

LQCD Workflow Mini-review

W. E. Brown, D. Holmgren, E. Neilsen, M. Paterno

March 16, 2009

1 Introduction

The major goal of the LQCD Workflow Project is to create a system for designing, maintaining, and running LQCD workflows, and provide for the storage and use of their results. After completing an evaluation of available products, one has been selected and a prototype LQCD workflow system has been developed.

The present report is the result of a requested assessment of this prototype system in order to guide its future development. In particular, the review team was asked:

1. How well does the design and prototype implementation fit the requirements?
2. How well does the current system match the configuration generation analysis?
3. What improvements should be made to the code (including design and data model)?
4. What areas are in need of further study, specification, or review?

1.1 Attendees

The review began with presentations from the principals involved in the preliminary evaluation and the subsequent design and implementation of the prototype system, aided by helpful comments from additional observers. The reviewers acknowledge, with thanks, the participation of the following presenters and observers:

- Abhishek Dubey (by phone)
- Jim Kowalkowski
- Paul Mackenzie
- Lucciano Piccoli (principal presenter)
- Jim Simone
- Amitoj Singh

1.2 Additional considerations

The following concerns had been singled out by the presenters/observers for special attention by the reviewers:

- the abstractions and concepts
- how well the package expresses the LQCD configuration generation workflow (and others)

- interfacing with legacy applications - Providing wrappers to applications for different workflow engines
- parameter set management
- the design of the package as a whole

1.3 Plan of this report

In section 2, we present an overview corresponding to our understanding of the LQCD Workflow Project. Section 3 comments on the Project's requirements as a whole, while section 4 details the reviewers' major concerns.

In section 5, we assess the progress of the prototype implementation vis-à-vis the Project's identified requirements. Section 6 makes recommendations regarding the Project's design, and section 7 focuses on recommendations regarding coding. Section 8 finishes the report with a brief conclusion.

1.4 Document conventions

Within this report, names of database tables and columns will be presented in this font, and code snippets (including names of types and objects) will be presented like `this`. Other names will be presented in *italicized* font. Finally, the reviewers' specific recommendations are introduced with the phrase "**We recommend . . .**"

2 Project overview

The goal of the LQCD Workflow Project is to provide an automated system for the execution of lattice QCD calculation campaigns. At present, the scientists doing these calculations rely on scripts that have evolved over many years. These scripts have been implemented in shell, Perl, and Python. They manage the tasks of submitting LQCD jobs to the batch systems of clusters and other supercomputers, monitoring the jobs, staging data products in and out of storage areas used by the jobs, logging progress, recording status, and various other bookkeeping tasks.

There are various types of campaigns, and there are sets of scripts unique to each type. The reviewers were given two specific campaign types as examples. These were "configuration generation" (*confgen*) and "two-point analysis."

Configuration generation campaigns generate ensembles of vacuum gauge configurations. These campaigns are very straightforward, with a single binary executed repeatedly, using the prior output data as input to the next iteration. A set of physics parameters describes a given campaign, including such variables as lattice size, lattice spacing, quark masses, and couplings. In the first phase of a campaign, called the tuning phase, a physics parameter (related to the average plaquette) is varied until consistent gauge configurations are generated. After the tuning phase is declared finished by a scientist who analyzes the stream of configurations, the tuning parameter is fixed and the campaign enters the phase where the actual ensemble of configurations is calculated and stored. As described to the reviewers, these campaigns can bifurcate into two

or more streams. At the bifurcation, parameters such as the random number set are changed to create a new calculation stream.

The second campaign example shown to the reviewers, in much less detail, uses an ensemble of gauge configurations to generate quark propagators via one or more binaries, and then performs calculations to determine correlations between these propagators. This style of campaign has many opportunities for parallel execution of the user binaries, as each gauge configuration is used independently to generate the various heavy and light quark propagators.

The reviewers were asked to concentrate on the as-yet unfinished implementation of a workflow system for the *confgen* campaigns.

The LQCD Workflow Project's members have investigated a number of workflow systems for suitability. These systems include *Kepler*, *Askalon*, *Swift*, and others that are documented in a report. One of the conclusions of their investigation was that the existing systems are inadequate to meet the requirements for an LQCD workflow system. A list of these requirements was provided to the reviewers. The Project has made a design choice to wrap an existing workflow system with front and back ends. The front and back ends will be responsible for parameter management, recording of provenance information for data products, tracking progress of the workflow campaigns, and organization and storage of secondary products.

The Project team has chosen Rails as the framework to implement the front and back ends. Although its goal is that any of a number of existing workflow systems can be used as the "wrapped" workflow engine, it has concentrated on *Swift* from the University of Chicago, and *Open-WFERu* (also known as *Ruote*, so termed in the rest of this document), an open source project developed using the Ruby language.

Ruote is a natural choice, as Rails is also implemented in Ruby. Rails provides a robust implementation of the "Active Record" design pattern. This design pattern is used to interact with relational databases. Relations (tables) are wrapped in classes, with attributes (columns) as data members of these classes. Object instances correspond to individual rows of the tables. The LQCD Workflow design uses classes in this pattern to represent individual physics parameters, sets of parameters, *participants* (atomic data or state transformation actions, such as an LQCD binary, or a simple shell script), data products, hardware descriptions (for example, cluster name and information), run history, and other relations necessary for workflow bookkeeping.

The wrapped workflow engine (currently *Ruote*) is responsible for executing the participants. A participant might be as simple as a shell command, with immediate return, or as complicated as the submission of a batch job or a mass storage access command, both of which would require logic to check for return status at later times.

Users must use the language of the wrapped workflow engine to express the logic of their LQCD workflows. A goal of the Project is to make these workflow specifications natural to the scientists; this implies that the vocabulary must include LQCD domain specific elements. We note that there are also many terms used that are specific to the domain of automated workflow, and that such nomenclature is generally unfamiliar to most scientists. (The reviewers were provided with a very useful document containing a glossary for the LQCD Workflow Project. This included general, workflow domain, and scientific domain terms. The glossary is available on the review web page.)

The configuration generation campaign example in this review was expressed in *Ruote*. However, the reviewers did not have the opportunity to look at this part of the *confgen* example, but

only at the front and back ends. The source tree provided to the reviewers also included a subdirectory with *Swift* code, and the reviewers were told that the prototype includes wrapping *Swift* as a workflow engine.

Two areas important to the successful execution of LQCD workflows, monitoring and fault tolerance, were outside of the scope of this review.

The Project makes use of the following abstractions:

- Parameter space, with values provided by the scientific users, grouped into sets:
 - physics parameters
 - algorithmic parameters
 - execution parameters
- Provenance space (*i.e.* history and dependencies of files, binaries, and other data products)
- Run history space (algorithm(s) used, cluster, used, nodes used, generated outputs, *etc.*)

LQCD workflows involve the following data file types:

- Vacuum gauge configurations, O(1Gbyte) each, ensembles O(1K) files each
- Intermediate analysis files: O(10Gbyte) each, approximately 3-5 per gauge configuration per analysis
- Final files: O(10Kbyte) each, O(1K) files per analysis

The presenters noted that the management of these final files was a substantive issue, and that it was hoped that the back end database would greatly help with organization and access to the parameters in these files.

3 Requirements

The reviewers were provided with a two page summary of requirements, numbered 1 through 13, that were from a summary section of a much longer document. Some of the requirements in this summary are vague or insufficiently specified. The full document may have sufficiently detailed requirement specifications, but this full document was not provided to the reviewers.

Now that a prototype is in place and lessons have been and are being learned, we feel that a rewrite of the requirements is necessary and **we recommend** this be done. The revised requirements should be more specific than the summary provided to the reviewers.

4 Major concerns

4.1 Testing

We found no evidence of systematic testing of the software, or even preparation for such. Systematic tests, both fully automated and documented manual tests, have important roles to play, and can do several things for a project. None of these can be done as well if the tests are developed or designed late in the development of the software.

Finding bugs The surface purpose of most tests is to determine whether the code performs as intended. While we assume that informal testing has been done as the code was being developed, systematic formal testing has several advantages. If a continuously expanded and maintained suite of automated tests is run regularly (perhaps daily), then errors introduced by new changes can be caught immediately. Documentation of manual tests serves as a reminder of what needs to be tested when more thorough testing is desired. If such testing relies on the memory of the developer when the testing is done, useful tests that were obviously important when the code was written may have been forgotten.

Specifying requirements One can think of a test of a feature of the code as a low level requirement on that code. Writing such tests is sometimes useful in helping developers thoroughly think through what the code actually needs to do, and helps maintainers and reviewers understand what the developers were trying to accomplish.

Demonstrating functionality A good test, whether it is fully automated or a documented manual test, can demonstrate to a reader what the code tested is supposed to do, and how it is intended to be used; a well written test is effectively an example.

Both Ruby and Rails have automated testing frameworks, and **we recommend** that the developer take advantage of these. In cases where an important feature is difficult to test using these frameworks, **we recommend** at least a few sentences describing a manual test be included in a test document.

When preparing for future review, **we recommend** that developers provide a short document (perhaps just a page or two) pointing out which tests are the most informative or demonstrative of functionality specified in the high level requirements document.

4.2 Checksum purpose and effectiveness

Many of the tables in the database store MD5 checksums. In many cases, the purpose of the MD5 checksum is unclear. In some cases, we speculate that the purpose is to serve as a key for the table. If this is true, there are several places where other columns or combinations of columns would serve as well, and the storage of the MD5 checksum is unnecessary.

In cases for which it would be useful as a key because there are no other columns or combinations of columns that would serve, the problem of hash collisions must be addressed. In some cases, the construction of the data to be hashed makes collisions very likely. For example, the model `ParameterSetParameter` creates a hash of the catenation of the set id and the parameter id. If the set id is 1 and the parameter id is 11, the data to be hashed will be 111. If the set id is 11 and the parameter id is 1, the data to be hashed will also be 111; these two rows will have the same MD5 checksum.

Even when the data to be hashed is guaranteed to be unique, the MD5 checksums are not. This will be much less common, but possible, and if the database is assuming the MD5 checksum column is a unique key, the results are disastrous; such collisions must be prevented.

If the MD5 checksum serves another purpose, and is not required to be unique by the database, then it may be (but is not necessarily) acceptable for there to be hash collisions. For example, the database may store the MD5 checksum of a file so that copies of it may be checked for corruption.

4.3 Create a design document

To move beyond the prototype, **we recommend** that the developers create and maintain a formal design document, with particular emphasis on the data model including motivations for each feature. Further, **we recommend** that this document be reviewed upon its completion.

Documentation generated from the source code files themselves, using tools such as those described at the end of section 7, can form the core of the low level elements of the design document. When writing comments in the code, **we recommend** the developers always keep in mind the documentation that will be derived from it.

We recommend that the design document also include a high level description of the architecture. The high level design should provide a basic introduction to the main elements of the design, and provide a context in which lower level design descriptions can be understood. After reading the high level design, a software developer previously unfamiliar with this Project should be able to navigate the low level design description to find the documentation for code that performs a specific task or database elements that hold specific data of interest. The developer should be able to figure out from the design document which requirements can be fulfilled by the current design, and which requirements each major architectural feature supports.

For this application, a thorough description of the database schema is essential. **We recommend** that the design document include not only an entity-relationship (ER) diagram but also, for every column of every table in the database, a prose description of what data that column is intended to contain and its purpose in the application. See section 7 for recommendations of tools for the creation and maintenance of such documentation from source code and comments contained therein.

4.4 Code consistency

The reviewed code base is not in a self-consistent state. **We recommend** that developers should keep the code in a self-consistent state. Periodic systematic testing can be an invaluable tool for reminding the developer of the different parts of the Project that must be updated for a major design change. Consistency is important not only in the code itself, but also the nomenclature. The same terms should be used for the same things consistently throughout both the code itself (in the naming of variables, classes, *etc.*) and the documentation.

The Rails framework provides a fine-grained modularity that supports a development method consisting of multiple (relatively) small steps. In section 7 we provide several coding recommendations that we believe will help take advantage of this feature of Rails, and which in turn will make keeping a consistent code base easier.

5 Progress towards Project requirements

This section lists by number and brief title each formal requirement for the LQCD Workflow Project, and gives the status of each requirement in the prototype. For the description of each of these requirements, see the summary document on the review web page (<http://home.fnal.gov/~piccoli/lqcd/>).

We note that several of the requirements (1.7 Assignment of Resources, 1.8 Stage In Configuration Files, 1.9 Fault Tolerance, 1.10 Manage Intermediate Files, 1.12 Campaign Execution Time, 1.13 Interact with Scheduler) are not addressed by the prototype. **We recommend** that the Project consider whether such requirements are still relevant, and if not, descope the Project accordingly.

1.1 Execute campaigns While it is clear that the current implementation is intended to be able to execute campaigns, it is unclear from either the code or any running example how this is to be done. How does a user start the execution of a workflow?

1.2 Campaign specification The current implementation has a mechanism for specifying workflows, but it is currently at too low a level for use by many users; there is a critical need for a higher level workflow specification mechanism. Such a mechanism could be implemented in either text or graphic mode. In the absence of such a higher level specification capability, there appear to be tools that could be of value to users for helping create *Ruote* workflows graphically, and displaying the corresponding low level text specification.

See http://difference.openwfe.org:4567/?pdef=onerror_0.rb for an example of the on-line tool in action.

There does not appear to be a way to organize a set of participants as a “named thing” with a version and/or additional tags, so that a scientist could easily specify a set of participants which are known to work together well.

1.3 Dispatch campaigns There appears to be some progress toward submitting batch campaigns, but significant work remains to be done.

1.4 Monitor progress There appears to be some progress toward supporting the monitoring of workflow progress, but significant work remains to be done. Is the Rails web application supposed to spawn long jobs and allow monitoring of them, or does it just prepare jobs to be spawned and monitored by some other application? The specification of this requirement needs to be more detailed.

1.5 Access execution history The current design appears to allow this, but work remains to be done. The specification of this requirement needs to be more detailed.

1.6 Handle multiple users The design appears to allow this without explicit support, but rather depends on capabilities of the underlying workflow engine and scheduling tools (*Ruote* and *Rufus*) that were not explored in this review.

The requirements for user management are underspecified, and little has been done to address possible implied requirements. The architecture does not seem to restrict use by multiple users - should an instance of a workflow be limited to a single user?

1.7 Assignment of resources This requirement does not appear to be addressed yet. Perhaps this is handled by *Rufus*, but this is unclear.

From the data model, it is possible for a participant instance to spawn jobs on multiple clusters. It would be sensible to limit participants to single execution entities.

1.8 Stage in configuration files This requirement does not appear to be addressed yet. Perhaps it can be implemented through a standardized participant?

1.9 Fault tolerance This requirement has not been addressed. Concern about restarts of not only participants but also the Rails web application is necessary at this stage. There are several possible approaches. A participant can update the database either directly, through a web application, directly, or using a “blackboard” that can be read by the web application later. In the later case, downtime in the database and web application need not cause status from running participants to be lost.

1.10 Manage intermediate files This requirement has not yet been addressed.

1.11 Data provenance Significant progress has been made on tracking data provenance. **We recommend** that hash codes, such as MD5 checksum, of all files that are tracked be included in the database.

Consider recording the provenance of the entries in the secondary products table. For example, the system might record the path of the log file from which information was extracted and enter it into a row of the secondary products table.

1.12 Campaign execution time This requirement has not yet been addressed.

1.13 Interact with scheduler This requirement has not been addressed, and perhaps the requirement itself is overly general and needs to be rewritten.

6 Design recommendations

6.1 Database schema

There are several places where the schema is either inconsistent, or uses column names that are confusing in light of Rails naming conventions. For example, the `plaquette_values` table has `config_file_id`, but there is no table named `config_files` (although there is a model `ConfigFile`). **We recommend** the schema (and also the object model) be brought into a consistent state before any additional functionality is added to the system.

We note that none of the database tables contains an index. **We recommend** reviewing all the tables to determine what indices should be formed. If query logs for the production database are available, **we recommend** reviewing them to help identify slow queries that would benefit from the addition of indices. In addition, some of the tables seem in need of uniqueness guarantees, which could be enforced by the database engine through the use of unique indices.

Most of the database schema is not specific to the *confgen* application, nor even to LQCD; it is applicable to any workflow. **We recommend** considering the factorization of the design to identify clearly the part that is generic and the part that is specific to the *confgen* application (e.g., the `ConfigFile` class).

6.2 Object model

6.2.1 ActiveRecord issues

In several places, object model features that could be provided by ActiveRecord seem to be implemented “by hand.” For example, the model `ParameterSetParameter` and the associated table `parameter_set_parameters` seem to provide a many-to-many relationship between `parameter_sets` and `parameters`. This model contains a hash code that seems not to provide any benefit. Except for the existence of this hash code, it seems that this model could be eliminated, the table replaced with a table named `parameter_sets_parameters` containing columns `parameter_set_id` and `parameter_id`, and the `has_and_belongs_to_many` relation used in the models to provide the association. The model `WorkflowInstanceParticipant` may be another example of a model that could be eliminated in favor of use of `has_and_belongs_to_many`

The model `ConfigParameterSet` is defined as inheriting from `ParameterSet`, and there is no corresponding table `config_parameter_sets`. The result is that *find_xxx* functions, when used on class `ConfigParameterSet`, will return objects that are not `ConfigParameterSets`. **We recommend** looking into use of the *Single Table Inheritance* pattern, as described in *Agile Web Development with Rails*.

6.2.2 Model classes

We recommend that the `Parameter` class be given a method that returns the contained value as the correct type, rather than the string representation.

Several of the model classes (e.g., `ParameterSet`, `PlaquetteSet`) contain hash codes that hash the value of one or two columns of the database. We see no value in such a hash, and **we recommend** reviewing the hash codes in all the models to verify their utility.

We don't understand the difference between `ProductProperty` and `Parameter`; the first seems to provide a strict subset of the functionality of the second. **We recommend** clarification of the purpose for each of these classes, and that consideration be given to collapsing the two into one class.

In the existing code, a product may be associated with the `ParameterSet` used in its creation (through the `ParameterSet`'s id). But it seems to us that it should be the *participant* that should be associated with the `ParameterSet`, and the product associated with the participant that created it. **We recommend** considering this modification.

6.3 Views

Some of the views (perhaps those that were generated using the code generation scripts in Rails) allow editing and destruction of records in the database. Since modifying or deleting any row which is referenced from elsewhere in the database could destroy the provenance tracking ability of the application, **we recommend** removing all such views.

7 Coding recommendations

We recommend the use of Rails features that allow views to be more automatically adaptive to changes in models. For example, many views are written in a form that explicitly makes use of the names of the data members of the associated model, rather than using the features of `ActiveRecord` that allow one to discover the data members dynamically (e.g., the `column_names` class method of every model class).

Rails is based on the *Model-View-Controller* design pattern; the separation of code into different realms of responsibility is a significant contributor to the ease of maintenance of a Rails application. Mixing responsibilities results in a loss of maintainability. In some of the views (e.g., `views/parameters/index.html.erb`), database queries are mixed in with the view code. **We recommend** that such queries be moved to the associated controller.

Some of the code uses hard-coded absolute paths, and is thus not portable. **We recommend** avoidance of absolute paths, and instead **we recommend** the use of Ruby and Rails idioms which rely on the standard Rails directory structure to work with paths relative to the local directory.

We recommend following the idiomatic Ruby naming scheme for predicate functions: *e.g.*, use `parameterSet?` rather than `isParameterSet`.

We note that the idiomatic Rails naming scheme for model classes is followed almost everywhere. **We recommend** that `PBSProvider` be brought into conformance, and renamed to `PbsProvider`, so that the automatic Rails table name association does not have trouble.

In some places classes are externally manipulated by their clients, rather than being given responsibility for performing work. For example, the `Parameter` object in the `addParameter` method of `ParameterSet` is told to calculate its hash. **We recommend** instead that `Parameter` objects be able to generate their hash as necessary, so that client code can merely ask for the hash, rather than having to first command its calculation and then request the value. In this specific method, it may be that the entire functionality implemented “externally” might instead be replaced by already-existing behavior in the class being manipulated: much of the code in this method could be replaced by just calling `parameter.save` (or perhaps `parameter.save!`).

`ActiveRecord` provides many utility functions for performing database queries, the use of which tends to simplify and clarify code; the current code takes insufficient advantage of these utility functions. For example, the method `addParameter` in `ParameterSet` uses

```
p = Parameter.find(:first,
                  :conditions => "hash_code='#{parameter.hash_code}'")
```

We recommend instead:

```
candidates = Parameter.find_all_by_hash_code(parameter.hash_code)
```

Note that we have also modified the code to deal with the possibility of non-uniqueness in the MD5 checksum, a problem discussed earlier in this document. As a second note on this same method, we believe that use of the `find_or_create_by_xxx` family of functions might dramatically simplify the code.

Ruby provides a documentation system (RDoc) that can create HTML documentation pages from comments in the source code, and that also allows for non-source based documentation to be included. **We recommend** that RDoc comments be included describing the purpose of each class, and of each public method. **We recommend** also the consideration of two additional Rails-related tools: the plug-in *annotate_models* (which queries the database schema to determine the database columns, and annotates each model class to reflect its associated table) and *railroad* (which can create entity-relationship diagrams showing the database schema). We note that the code must

be in a self-consistent state for *railroad* to work, and thus **we recommend** that this work be done before further development is done.

8 Conclusion

This is an excellent effort at a prototype. Significant work remains to be done, using the experience gained from the prototype to generate a more detailed set of requirements and a design document for a production implementation.