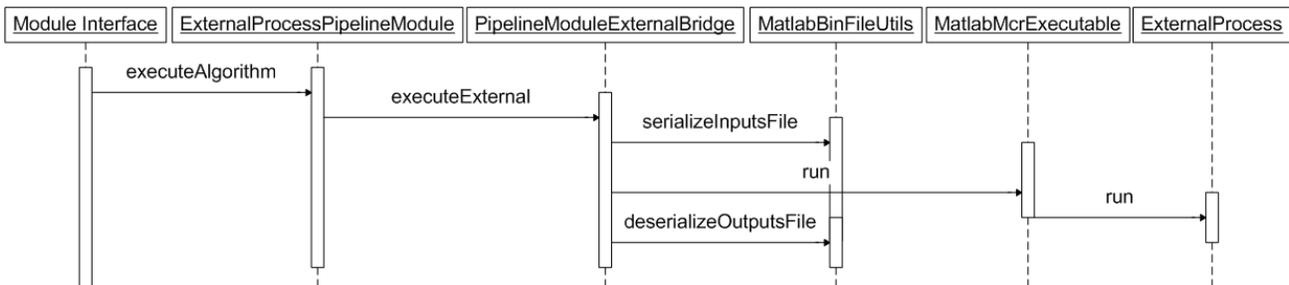


Figure 8: Execution of an External Process.

First, the module interface code invokes the `executeAlgorithm` method of `ExternalProcessPipelineModule`, passing it a fully populated inputs object and an empty outputs object. This call will block until the external process completes and the outputs are deserialized.

The `executeAlgorithm` method creates a new instance of `PipelineModuleExternalBridge` and invokes the `executeExternal` method on that object.

`PipelineModuleExternalBridge` uses `WorkingDirManager` to initialize the working directory for the MATLAB process. This directory contains the input and output SDF files for the algorithm as well as any temporary files created by the algorithm. The contents of the working directory are useful for debugging problems or for off-line analysis of the data, so those data are not deleted immediately after job processing completes. Instead, `WorkingDirManager` keeps track of all of the working directories created by a worker process and deletes the oldest directories when the total number of directories exceeds a configurable threshold.

`PipelineModuleExternalBridge` then uses `MatlabBinFileUtils` to serialize the inputs object tree to an SDF file. `MatlabBinFileUtils` is a thin wrapper around the `Persistable*Stream` classes and manages such details as filenames for SDF files and error files generated by the MATLAB process (described below). Some pipeline modules invoke the MATLAB algorithm multiple times for a single unit of work; for example, when memory constraints prevent the algorithm from accommodating all of the data for the unit of work. `PipelineModuleExternalBridge` supports this by maintaining a sequence number that is incremented for each invocation. This sequence number is used in the

filenames of the SDF files and in MATLAB log files so that files from one invocation do not overwrite files from a previous invocation.

At this point, the inputs SDF file exists in the MATLAB working directory and the MATLAB process can be launched. This is done by calling the `run` method on `MatlabMcrExecutable` class. This class manages the setup needed to run an executable generated by the MATLAB compiler, including telling the process where to find the MATLAB compiler runtime libraries.

The `ExternalProcess` class launches and monitors the MATLAB process. The `stdout` and `stderr` stream outputs from the MATLAB process are redirected to log files in the working directory. `ExternalProcess` also supports a retry mode, where the external process will be relaunched a configurable number of times in the event of an abnormal exit. We use a retry count of one to minimize the chances of stalling the pipeline due to a transient error.

On the MATLAB side, the main function deserializes the inputs file using the framework function `sdf_to_struct.m`. Control is then handed over to the MATLAB controller for the module. The algorithm developer is responsible for writing the controller function. By convention, this call takes the following form:

```
outputsStruct = tps_matlab_controller(inputsStruct);
```

If the controller call completes successfully, the resulting outputs struct is serialized to an SDF file using `struct_to_sdf.m`.

If the controller or other code called by the controller (i.e., other algorithm code or built-in MATLAB functions) throws an error, the main function catches the error and serializes the error message and the call stack to an error SDF file for subsequent processing by the Java side (as described above).

After the process completes (or exhausts the retry attempts), `MatlabBinFileUtils` checks for the presence of an error SDF file. The auto-generated MATLAB main function creates this file if the algorithm throws an error. The error SDF file contains the text of the error message and the MATLAB stack at the time of the error. If the error file exists, the main function deserializes it and throws an exception containing the error. The framework code that initiated the processing catches the exception and rolls back the job transaction. The module code does not need to handle the exception and assumes that if control is returned from the `executeAlgorithm` call, MATLAB processing completed successfully. The error information propagates to the operator via the console, and the operator can then forward it to the developer to aid in debugging problems. If the error file is not present, then the MATLAB process completed successfully and the outputs SDF file is deserialized. The resulting outputs object tree is then returned to the module code for storage in the databases.

4. SUMMARY AND CONCLUSIONS

To process *Kepler* data, the SOC has developed several pipelines using the *Kepler* pipeline framework. We have described the custom unit-of-work generators that we developed for the *Kepler* pipeline modules, and how the various pipelines were configured. We have described how we used a combination of Java (for data management) and MATLAB (for scientific algorithms) code to implement the various pipeline modules, and how data are shared between these two components via files that are also useful for offline debugging and analysis.

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] Koch, D. G., et al., “*Kepler* Mission design, realized photometric performance, and early science,” *ApJL*, **713(2)**, L79-L86 (2010).
- [3] Klaus, T. C., et al., “*Kepler* Science Operations Center pipeline framework,” *Proc. SPIE 7740*, in press (2010).
- [4] McCauliff, S., et al., “The *Kepler* DB, a database management system for arrays, sparse arrays and binary data,” *Proc. SPIE 7740*, in press (2010).
- [5] Twicken, J. D., et al., “Presearch data conditioning in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [6] Jenkins, J. M., et al., “Transiting planet search in the *Kepler* pipeline,” *Proc. SPIE 7740*, in press (2010).
- [7] Wu, H., et al., “Data validation in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [8] Quintana, E. V., et al., “Pixel-level calibration in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [9] Twicken, J. D., et al., “Photometric analysis in the *Kepler* Science Operations Center pipeline,” *Proc. SPIE 7740*, in press (2010).
- [10] Li, J., et al., “Photometer performance assessment in *Kepler* science data processing,” *Proc. SPIE 7740*, in press (2010).
- [11] Jenkins, J. M., et al., “Semi-Weekly Monitoring of the Performance and Attitude of *Kepler* Using a Sparse Set of Targets,” *Proc. SPIE 7740*, in press (2010).
- [12] Bryson, S. T., et al., “The *Kepler* end-to-end model: creating high-fidelity simulations to test *Kepler* ground processing,” *Proc. SPIE 7738*, in press (2010).