SuperTask WG report

G. Dubois-Felsmann, J. Bosch, M. Gower, N. Pease, A. Salnikov, J. Sick

Purpose of the working group

- Resolve long-standing design questions regarding the packaging of Science Pipelines* code for execution and the interfaces for provision of data
  - The key use case is the definition and execution of production pipelines
  - Pipeline scientific logic is defined in terms of instances of (I/O-free) Task subclasses, with configuration; how is this handed off for production?
  - A layer above Task has to do the I/O (currently in subclasses of CmdLineTask)
  - We would like to have an agreed-upon way for steps in pipelines to express their I/O grouping and parallelization constraints to a workflow system

* and other applications, e.g., data quality monitoring

Working group mechanics

- The central principle is to get authoritative input from stakeholders and developers

• Jim Bosch	Science Pipelines
• Michelle Gower	Workflow / NCSA
• Jonathan Sick	SQuaRE
• Andy Salnikov	Lead developer
• Nate Pease	Butler lead developer
• Gregory Dubois-Felsmann	Architecture (chair)

- Original plan was to wrap up by April 12 with recommendations
- WG members were told to plan for an 0.2 FTE level of effort

Activity to date

- We met for 4 hours / week in March and April
  - WG members also spent considerable offline time
- Compared to the plan:
  - Took longer to go through all the requirement-defining inputs, mainly:
    - Understanding the data-grouping and parallelization structures needed by the pipelines
    - Understanding the production-control issues articulated by NCSA
  - Discovered that "DataId-mapping" (*vide infra*) was the crux of the design
- There are still some unresolved issues, but we are ready to report and to get feedback from the larger group

Summary of the idea

A Pythonic API for defining a sequence of processing steps,

- each built from Tasks, with a pex_config-based configuration,
- exposing their data-grouping requirements and associated changes of parallelization,
- relying on the referencing of inputs and outputs in terms of an evolution of the DataId concept,
- and on I/O through a Butler

and a generic mechanism for: invoking any such sequence of processing

- on a defined set of input and/or output DataIds,
- computing a DAG for the processing steps,
- as well as a complete manifest of input and output (DataId, type) pairs for each execution of each processing step,
- and executing it in a variety of processing environments.

An indicative reference case for contrast

- A more explicitly command- and file-based approach
- Processing steps as Unix commands
  - Implemented as a refactored evolution of CmdLineTask
- Explicitly file-based input and output specifications
- Data dependencies expressed externally to the processing steps, perhaps in a Common Workflow Language-like manner
- Substantial intelligence moved to a scripting layer above the step-commands

Key constraining requirements

- Pythonic API
- Support of development and production through a common interface
- Ability to predict all inputs and outputs to support data staging and creation of "walled gardens" for production jobs
- Butler I/O
- Flexible parallelization changes
- Parallelization specifications represented by code in each processing step (i.e., the grouping constraints of co-addition go along with a co-addition SuperTask)

A more detailed requirements-oriented view

- Provide interface for delivering a complete algorithmic work specification (a “Pipeline”) from Science Pipelines to an execution system, notably the production system.
    - A Pipeline specification fully represents the transformations to be performed, but does not represent the specific data to which the transformation is to be applied.

Pipeline specifications

- A Pipeline specification must:
  - Specify the units of code to be run and a sequence in which they are to be run.
    - The sequence specification need only be a explicit ordered list. It is not required to support looping, branching, or step-skipping.
  - Specify the configurations of all the units of code to be run, using the existing LSST stack "pex_config" mechanism.
  - Specify how datasets must be grouped for each step in the sequence. (E.g., for the inputs to coaddition.)
  - Permit each step in a sequence to have a different required data grouping, and therefore an implied change of permissible parallelization.
  - Support the organization of work in terms of Tasks, assumed to obey the "no-I/O rule", i.e., operating solely on Python-domain objects, and supply the configurations they require.

Pipeline specifications (2)

- Permit a common supervisory framework to execute any Pipeline, based solely on information obtained programmatically from the Pipeline specification.
- Provide APIs that support “pre-flight” and “run” phases of execution organized by the supervisory framework. These are further constrained in the next section; the basic definition is:
  - Pre-flight: support the computation of a DAG for the application of a Pipeline to a specification of inputs and/or outputs as DataIds.
  - Run: invoke the units of work defined in the DAG (pairings of a processing step with its input and/or output DataIds)
- Provide APIs that support resolution of full DataId specifications for “implied inputs” such as calibration frames.

Pipeline specifications (3)

- Be able to use a Butler instance (provided by the supervisory framework) to perform all required I/O for each step in the "run" phase.
- Use the configuration mechanism to control the Butler dataset types used by each processing step
    - Some exceptional cases requiring direct I/O to databases may be excluded from this restriction. (E.g., to permit database ingest itself to be handled in this framework.)

Pipeline specifications (4)

- Support pre-execution programmatic insertions to an already-specified Pipeline's processing sequence, with the intent that this interface could be used by a supervisory framework to add, e.g., quality analysis or other monitoring steps.
  - These must be capable of being captured for purposes of provenance recording.
- Support pre-execution programmatic overrides to the configurations specified for a Pipeline.
  - These must be capable of being captured for purposes of provenance recording.
  - We assume that it will continue to be possible to capture the full run-time configuration as a snapshot.

Supervisory framework

- The supervisory framework must:
  - Be designed to support the creation of multiple specializations of the supervisory framework for different execution environments.
    - It must support specializations suitable for at least the following execution environments:
      - Level 2 (DRP), CPP, and other non-real-time production
      - Level 1 near-real-time production
      - Interactive, command-line execution
      - Execution from a Python prompt (e.g., in a notebook)
      - Execution in a persistent server (e.g., to support the SUIT Portal)
      - Automated CI and verification testing

Supervisory framework (2)

- Provide a common implementation of the logic required for interpretation of the Pipeline steps and their parallelization changes.
  - The basic logic would be applied uniformly in all specializations.
- Support “pre-flight” of execution of a Pipeline on a specified set of inputs and/or desired outputs, resulting in a DAG for the processing, with the nodes in the DAG being the units of work to be executed, each one being:
  the combination of one of the processing steps in the Pipeline with a complete list of the inputs and outputs for each job (specified as pairs of fully specified DataIds and Butler dataset types).
  - Note that a specific supervisory framework specalization is free to consolidate these units of work “vertically” (along the processing flow) and/or “horizontally”, for efficiency.
- Provide a serialization form for the results of pre-flight, so that they can be computed in one place and executed in another

Supervisory framework (3)

- Create and supply the Butler required to support the I/O that will be performed in the “run” phase, for each unit of work.
- Provide for “non-intrusive provenance” discovery (desirable)
  - Tracking the actual execution of units of processing on input datasets, their outputs, and their associated Task structure and configuration
  - Done via instrumentation of Butler calls
  - Recording mechanism is TBD
- Among other housekeeping functions:
  - Set up logging for the execution of each processing step.
  - Set up any provenance-recording mechanism(s) for the execution of each processing step, either or both of:
    - The above coarse-grained non-intrusive provenance
    - Any “fine-grained” (i.e, intrusive) provenance recording (c.f. provenance_proto)

Supervisory framework (4)

And, the central issue, as it turned out:

- Accept Pipeline "campaign" specifications including:
  - Specifications of outputs to be produced from a universe of available inputs, with the Pipeline processing the minimal set of inputs required to make the outputs
  - (Possibly) Specifications of inputs to be processed, with the Pipeline producing all possible outputs deriving from these inputs
  - Specifications of both inputs and outputs, with the inputs specifications treated as restrictions on the universe of available inputs; i.e., "intersection" logic is applied

(This is an area we are still actively thinking about as we write up.)

Requirements from SQuaRE, SUIT

- The principal contributions to the requirements that were strongly driven by SQuaRE and/or SUIT:
  - The ability to insert additional steps into an already-defined Pipeline (e.g., for data quality monitoring).
  - The ability for the supervisory framework to be built into a persistent server.
  - Strong support for the interface being Pythonic.

Quick design tour

Major open issues

- The details of the revision of the DataId concept
  - We believe making it more concrete and limited will make it substantially easier to implement

- The specification of concrete Pipelines in terms of SuperTasks; is it
  - Done purely Pythonically – i.e., a Pipeline spec being essentially a piece of Python code that constructs an aggregate of SuperTasks – or
  - Done with some sort of YAML, JSON, or other text-file spec?

Other open issues

- Use of community tools
  - It appears likely that community tools like the Common Workflow Language are compatible with the design we have developed
  - It may even be possible to adopt elements of community Python-based CWL-oriented workflow systems (e.g., Toil) to help us build frameworks
  - Additional research would be useful
- Configuration
  - There is an existing issue of how to separate “algorithmic configuration” (that has effects on the formal data products of a Pipeline) from “manner-of-execution configuration” (which should not). The former should be protected against change separately from the latter. This should be a feature of the configuration mechanism and be exposed through SuperTask. (One crucial issue, for example, is that data processed with different “algorithmic configurations” should not be mixed in the same output repository.)

Current state and next steps

- Current state is, roughly: “Looks like it should work, but can’t commit without making it more concrete”
- I believe this is not realistic to resolve with paper design work, so, my recommendation is:
- Finish writing up what we have already learned more comprehensively, but then very quickly proceed to:
- Prototype implementation ”hack week”
    - Schedule it? Perhaps May 30 – June 2 week?