LSST Data Management Software Development Practices and Tools

Tim Jenness[a], Frossie Economou[a], Krzysztof Findeisen[b], Fabio Hernandez[c], Josh Hoblitt[a], K. Simon Krughoff[a], Kian-Tat Lim[d], Robert H. Lupton[e], Fritz Mueller[d], William O'Mullane[a], Russell Owen[b], Stephen R. Pietrowicz[f], Pim Schellart[e], Jonathan Sick[a], John Swinbank[b], Adam Thornton[a], James Bosch[e], Jeffrey Kantor[a], John K. Parejko[b], Paul Price[e], and Brian van Klaveren[d]

[a]LSST Project Office, 950 N. Cherry Avenue, Tucson, AZ 85719, USA
[b]University of Washington, Dept. of Astronomy, Box 351580, Seattle, WA 98195, USA
[c]CNRS, CC-IN2P3, 21 avenue Pierre de Coubertin, CS70202, 69627 Villeurbanne cedex, France
[d]SLAC National Accelerator Laboratory, 2575 Sand Hill Rd, Menlo Park, CA 94025, USA
[e]Department of Astrophysical Sciences, Princeton University, Princeton, NJ 08544, USA
[f]NCSA, University of Illinois at Urbana-Champaign, 1205 W. Clark St. Urbana, IL 61801, USA

ABSTRACT

The Large Synoptic Survey Telescope (LSST) is an 8.4m optical survey telescope being constructed on Cerro Pachón in Chile. The data management system being developed must be able to process the nightly alert data, 20,000 expected transient alerts per minute, in near real time, and construct annual data releases at the petabyte scale. The development team consists of more than 90 people working in six different sites across the US developing an integrated set of software to realize the LSST science goals. In this paper we discuss our agile software development methodology and our API and developer decision making process. We also discuss the software tools that we use for continuous integration and deployment.

Keywords: Agile Development, LSST, Distributed Development, Continuous Integration

1. INTRODUCTION

The Data Management System (DMS)[1] for the Large Synoptic Survey Telescope (LSST)[2] has been under development since at least 2004.[3] During that time a number of technologies have been adopted and our development practices have evolved as we transitioned from the research and development phase to construction. The Data Management (DM) team is distributed across the states (see affiliations above), with some external contributions from CC-IN2P3 in Lyon, France. As a uniform survey, LSST is highly dependent on its Data Management System, which, like the telescope and camera, has requirements that have historically been at or beyond the state of the art. From the beginning LSST was recognized as needing a substantial software effort, and significant portions of both the design and development and the construction budgets have been devoted to that effort. Hence LSST has been able to hire full-time software engineering staff, including developers and developer support staff. Founding and growing the LSST Data Management team has thus been like creating a start-up software company. Developers with experience in software engineering, not just scientific programming, were sought, several with experience in industry. Best practices and tools used by open-source companies were adopted. Evolution, not only of the code base but also of the development process itself, was embraced; developers were empowered to make changes. Given the distributed team, it is important that communication channels are open and easy to use and that our tools evolve as community standards evolve. Being agile enough to be able to migrate from one tool to another during the lifetime of a project is key when the software, processes, and people, change over what will be a 25 year period once the 10-year survey completes. For example, over the years we have migrated

Further author information: (Send correspondence to T.J.)
T.J.: E-mail: tjenness@lsst.org

the codebase from Subversion to git (§4.1.1); we have switched instant messaging from HipChat to Slack (§5.1); we have migrated continuous integration from Buildbot on our own hosts to Jenkins running in the cloud (§8.1); and we have extended our documentation standards from just Doxygen to include Sphinx (§6.2.3) as well. In the following sections we describe the current development practices for LSST DM.

2. AGILE DEVELOPMENT

LSST data management follows a long term plan with six month cycles. The development approach falls in the rather broad *Agile* methodology. It is also beholden to organizations such as NSF which require a more traditional approach to project development such as the Earned Value Management System. We have previously presented this from both the European Space Agency and LSST perspective,[4] and in 2016 we presented a more complete approach for LSST to the problem of Agile in the earned value world.[5] Here we provide just a brief update on that paper concentrating more on the Agile aspects.

2.1 Management

The Project Management Guide[6] provides comprehensive details on the mechanisms underpinning LSST Data Management's approach to project management; the reader is referred to that document for details as required. LSST Work Breakdown Structures (WBS) use a hierarchical numbering scheme with the first component always *1*, referring to the LSST Construction Project. The second component, *02C*, corresponds to Data Management Construction. Subdivisions thereof are indicated by further digits corresponding to teams (each led by an institution) within the DM project. These subdivisions are referred to as the third level WBS, and each institution has a Technical/Control Account Manager (T/CAM) responsible for planning, estimating, monitoring and Earned Value Management reporting for those WBS elements. The detailed management plan is defined in LDM-294.[7]

2.2 Basic Assumptions

The Project assumes that a full-time individual works for a total of 1,800 hours per year: this figure is *after* all vacations, sick leave, etc., are taken into account. Staff appointed to "developer" positions are expected to devote this effort directly to LSST. Appointment as a *scientist* includes a 20% personal research time allowance. That is, scientists are expected to devote 1,440 hours per year to LSST, and the remainder of their time to personal research.

Our base assumption is that 30% of an individual's LSST time (i.e., 540 hours/year for a developer, 432 hours/year for a scientist) are devoted to overhead for regular meetings*, ad-hoc discussions and other interruptions. This is similar to the standard *Agile* discount, however in the earned value world that must be accounted for, and it is considered Level of Effort.

These assumptions seem to hold well for DM planning and reporting.

2.3 Long Term Planning

The plan for the duration of construction is embodied in:

1. A series of *planning packages*, which describe major pieces of technical work. Planning packages are associated with concrete, albeit high-level, deliverables (in the shape of milestones), and have specific resource loads (staff assignments), start dates, and durations. The entire DM system is covered by around 100 of these planning packages.

2. *Milestones* represent the delivery or availability of specific functionality. Each planning package culminates in a milestone, and may contain other milestones describing intermediate results.

* "Meetings" include, for example, scheduled weekly team meetings, stand-ups, etc.; major conferences or project meetings involving preparation, travel time, etc., should be scheduled in advance and allocated Story Points

Planning packages are defined at the fourth level of the WBS. All WBS elements are related to the set of Data Management Products embodied in the System Design.[8] Each product has a *product owner* to guide the agile development, this is frequently one of the Data Management Scientists. During the planning of each 6 month cycle, effort is drawn from the budget allocated to the planning packages to generate the cycle plan, described in terms of epics in Jira. Each epic is associated with a single WBS element with story points accumulating against that budget. This budget is subtracted from that available in the planning package at the point when the epic is defined. Team members then add specific stories to these epics. The Jira system is synchronized with the Primavera project management tool every three months using in-house tools.

In order for the DM system to deliver the science potential of LSST, new algorithmic or engineering approaches must sometimes be researched. It is appropriate to budget time for this research work in planning packages and they result in epics of defined length (rather than specific deliverable). Areas where successful delivery of the DM system is dependent on speculative research are a source of risk: where possible, the plan also provides for a fallback option to be taken when research objectives are not achieved. This may also lead to an entry in the risk register.[5]

Some work is *emergent*: we can predict in advance that it will be necessary, but we cannot predict exactly what form it will take. The typical example of this is fixing bugs: we can reasonably assume that bugs will be discovered in the codebase and will need to be addressed, but we cannot predict in advance what those bugs will be. We included this in the schedule by defining a *bucket* epic, with a small allocation of budget, in which stories can be created when necessary during the course of a cycle.

2.4 Sprints

The broad plan (to 2022) is laid out in planning packages. For a given cycle (six months) more detail is put in the form of epics which are shared across the teams in a face-to-face meeting a little ahead of the start of cycle. The team then start to populate the epics with stories (both of which are Jira tickets) which are at an appropriate level for day to day planning. Within the cycle we follow monthly sprints with the usual agile steps: **Preparation:** In a local team meeting, stories for each team member to work on are chosen from the epics and verified to fit within the available points. **Execution:** As the cycle progresses, virtual or physical standup meetings are held several times per week to catch blockers and clarify stories. At the higher level there are two T/CAM standups each week on Tuesday and Friday to raise issues across team boundaries. **Review:** At cycle end a retrospective should be conducted, although these are not yet formally part of our process.

We use Jira's Agile capabilities to manage our sprints. Each technical manager is responsible for defining and maintaining their own agile board. The board may be configured for either Scrum or Kanban style work as appropriate.

2.5 Reporting

DM produces a mandatory written monthly report for NSF/DOE in which each team reports on progress and issues. A new addition to this report since last year is to clearly track the state of milestones which have been met or delayed. The Primavera[5] system outputs standard project metrics on variance and budget which are reasonable for reporting hardware based activities, but it does not track Agile progress well unless there are well defined deliverables. In 2017 we laid out a set of test driven milestones to show progress in Data Management.[9] These milestones are completed by execution of tests laid out in a test specification tying them to requirements, resulting in a test report.

3. MODEL-BASED SYSTEMS ENGINEERING

The overall LSST project uses Model-Based Systems Engineering,[10] with designs, requirements, and behavior described using the SysML language and stored within a database-backed tool, initially Enterprise Architect and now MagicDraw. MagicDraw permits communication between teams through a unified model of the system and relatively easy maintenance of the requirements at all levels of the system including traceability between requirements and the system components that satisfy them. The SysML model is also useful for verification of the system; test cases can be derived from the requirements and behavior descriptions in the model.

3.1 Requirements

Requirements are written using a combination of a formal specification, which uses verbs such as "shall" and "will" in a stylized manner, along with a description that explains the context and interpretation of the requirement. Constraints are added to the requirement that document values of parameters used within the specification, along with their descriptions and units. Each requirement is given a unique identifier generated based on its containing document and a monotonically-increasing sequence number. This requirement id allows requirements to be safely referred to and linked together even if the containing document is reorganized or if a requirement is deleted or replaced by another. Requirements document organization into sections is handled by grouping requirements into SysML packages. Numbers prefixed to the titles of the packages and the requirements within them allow them to be sorted into a human-meaningful order. These numbers are sequential within a containing package, not hierarchical, making renumbering and reorganization easier. Requirements documents are generated using one of two custom macros, one for Microsoft Word documents and another for LaTeX documents. The document generation macros sort the elements within packages and strip off the sequential prefix numbers, allowing the word processing tools to create the hierarchical section and requirement title numbers. When LaTeX is used, which is the case for all DM requirements documents, the generated files are stored in git repositories and then follow the standard DM documentation release process.

3.2 Requirements Documents

Five levels of requirements documents are applicable to Data Management. At the highest level, the LSST Science Requirements Document (SRD)[11] sets out the overall scientific goals of the project in prose. The LSST System Requirements[12] are the project's formal response to the SRD, setting minimum, design, and "stretch" goals for requirement parameters. The Observatory System Specifications[13] documents requirements and budgets based on a high-level design, in particular partitioning the LSST system into Telescope & Site, Camera, Data Management, and Education & Public Outreach subsystems. At this level, the Data Products Definition Document[14] provides a full specification of the data products to be delivered by the LSST system. The Data Management System Requirements (DMSR)[15] are the flowdown to DM specifically. At the same level as the DMSR, Interface Control Documents (ICDs) specify the relationships between the different subsystems. Finally, within the DM system, component requirements documents are used where appropriate to constrain designs and provide for internal verification.

One excellent feature of MagicDraw is the ability to rapidly document the flowdown of requirements. Using a table with higher-level requirements on one side and lower-level requirements on the other, the "refines" relationship between the latter and the former can be created by merely toggling an arrow within appropriate table cells. Unfortunately, it is more difficult to use the SysML "copy" relationship in cases where the lower-level requirement happens to be identical to the higher-level one. It has been necessary to manually copy the requirement text (and generate a new requirement id, of course). Since system components are also present in the MagicDraw model, the tool also helps with recording which requirements apply to each component, and, conversely, which components are needed to satisfy each requirement.

3.3 Conclusion

Ultimately, SysML and MagicDraw have been most useful for systems engineering purposes at the LSST system level, rather than within the Data Management system itself. While the tool is fully capable of recording component designs in diagrammatic language, adoption has been somewhat difficult; developers tend to prefer to work directly on code. In its place, more limited diagrams are produced using other tools such as OmniGraffle, Gliffy (particularly as a plugin for Confluence), LaTeX, Archi, and websequencediagrams.com.

4. SOFTWARE DEVELOPMENT

4.1 Development Model

To enable concurrent development across all the LSST DM sites, the development team uses a decentralized development model based on the Git version control system[†].

[†] https://git-scm.com/

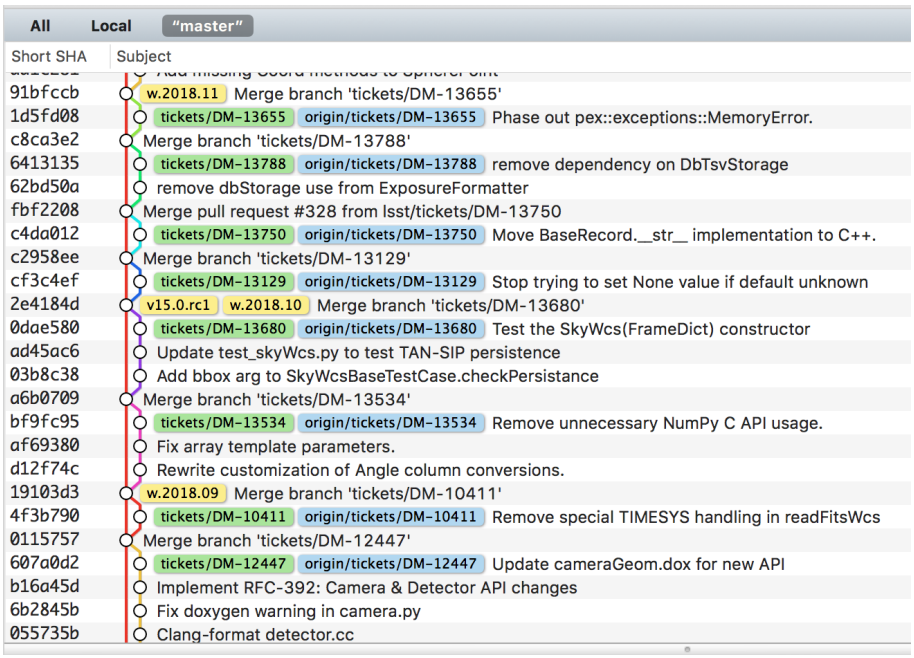

Figure 1. A subset of the commit history from an LSST package showing how each feature branch has been rebased before merging to `master`, resulting in an empty merge commit. Note also tags for the weekly releases and a recent release candidate.

4.1.1 Version Control

With a distributed team working across multiple time zones, we decided to move the source code from Subversion to Git in 2011. One key advantage of Git is the ability to work locally and make detailed commits without requiring that the remote server is always running. It was decided that the best fit for our monolithic Subversion repository was to convert it to many individual Git repositories, as is common practice. Initially we used gitolite for hosting git repositories on our own servers but in late 2014 we decided that we should move all development to GitHub;[16] the move was completed in 2015. DM was already a strong advocate of open development within LSST and this change made it considerably easier for the wider community to see what we were doing. Use of GitHub has made it easier to leverage tools such as Travis CI (§8.2), improved our code review abilities, and made it easier to link our pull requests to issues in other people's repositories.

4.1.2 Code Organization

The data management system is divided into several hundred individual Git repositories, hosted on GitHub[‡]. Each repository corresponds to one code package, and all but a handful of repositories usually have at most one developer working on them at a time. Isolating code in small repositories minimizes the need for developers to frequently download code updates, something the Git framework requires whenever developers make concurrent changes to the `master` branch of the same repository. Representing each package by an independent repository also makes it easy to formalize and track inter-package dependencies using tools like EUPS (§4.3).

4.1.3 The LSST Workflow

LSST software development is based on the shared repository model: work is done on branches of a single repository rather than on user forks. Development work is associated with a Jira ticket (§2.4) in a feature branch named after the ticket. Keeping all work in a shared repository makes it easier to link code changes to their corresponding Jira tickets. This in turn makes it easier to audit work in terms of Jira tickets. Developers must test their branch as part of the integrated data management system (§8.1) before the code is considered ready

[‡] `http://github.com/lsst/` and `https://github.com/lsst-dm/`

to review or merge. GitHub pull requests from the ticket branch to `master` are used for code review but not to merge the code. Instead, feature branches are merged by first rebasing them onto the latest version of `master`, then forcing a merge without Git's fast-forward optimization (Fig. 1). This creates an empty merge commit that serves as a marker to distinguish which commits came from which feature branch. Rebasing ensures the Git commit history appears linear to later inspection, and that all ticketed work is present on the corresponding branch. If we did not rebase before merging, merge commits could include significant cleanup work that would be difficult to resolve into their original tickets.

The LSST workflow has changed in two significant ways since it was first introduced. Originally, developers were allowed to commit fixes to Python docstrings and C++ documentation comments directly to `master`, without the need to create a ticket branch. However, as we expanded our tests to enforce coding style guidelines (§4.2.1), documentation edits began triggering build failures. In the current workflow all changes to code files, including documentation-only changes, must be made on branches and go through the review and testing process.

In addition, the workflow was at first enforced by individual developers and their reviewers. While it worked well most of the time, there were occasional cases where a developer would accidentally push commits directly to `master`, merge improperly, or rebase onto an out-of-date `master`. Therefore, we began using GitHub's branch protection feature to block both direct commits to `master` and merges of unrebased ticket branches. Branch protection is transparent to developers so long as they follow the workflow, so this improvement catches errors without altering the overall development process.

4.2 Code Quality

LSST DM is responsible for writing and supporting a large code base. As of April 2018, the Qserv database software[17] is about 100,000 lines of C++, Firefly has about the same amount of JavaScript,[18] and the Science Pipelines software[19] is approximately 290,000 lines of Python and 225,000 lines of C++.[§]

Code quality is hard to define and notoriously difficult to tackle when hundreds of thousands of lines of code are to be supported. In DM we have a coding standard[20] enforced with tooling where possible. We ensure that warnings from static analyzers are dealt with and where possible have the build fail e.g., if a `flake8` warning is issued; enforce a specific C++ standard (C++14) by configuring the C++ compiler to fail if it is violated; run unit tests for every commit, which also generates code coverage and test statistics; issue verbose warnings, including Python's `DeprecationWarning`, to warn us of any possible future issues; and run regular integration tests while tracking metrics of their data products and performance to ensure that we are improving with time.

4.2.1 Coding Standards

Over the years, the DM team have defined extensive coding standards[20] covering C++ and Python. This includes variable naming conventions, indenting policies and also how to document code. We use `numpydoc`[21] for Python, Doxygen[22] for C++, and JSDoc[23] for JavaScript, and we automate API documentation generation.

In 2017 we significantly simplified our Python standard by defining it relative to the PEP 8 standard.[24] We could not fully adopt PEP 8 since we had to balance the benefits of automated code linting versus the cost of changing tens of thousands of lines of code. In particular, the variable and method naming standard had been to use camel case everywhere, and it was felt to be too disruptive to change all the code, especially since to minimize confusion we would have to also modify the C++ code that is called from Python. There are two other areas where we are not compliant with PEP 8. Firstly, we disagree with the whitespace rules concerning binary operators and therefore disable this test. Secondly, we feel that an 80 character line limit is too short and we continue to use a 110 character limit for code. We did experiment with switching to 80 characters for one package but did not like the result. For documentation strings and comments we have adopted a 79 character limit as a compromise between the 72 characters specified in PEP 8 and our standard code limit. We have written, and had accepted, a patch for the `pycodestyle` tool to implement a check to allow us to have different limits for code and documentation. The `flake8` tool is used to validate code compliance, and we are happy with the results.

[§]Line counts include comments but not blank lines. Python interfaces are implemented using `pybind11` and that is counted as C++ code. Science pipelines software is defined as the `lsst_distrib` metapackage and does not include code from third party packages.

Once implemented, code reviews no longer waste time worrying about whether a space should be in a particular place and can now focus on the code functionality.

Based on the positive `flake8` experience on the Python code base, we investigated options for C++. Our code layout and formatting rules were updated to be based on the Google C++ Style Guide[25] with a few minor modifications, such as 4 space indentation and a 110 character line limit to maintain consistency with our Python coding standards. This allowed us to easily enforce these rules using the `clang-format`[26] tool with minimal additional configuration, though developers are also permitted to format their code manually or with any tool consistent with our standards. We are also investigating C++ linting with `clang-tidy`.[27] Although higher runtime and configuration cost at this time make it unlikely to be used for automatic linting at pull-request time, it is still a valuable tool in our workflow. Because it is based on the Clang C++ compiler `clang-tidy` has access to all the requisite information needed to find common causes for bugs using static analysis, identify things that are hard to find by eye (such as whether `0` represents a pointer value to be replaced by `nullptr` and whether a member function overrides a virtual function from a base class and should be marked with `override`), and automatically update a whole codebase to use features from modern C++.

4.2.2 Language Standards

We do not require a specific C++ compiler to build the DM software, but we do require compliance with the ISO C++14 standard.[28] We configure the Clang and GCC compilers to fail if the code is non-compliant. There is still code that uses C++98 but we are slowly updating the code as we encounter constructs that can be improved or simplified. We are monitoring the support of the C++17 standard in common compilers.

Given the commissioning timeline for LSST and the support schedule for Python 2.7, it is clear that Python 3 should be our ultimate target. We completed the migration to support both Python 3 and 2.7 during 2017.[29] In April 2018 we dropped support for Python 2 and have now standardized on Python 3.6.[30] This will allow us to use all Python 3 features for the first time.

4.2.3 Unit Testing

Unit testing is critically important when writing code. In DM we do almost all our testing from Python using the `unittest` module and we test the C++ code by using the Python bindings. Tests are run using `pytest`,[31] including several useful features provided via plugins. We run the tests in a multi-process mode, based on the number of cores on the system, and all code is tested with `flake8` for compliance. The output from the tests, including failure information, test count, and execution time, are written in JUnit XML format. This output is parsed by the Jenkins system to provide a test report to the relevant developer. We are also investigating the code coverage plugin and intend to record metrics of code coverage and test execution time as a function of time for long term trending analysis. Tests are run with full Python warnings enabled, including `DeprecationWarning`, and we expect these warnings to be fixed. Newer versions of `numpy` began warning when `NaN`s are used in numerical calculations. We have had to disable those warnings on a case-by-case basis once we confirm that `NaN` is a reasonable value for that particular code path; we decided not to globally disable all the numeric warnings.

4.2.4 Algorithmic Performance Tracking

At its most basic, the LSST DM team has to deliver algorithms that can calculate results for photometry and astrometry with sufficient accuracy to meet the overall science requirements of the LSST.[11] Additionally, we have fixed timing and computational performance requirements, and our algorithms must fit within the expected compute resources. It is important for us to know quickly if an algorithm suddenly gets worse during code development. We have developed a set of Key Performance Metrics that we calculate on a weekly basis using precursor data from instruments on CFHT and Subaru. These metrics are logged with the SQuaSH system[32] to enable scientists to examine trends and look for anomalies and to allow developers to test their development branches to ensure that there aren't unexpected regressions in algorithmic performance. We have found that tracking metrics is not quite enough because of the complex interplay between datasets and algorithms. Care must be taken to evaluate the metrics with knowledge of the context, rather than relying on simple thresholds. When unexpected changes in metrics are discovered, domain experts and DM scientists work together to understand why the change occurred.

4.3 EUPS

Managing the dependencies of many separate packages all undergoing rapid change, including changing APIs, is a hard problem. At LSST we use the Extended Unix Product System (EUPS).[33] EUPS was initially developed at FNAL and adopted by SDSS as UPS. It has evolved over time migrating from C to Perl to Python. The EUPS philosophy is to organize the hierarchy of components that are required to make a release of a complex software product on a particular platform (e.g., macOS); optional and required dependencies are declared recursively using "table" files. As products are built they are installed into a directory tree where each built product is isolated from other products and associated with the exact set of dependencies used to build it. All built versions of a product are available at any time, along with the correct set of dependencies stored as "expanded" table files. When a user requests that a particular product be "set up", EUPS sets environment variables in the current shell identifying the location of the appropriate version of every dependency. Each product can specify which additional environment variables are relevant (for example `LD_LIBRARY_PATH`, `PATH`, `PYTHONPATH`) in the table file. For each product that is being set up, EUPS adjusts the environment and sets, prepends, or appends to any previous value as requested. When a product is "un-set up" this procedure is reversed and the environment cleaned up.

A common use case is to set up a base environment and override versions of specific packages in order to debug some regression or to develop a new feature. EUPS excels at this use case since it is optimized for mixing and matching collections of products that have differing dependencies and ensuring that exactly the correct set are being tested; for example a user may use a specific version of a high-level product, with its exact dependencies, while overriding one or more dependencies with versions checked out from, for example, `git`. EUPS supports the concept of tagging a *current* version, and many users are able to simply use this version; other options are to accept a *stable* or other well-known tag. Unfortunately LSST has not done a good job of educating our users in best practices, and some find the EUPS user interface to be far too complex and opaque and indeed harder to understand than setting up a Python virtual environment and installing the relevant packages.

The reliance on environment variables has caused some issues on modern macOS installations because of System Integrity Protection (SIP).[34] SIP strips `DYLD_LIBRARY_PATH` from any subprocesses that are launched from a protected binary. Protected binaries include the standard shells, so shell scripts we run have no idea where to find LSST shared libraries. To overcome this problem we define a special environment variable, `LSST_LIBRARY_PATH`, that includes everything that would be set in `DYLD_LIBRARY_PATH` but will not be stripped by SIP and can therefore be available to shell scripts and build systems. This technique is not ideal but does allow us to continue to use macOS. There is a worry that Apple are deprecating `DYLD_LIBRARY_PATH`, and for that reason we are considering the use of link farms mapping the dependencies into a single temporary directory and using `@rpath` to locate the libraries. This is messy given that the file system will have to be modified every time a product is set up, and it is not clear how to remove the soft links when a product is no longer needed.

In addition to managing the environment and tracking of installed packages, EUPS also supports cloning a set of packages from one machine to another, distributing either source or binary packages from an EUPS server. EUPS is agnostic about the mechanism used to clone packages, supporting an extendable set of methods (e.g., `curl` or `scp` of tarballs). To support source distributions using a variety of build systems, the bundled `eupspkg` scripts provide a general system for building packages, supporting customization through a shell script that can be included with each package.

4.4 Building the software

With more than one hundred distinct git repositories, building a coherent version of the LSST Science Pipelines software is not as simple as it would be with a single large repository containing all the code. We use EUPS to specify the dependencies between packages; the `ups` directory within each package contains those dependencies and a description of how the code should be accessed from other packages.

4.4.1 Software Packaging

There are four types of software that are required to be handled in order to install the LSST DM Science Pipelines software packages. The first type are the prerequisites that we need to have available on the system: this can include compilers and build support tools such as `cmake`. The second type is Python and the set of

standard Python packages that we require. Theoretically the software should build with any valid Python, but for LSST development and testing we always install our own Anaconda, including prerequisite packages `numpy`,[35] `matplotlib`,[36] and `astropy`.[37] The third type of packages are EUPS "tar and patch" packages of third-party code which consist of a (compressed) tar file of the upstream distribution, an optional directory of patches to be applied after the source code has been extracted, and a `ups` directory containing a table file for dependency management along with a file named `eupspkg.cfg.sh` that provides instructions for overriding the default build and install behavior. This approach to third party packages allows us to know what patches we have applied and has the benefit of giving us direct control over a specific version that we wish to build against. The fourth type are "stub" packages whose job is solely to validate that a prerequisite Python package is installed and has a supported version. This also allows our own packages to explicitly require these packages be available for dependency tracking.

In a few cases, where the third-party software is available on GitHub and we are closely affiliated with upstream, we fork the code repository and use a branch named `lsst-dev` for local modifications such as adding EUPS support. For Science Pipelines we use this technique for the AST library[38] for handling coordinate transformations. The LSST EUPS-based software packages themselves use a standard layout with SCons[39] being used to build each package. We have written extensions to SCons to support our use of C++ with Pybind11 wrappers and Python code tested with pytest. Some support packages from the Science Quality and Reliability Engineering group do not use EUPS; instead, they use industry standard packaging systems based on the language in the package, with Python packages being distributed through PyPI and dependencies managed in the standard Python manner.

4.4.2 Building from git

With all the prerequisites installed we needed a way to retrieve all the source code from GitHub and to work out which packages should be built and in what order. We have implemented this by writing support packages that use EUPS for dependency management and `eupspkg` for builds. The `lsstsw` package was originally written purely to support continuous integration services but is now the default choice for developers who wish to build the most up to date version of the software from source. There is a shell script to install the local build system by installing Python and EUPS, and there is a Python program, `rebuild`, for doing a full build. This `rebuild` program within `lsstsw` is a wrapper around the `lsst_build` package. There are two key phases to doing a build. In the first phase all the source is cloned from GitHub and the correct git reference is checked out based on the user request (usually you list the feature branch name that you are testing). We have a master list in a GitHub repository that maps product name to git repository, and the command can be made to refresh this list every time it runs. Once the code has been cloned and the source repositories configured properly, the second phase is to build the code in dependency order and install it using EUPS tagged with a unique build number. The build tag can then be used to setup the products from that specific build.

4.4.3 Building from a source distribution

Not everyone needs to be able to build the cutting-edge version of the code. For those users we publish packages to an EUPS server and EUPS can download the packages associated with a public EUPS tag. This is usually a weekly release, for example `w_2018_13`, or an official release, such as `v15_0`. These tags are distinct from git tags, the corresponding version of which would be, say, `w.2018.13` in each repository (see Fig 1).

4.4.4 Future Considerations

We have considered creating a git repository similar to the current `lsst_distrib` metapackage that would contain all of its dependencies as git submodules. This is how we implement a single git repository of all the DM documentation. This would have the advantage of being able to check out specific versions of all the desired packages. Whilst the updates to the metapackage could be automated, this comes at the added complication of submodules. We are also investigating the (re-)adoption of true semantic versioning[40] for all our modules and conversion of the packaging to standard Python packages that can be `pip`-installable (see for example the discussion in Ref. 41), but this is not yet past a proof of concept stage.

5. COMMUNICATION

5.1 Slack

The data management group uses instant messaging tools to answer questions, share code snippets and errors, and coordinate activities in real-time. Before using instant messaging, we would communicate primarily via e-mail or over the phone. While these tools do still have their place, being able to have conversations in a team-wide application has been very helpful. Site channels help coordinate discussions useful to local teams. We have non-work related channels where we can discuss various topics, which has sparked conversations about previously unrealized common interests. It has been beneficial to team building and helped foster friendships.

We used HipChat for two years and eventually started looking at other options because we found the user experience was, at times, inconsistent and frustrating – especially the treatment of message history. After a brief testing period, we transitioned to Slack. The move from HipChat to Slack was straightforward, although we did lose online access to the discussion history from HipChat. At first, we thought this would be an issue, but in practice, we rarely search history older than a couple of weeks. We have had difficulties in balancing when to archive these discussions for the community. We've also had some problems making sure that decisions made within a Slack conversation were distributed to more team members, since everyone may not have been in the channel at that time. We continue to improve our efforts in this regard by starting RFC's (§7.1) or Community discussions (§5.2) to distribute information more widely.

Slack initially didn't have video chat when we switched from HipChat, but it does now. We use this for ad-hoc calls for the channel, which has been easier than coordinating a call via Google Hangouts or Blue Jeans.

Direct messaging can be set up for small groups, instead of creating a public chat room for an ephemeral discussion topic. We create temporary channels for members to communicate during workshops and conferences, which helps team members coordinate while on-site and to involve people who aren't attending. This paper itself was even organized within a Slack channel, allowing us to discuss it and to get automatic notifications of changes to it via the GitHub Slack bot. Other features include previews of URLs, global and per channel notification preferences, and an API to integrate third-party and custom chat-bot apps. One of those apps, sqrbot, is discussed in the next section.

5.1.1 ChatOps: SQuaRE Bot

DM-SQuaRE is using a Slack chatbot called `sqrbot` to make some tasks easier. Currently it performs a range of functions, from returning the status of various infrastructure machines to creating technotes to monitoring whether metrics in our processing stack have changed between CI builds; it also listens for mentions of Jira tickets and, when it hears one, posts a link to that ticket.

All of these tasks are frequently requested actions that formerly required breaking workflow. Either (as with technote creation) the requestor would have to interrupt someone else in order to get the work performed, or they'd have to interrupt their own workflow to, for instance, copy the ticket name, go to Jira, and search for the ticket. With `sqrbot`, the requestor can simply ask `sqrbot` in a Slack channel for the information (or creation of a technote skeleton, etc.) and get immediate gratification without needing to task-switch or interrupt a co-worker.

The basic architecture is simple: `sqrbot` is a collection of hubot scripts running as a Slack bot, which in turn drive microservices, written in Python and implemented using the Flask framework. These microservices have an API that responds with JSON, so the job of `sqrbot` is simply to accept commands, create appropriate HTTP transactions, and then reformat the output into a conversational format. The whole assembly runs in a Kubernetes cluster.

5.2 Community forum

Data Management operates the public LSST Community forum[¶]. Slack carries the bulk of our team communication because it is fast and team-focused, while the Community forum encourages longer, more in-depth posts that lead to slower-paced conversations. These slower-paced conversations tend to be more accessible for those who cannot work synchronously with the DM team, e.g., because of factors like time zones. Much of DM's own

[¶] https://community.lsst.org

use of the forum, therefore, tends to be intentionally public facing e.g., small announcements of new features in the LSST software. We host a support category to crowd-source help for astronomers who are using LSST software in their own investigations. To reach the LSST user community who are more interested in planning for LSST's data releases than using our software, we host categories for science and data discussions.

To implement the forum, we chose the open source Discourse[‖] platform because of its modern architecture and vibrant open source community. Discourse is designed around a user trust system that largely eliminates the need for moderation (though the moderation tools are also excellent). On the whole, Discourse is a low-maintenance system for DM to deploy and operate. Prior to Discourse, we used mailing lists for a similar purpose. The problem with mailing lists, though, is that their archives are largely undiscoverable, and their conversations are harder to link to. We created tooling to forward conversations from the Community forum to the old mailing lists as we deprecated them. We continue to use that forwarding infrastructure, though Discourse's native email notification system is largely sufficient for newer users.

5.3 Confluence

We use Confluence as a complement to Jira, e.g., for meeting minutes and for initial drafts of some technical notes. The ability to embed Jira tickets into pages and for pages to have embedded Jira queries, along with the simple tasks system, makes Confluence an excellent choice for storing meeting minutes. We do assign tasks during meetings and track open tasks, but we have a policy to migrate tasks to Jira tickets if it looks like a task is going to involve significant work. It can then be scheduled as part of the normal planning process. We have found that another place where Confluence is useful is for collaborating on hack sessions. A simple table on a page showing the status of related Jira tickets and associated people can help to avoid duplicate work. Finally, Confluence can be used for collating information for a transient purpose, with the understanding that if the information is to be persisted it should be migrated to a Technical Note (§6.1.1).

6. DOCUMENTATION

6.1 Types of Documentation

Data Management creates a variety of documentation for different audiences, and with different purposes. Broadly, we break documentation efforts into two categories: project documentation and user documentation.

6.1.1 Project documentation

Documentation tree. DM creates project documentation that fits within LSST's overall Document Management Plan.[42] As is standard practice, we organize the collection of project documents in a tree.[7] LSST Project Management and Systems Engineering requirements documents form the tree's trunk, along with interface control documents that DM collaborates on with other LSST subsystems. These flow into Data Management requirements documents for each DM product, and then from each requirements document we derive test specification documents. Overall, the DM Validation and Test Plan[9] coordinates these test specifications. Finally, we write test reports on the execution of the test plans. We have processes for managing project documents and revisions at the project level and within DM; they are described in §7.2.

Technical notes. We recognized that DM team members need or wish to write documents as a means of communication within the project. These documents might be design proposals, summaries of data processing experiments, or descriptions of a new internal tool or service. This content does not fit within the formal document tree, nor does it require oversight from the DM CCB (§7.2). Technical notes fill this niche for DM. We seek to automate as much of the maintenance surrounding these documents as possible. By allowing an author to create a new document and publish it without formal approval from management or waiting on a human coordinator, we maximize the likelihood that DM team members will choose to share information through technical notes. To minimize the barrier to entry we have implemented a self-service system based on ChatOps that allows any Data Management staff member to create and publish a technical note (§5.1.1). These technical notes are edited on GitHub and published as websites, and, in the future, we plan to make these documents readily citeable with DOIs and listings in archives. Most technical notes are written as part of the standard

[‖] https://discourse.org

development process and are reviewed in the same manner as code. As a cultural benefit, we see technical notes as a way for junior members to gain exposure within the project, and beyond, by publishing a citeable article featuring their contributions.

Developer Guide. The DM team is large and distributed, yet we need to work coherently. An effective way to promote this is through strong documentation of internal processes and policies. These range from a description of how we communicate, to the development workflow, code style guides, and user guides for internal services. While we could conceivably publish this information within the official documentation tree (see above), the content of the Developer Guide doesn't fit into a narrative format. Instead, the content is made up of isolated, yet inter-linked, how-to pages and guides. Thus we chose to publish our Developer Guide as a website,[20] using the same systems that support our user documentation. DM team members are encouraged to contribute improvements to the Developer Guide through GitHub. We use GitHub pull requests to get feedback on contributions and provide approval for changes to controlled pages such as coding style guides.

6.1.2 User documentation

User documentation describes what is built, what has changed, and most importantly, how to operate it. This is a deliverable of the LSST construction project, on par with the software itself. At the time of this publication, most DM effort towards user documentation has gone into the LSST Science Pipelines product[43] and user guides for internal tooling. The LSST Science Platform[44] is the second major documentation project.

We begin a new user documentation project by studying the nature of the product user needs. Then we work towards an information architecture plan that maps out topics in the user documentation project. As described by 45, topics are self-contained (though highly interlinked) articles that conform to a type. The general categories of topic types are concept guides, tutorials, how-to guides, and references.[46] This exercise allows us not only to systematically map the content needed to document a project but also to understand the infrastructure needed to support each topic type (such as the tools needed to generate an API reference, or the tools needed to deliver and test a tutorial). The Science Pipelines Documentation Design[47] technical note is our living design for the LSST Science Pipelines user documentation.

6.2 How we build documentation

Besides creating documentation content, we invest in infrastructure that helps us more effectively create and serve that documentation. Overall, we use a "docs-like-code" approach.[48] This means, broadly, that we write documentation like we write code by using plain text source formats, Git and GitHub for version control and collaboration, and continuous integration servers for both validating the content and deploying it to web servers. By using the same development processes for both code and documentation, our developers need minimal additional training and documentation. In the particular case of software user documentation, the docs-like-code approach ensures that documentation is versioned with the software since both reside in the same Git repository. The docs-like-code approach is also convenient for building custom tooling to automate documentation processes and integrate with the code base.

6.2.1 Versioned documentation delivery

A critical component of our docs-like-code system is the documentation hosting service, which we call *LSST the Docs*[49**]. We designed *LSST the Docs* around three goals: reliable scalability, support for versioned documentation, and flexible integration with continuous delivery workflows. LSST documentation is statically rendered at build time, stored in the cloud (Amazon S3), and delivered via a content distribution network (Fastly). This architecture is arbitrarily scalable and avoids the need to maintain servers and applications to dynamically render HTML. The static documentation approach also lends itself to *versioned* documentation. *LSST the Docs* uses URL path prefixes to deliver documentation for different versions of the same project. We often deploy draft versions of documentation based on Git branches to support code reviews.

Finally, *LSST the Docs*'s modular architecture makes it easy to integrate into continuous integration pipelines. Generally all documentation projects hosted on *LSST the Docs* have some form of continuous integration server.

**In reference to the popular https://readthedocs.org

Our LaTeX projects and technical notes are built with Travis CI, triggered automatically by GitHub activity. Documentation for the larger, and more complex, LSST Stack software is built on a Jenkins CI server. In each case, the continuous integration pipeline uses a client to upload built documentation to *LSST the Docs*. *LSST the Docs* does not provide its own documentation build service. This is in distinct contrast to other static documentation hosting services, like Read the Docs, and allows the necessary flexibility to let software and documentation projects define their own build environments. In the next sections we describe the build systems for specific types of DM documentation that are deployed with *LSST the Docs*.

6.2.2 LaTeX tooling for project documentation

We write most project documentation with LaTeX. To standardize these documents, and ease maintenance, we use an lsst-texmf[50] LaTeX package (based on LaTeX classes developed for Gaia) that provides a document class, styles, and bibliography styles. The package also contains a common list of acronyms to populate document glossaries. Additionally, we package common BibTeX bibliography files in the lsst-texmf repository. Every DM LaTeX document is expected to upstream all of its citations into these common bibliography files. We also maintain a common repository of images, though in a separate Git repository. Our LaTeX documents typically incorporate these images as a Git submodule.

We write LaTeX documents on GitHub (each document in its own Git repository). When changes are pushed to a branch on GitHub, Travis CI builds the document. After building the document, the Travis CI job uploads the versioned document (corresponding to a Git branch or tag) to *LSST the Docs*. Since LaTeX documents compile to PDF, we create HTML landing pages that wrap the PDF and provide metadata to readers.[††]

Developing a document on GitHub is convenient for reviewing and approving change-controlled project documentation. When changes on the `master` branch are ready to become the baseline, we create a release branch to incorporate feedback from the Change Control Board. Once the board approves a document, we tag that version. LSST archives baselined documents, as well as intermediate drafts, in our Xerox Docushare repository.

6.2.3 Sphinx-based user documentation

For user guides, such as DM's Developer Guide and the LSST Science Pipelines documentation, we use Sphinx to generate static websites that we deploy to *LSST the Docs*. Sphinx is ideal because it integrates well with our Python codebase. For example, docstrings within our Python code are being converted to use the Numpydoc[21] format, which is incorporated in our codebase with the Numpydoc toolchain that is wrapped with extensions from the Astropy[37] project. Beyond Numpydoc we take further advantage of extensions developed by Sphinx's open source community. For example, the breathe extension pulls data about our C++ APIs from Doxygen, allowing us to generate C++ API references with Sphinx. We are also developing our own custom extensions, specific to LSST documentation. These are being developed in our Documenteer Python package.

For software projects, we incorporate the Sphinx site directly into the software's codebase. Since our software stacks are composed of many Git repositories, we maintain documentation in each repository, but combine that documentation at build-time into a single documentation website.

7. DECISION MAKING PROCESS

In 2014, anticipating the growth of the DM team with the start of construction, we adopted a new decision-making and change management process. Its goals were twofold: first, empower developers to make small, innocuous changes without any formal approval other than peer code review, as long as they are willing to be responsible for any fallout, and second, enable developers to easily propose larger changes, gaining approval rapidly when there are no objections. This has been particularly useful for changes related to APIs, code style, and the build environment.

[††] https://github.com/lsst-sqre/lander

7.1 RFC Process

A larger change is proposed through a "Request for Comments" (RFC) that is managed through a Jira workflow since DM developers are familiar with the tool. By default, the proposer of an RFC is expected to be responsible for handling any effects of the change. If they propose an API change, they are agreeing that they will fix any breakage in other DM code; if they are proposing a change to the style guide they will make the change to the style guide. When an RFC is submitted it enters the *Proposed* state, the proposal is mailed to the DM developer list, and an announcement is made on the main DM Slack channel. Each RFC has a defined due date by which a decision is to be made. For changes that are expected to have no real impact outside of a single package, a due date of 3 days is acceptable. For changes that might result in significant debate, for example adding a new package to be supported by DM or changing an API used by many packages, the proposer is advised to give at least a week and possibly two weeks for debate. Anyone can comment on the RFC, and when consensus is reached the RFC can be *Adopted* by the proposer. If consensus could not be reached the proposer has the option of withdrawing the RFC completely, adding more time, or flagging the RFC to the Change Control Board (§7.2) for more formal decision making. On adoption, an RFC must be associated with actual work by filing tickets in DM Jira and connecting them with an "is triggering" relationship. The DM Systems Engineer is tasked with looking at RFCs on a weekly basis to ensure that proposed RFCs are not languishing and that implemented RFCs are correctly marked as such. The *Implemented* state is recognized by scanning all the *Adopted* RFCs and seeing that all triggered work has been completed.

The RFC process has been extremely successful with 467 RFCs filed since 2014, with 321 implemented, 51 withdrawn, 5 flagged, and 90 adopted with work pending. We have found that some RFCs are filed asking for changes that the proposer feels are desirable but which they themselves are not going to be responsible for implementing. These are sometimes a very good idea, but, since there is no lead implementer, RFCs like this depend on T/CAMs picking up the work and scheduling it as part of their normal planning process.

7.2 Change Control Board

The LSST has a project level Change Control Board (CCB) for managing evolution of budgets, requirements and subsystem interfaces. The LSST CCB meets regularly with formal meetings each month and weekly video calls. The CCB process itself is mediated by a bespoke Drupal web app written early in the life of the project. DM use Jira for work directly resulting from a CCB investigation or for noting issues with project level change-controlled documentation that may need to be addressed when the person does not necessarily have a formal request prepared.

Inside DM we have our own formal DM Change Control Board chaired by the DM Systems Engineer and composed of the DM engineering and science leadership, including the Project Manager and Subsystem Scientist. DM CCB uses the same RFC process as described above, but issues to be discussed by the DM CCB are immediately submitted into the *Flagged* state. This is mainly used to discuss changes to the baseline documents and to approve test reports associated with formal milestones. It is also used to give formal DM approval to changes that DM would like to be made in project level documentation. When deemed necessary some RFCs may require a video conference meeting of the DM CCB to allow more direct discussion.

8. CONTINUOUS INTEGRATION

Continuous Integration (CI) and *Continuous Deployment* (CD) go hand in hand with Agile (§2) development and are critical in the modern astronomical observatory.[51]

8.1 Jenkins

The initial CI system used for pre-merge testing of science pipeline code in 2014 was Buildbot, partly because it is written in Python. After evaluating several potential CI/CD systems as a replacement, Jenkins was selected for numerous reasons. It has, e.g., the GitHub OAuth plugin, a healthy extension ecosystem, and an active core project.

Over time, usage of Jenkins has evolved from being used solely for pre-merge CI testing to automation of a number of common tasks. Among various sundry tasks, it is being used to build Docker images, update local

software mirrors, and schedule regular backups. Perhaps most notably, it is being used to drive a CD workflow for completely automated nightly and weekly release/publication of science pipelines codes (§9.1).

8.1.1 Configuration & Deployment

The Jenkins core and various plugins need to be version-managed and configured. Although there is currently a major effort underway to add native "configuration by code" to the Jenkins core, this is not yet considered production ready and did not exist at the time DM was transitioning away from Buildbot.

Puppet was selected as a configuration management (CM) tool as its `puppet-jenkins` module provided the most sophisticated Jenkins management abilities among the tools surveyed at that time. Non-trivial improvements have been contributed by DM staff to this module[52] in order to make it more suitable for managing a Jenkins deployment.

Configuration of Jenkins jobs is handled via the `job-dsl` plugin. This enables a `groovy` based domain-specific language (DSL) and a special job type that will "seed" Jenkins jobs from a git repository that contains `job-dsl` script(s). The seed job itself is maintained as XML that is installed by puppet. The result is that all jobs configuration is managed via source code management and no manual configuration via the Jenkins UI is required to add/delete/change or stand up a testing environment.

8.2 Travis CI

We use Jenkins to deal with our non-standard build system and unit testing of development branches, but the close integration of Travis CI with GitHub gives us the opportunity of running additional lightweight testing and deployment. As described in §6.2.1 Travis is used for deployment of our documentation artifacts. We also use Travis CI for linting: we validate that important CI configuration files have the correct formatting, and we run the `flake8` and `shellcheck` tools on our code. These Travis checks allow us to enforce our code merging policies (§4.1.3) by requiring that branches are up to date before merges are allowed.

9. CONTINUOUS DEPLOYMENT

9.1 Deploying Science Pipelines Releases

DM has three types of "release", each with a different cadence, for science pipeline related products: `daily` (also referred to as a `nightly`), `weekly`, and `cycle` or "major" release, which have historically occurred once per DM planning cycle (§2). A "major" release includes a human edited "changelog", updated installation instructions, usage notes, and a formal data processing characterization report, and thus cannot be completely automated. However, the mechanical steps of tagging, building, and publishing software do share Jenkins pipeline code with the automatically triggered `daily` and `weekly` releases. The `daily` and `weekly` are fully automated and triggered by Jenkins on a fixed schedule.

9.1.1 The "Weekly Release" pipeline

The weekly and nightly release pipelines (Fig. 2) are highly similar and principally differ in that a nightly release does not tag git repositories. The rationale is that a nightly is used both for developer convenience and as a *canary* to ensure that code changes have not broken the release workflow, but they are not a product that needs to be reproducible/rebuildable from source. Both release types share a library of common Jenkins pipeline methods and will be merged in the near future into a single pipeline script that uses a separate external configuration per release type.

"Touchstone" build & publish doxygen docs. The first step in producing a release is to make a build from the `master` branch of the source git repositories on the "reference" platform, which is presently `CentOS 7`. If the per product unit tests and a "quick" integration test succeed, a cross-package HTML formatted documentation set is built using `doxygen`[22] and is published to a service that developers may access with a web browser. A "manifest" listing the `eups` products that were part of the build, git repository SHA1s for those products, and build dependencies are recorded by `lsst_build` at this stage in a git repository structure which the tool refers to as `versiondb`. The canonical "versiondb" instance for science pipeline releases is published on GitHub at `lsst/versiondb`.

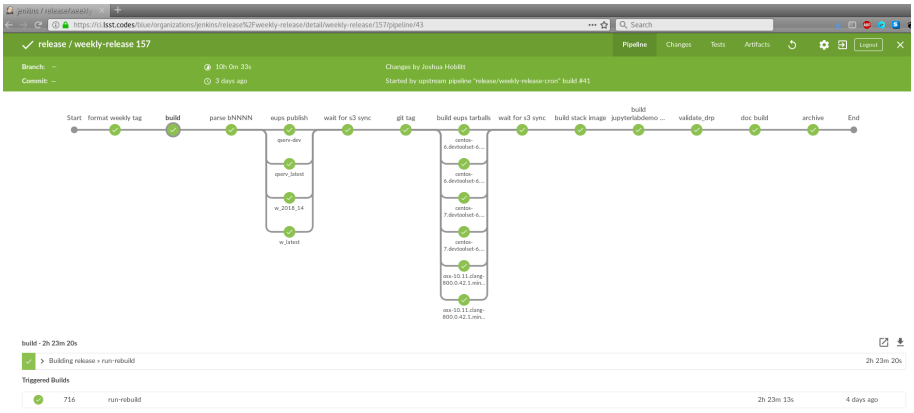

Figure 2. The Jenkins pipeline used for the weekly releases.

Publish eups source packages. `eups` (§4.3) "source packages", known as `eupspkg`, are produced from the installation tree resulting from the prior step. Note that although `eupspkg` are source-based, and do not contain binary objects, they require a complete binary installation as input to the packaging process. In addition to `eupspkg`, a `eups` "distrib tag" is also published, which is used to refer to a specific set of packages at fixed versions. Examples of `eups` tags are: `w_2018_14`, `d_2018_04_10`.

The `eupspkg` are pushed to an AWS S3 "bucket", which is used as the canonical repository for both source and binary `eups` packages. The S3 bucket is not publicly accessible as `eups` only supports remote package repositories over `http[s]` that have HTML "index pages" in the traditional format produced by the `apache` webserver. To accommodate this requirement, the S3 bucket is periodically replicated to a simple service that deploys on a Kubernetes cluster providing `https://eups.lsst.codes/stack/src/`.

Tag git repos. A script contained in the `sqre-codekit` package uses the initial build "manifest" to apply an annotated git tag to all repositories from which an `eups` product was pulled as part of the build along with a number of ancillary git repositories that contain build scripts, documentation, and other items that would be needed to exactly reproduce the release.

Build/publish eups binary "tarball" packages. This is perhaps the most complex stage in a release. Multiple builds/publishes are run in parallel. As of the `15.0` (`eups` distrib tag `v15_0`) release, the build matrix is: Python 3 on CentOS 6, CentOS 7, and macOS 10.12 (in 10.9 compatibility mode). We use the GCC 6 compiler from `devtoolset-6` on Linux and the Apple `clang` compiler matching the macOS version.

Every build is started from scratch and bootstraps a development/installation environment using the `newinstall.sh`[43] script, just as an "end user" would when building/installing science pipeline software.

Then a `eups distrib install` is performed, also as an end-user would run. This is followed by publishing `eups` "distrib tarballs" (binary packages) to a local path. `newinstall.sh` is again executed but into a new path. The second, pristine, installation is pointed at the local publication path, rather than the public `EUPS_PKGROOT`, and does a complete binary installation. This is followed by some simple acceptance tests, such as building another package from source against the binary installation. If there are no failures, the binary "tarballs" are then publicly published in a similar manner to the "source" `eupspkg` but using a different directory prefix to prevent source and binary packages from different environments from being intermixed.

Build docker image. A Docker image is constructed with the "tarballs" from the release pre-installed, which is published to Docker hub in the `lsstsqre/centos` repository.

The SCons-based build/installation (§4.4.1) that is used by all science pipelines software installs virtually all files from the build tree. This includes `.o` files, unit test scripts, test data, and `doxygen` HTML and XML output. In addition, none of the executable binaries have been `strip(1)`d. This is primarily to assist further development and debugging. These excess materials, which are unlikely to be of interest to an end user or in a production system, are stripped from the Docker layer resulting in an uncompressed image of less than 6 GB.

Build jupyterlab docker image The "release" docker image is used as the base (`FROM`) for the jupyterlabdemo base image (§9.3). This results in both "nightly" and "weekly" builds becoming available in a Jupyterlab environment with relatively low latency.

Process test data sets and collect metrics The "release" docker image is also used to run several test data sets using a package called `validate_drp`. Metrics are collected from the processing results and are shipped to a dashboard called "SQuaSH",[32] which plots trends over time.

Experimental documentation build & publish Due to the ease of "composition" with Jenkins pipelines, it is relatively low cost to add additional pipeline stages on a temporary basis in order to test out new features. Presently, this stage is "tacked on" to the regular release workflow in order to evaluate a new documentation build and publish process in parallel with the existing `doxygen`-based build.

9.2 Case Study: Data Access Services

The Data Access Services are a subset of the online services provided by the LSST Science Platform,[44] providing access to catalog and image data within the LSST data releases as well as some management of user-generated data products. In general the services are multi-user, scalable, and designed to be deployed in a data center context rather than on individual developer machines.

The various components of the Data Access Services are built within LSST CI (Jenkins) as Docker containers. Per general LSST practice, unit tests included in each module are executed as part of the build, and the build will fail if these do not pass. In addition, some parts of the service suite have coverage by small-scale automated integration tests; these tests are run by Travis CI on each GitHub branch commit or pull request. The automated integration tests launch a constellation of containers configured with test datasets, then make a series of service requests and evaluate received responses against known/expected results.

The Data Access Services are at present deployed at scale for testing and development on three compute clusters: one thirty-node cluster at NCSA, and two twenty-five node clusters at CC-IN2P3. These deployments host a combination of synthetic test data and scientifically valid test data sourced from other astronomical surveys such as SDSS, WISE, and HSC. Scale testing is re-provisioned annually on clusters of increasing size with datasets of increasing size, on a ramp toward the scale necessary to support start of operations.

Cluster deployments of the Data Access Services containers are done at present with a collection of ad-hoc administration scripts, but the project is working currently to replace these with Kubernetes tooling.

9.3 Case Study: Deploying releases to JupyterLab

As our exploration of JupyterLab as the interactive notebook component of the LSST Science Platform[44] proceeds, we have realized the need for automated deployments of the lab environment. One of the build artifacts mentioned above is a Docker container that includes the stack. Since the JupyterLab environment is Kubernetes-ready, we have appended a phase to the stack builds that starts with the Docker stack container and adds the JupyterLab server and other Python modules we need, as well as the user provisioning machinery and Lab startup scripts. Thus, with each nightly build of the stack, we also get a nightly build of the JupyterLab environment wrapping that version of the stack. In JupyterHub, we have configuration directives that scan a Docker repository looking for an image with tags in a particular format or set of formats. From this we can construct a menu of the most recent daily, weekly, and release builds, and present a choice of stack versions to the user at JupyterHub login time.

Automated deployment serves two purposes: first, it makes it possible to immediately QA changes to the lab environment itself, since the deployment can become part of a CI process. Second, it makes the process of standing up an environment for a tutorial session or a conference extremely easy. Our initial version of the automated deployment tool relies on Google Compute Engine as the back end, AWS Route 53 as the DNS provider, and either Github or CILogon as the OAuth2 provider. We will add more options as demand warrants. The deployment tool provides a sequencer around `gcloud`, `kubectl`, and `awscli`. It is driven from a YAML file, from environment variables, or from the command line. The current version substitutes templated values from its configuration, and then runs `gcloud` and `kubectl` to create the various components of the lab installation. Future improvements may include a move to Helm charts rather than raw Kubernetes YAML, a rewrite so

that deployment becomes a Terraform configuration rather than a python module, and/or integration of TLS certificate management into the tool. A more detailed description of the LSST JupyterLab environment can be found in Ref. 53.

9.4 Deployment at IN2P3

As part of its preparation for contributing 50% of the processing of the annual LSST data releases, IN2P3 computing center (CC-IN2P3) deploys each release of the LSST software stack as soon as it is available. Two mechanisms are used for this deployment. First, a deployment of both stable and weekly releases of the LSST software is made on a networked file system, accessible from all the hosts in both the login and batch farms of the site. The second mechanism used is the CernVM File System (CernVM-FS),[54] an HTTP-based mechanism for distributing software packages over a wide area network. It is implemented as a POSIX read-only file system in user space with intelligent cache management. This file system can be mounted on a scientist's personal computer as well as on a compute node of a batch farm. This mechanism is well suited to guarantee that exactly the same software is used by scientists when developing their own software on top of LSST's and when executing their software at scale on CC-IN2P3's data processing platform.

Each deployment, be it of a weekly or a stable release, is as self-contained as possible, including most of its software dependencies and the Python interpreter it is built for. In addition, a select set of third party packages is deployed to extend the LSST software, making sure that no dependencies of the LSST stack are modified. The C++ compiler and runtime library, which the LSST software depends on, are not included in each release but are pre-installed in all hosts of the site. Each extended release is installed from binaries (if available) or from source (see §9.1.1). It is typically composed of about 150,000 files and directories which, aggregated, take about 13GB of storage. At least the 12 most recent weekly releases and all stable releases are available at any given moment.

IN2P3 scientists, the end users of this continuous deployment, use the installed releases in several ways. They prepare batch jobs which use the LSST software stack for processing simulated or precursor data. They also use those releases to launch Python notebooks for data exploration, visualization, and rapid prototyping: they can easily choose the version they want to use for a particular notebook and launch it for execution on any host of the CC-IN2P3 login farm, which is co-located with the datasets they want to process. `stackyter`,[55] a convenience SSH wrapper, is typically used for seamlessly and securely launching notebooks at CC-IN2P3 using the end user's credentials.

Dependable access to a data processing facility where scientists can use recent LSST software to process co-located datasets of significant size, without worrying about the technicalities of software deployment, has been proven very effective. It allows them to test new capabilities of the software and promptly detect regressions and provide feedback to the developers. The policy of removal of older weekly releases encourages them to continuously validate their own software against very recent LSST software.

10. CONCLUSION

The LSST Data Management Team consists of more than ninety people spread across multiple locations. In this paper we have described our current developer processes and explained both how they have evolved over time and how we foresee them evolving in the future. The LSST Data Management software development has been ongoing for at least 14 years, and we expect that our processes will continue to evolve, based on experience and new technologies, as we transition from construction to operations through to the end of the survey in 2032.

ACKNOWLEDGMENTS

We thank all the people who have contributed to the tooling, processes, and discussions over the years. This material is based upon work supported in part by the National Science Foundation through Cooperative Agreement 1258333 managed by the Association of Universities for Research in Astronomy (AURA), and the Department of Energy under Contract No. DE-AC02-76SF00515 with the SLAC National Accelerator Laboratory. Additional LSST funding comes from private donations, grants to universities, and in-kind support from LSSTC Institutional Members. This work includes a contribution funded by France's CNRS / IN2P3 (Institut National de Physique Nucléaire et de Physique des Particules).

REFERENCES

[1] Jurić, M., Kantor, J., Lim, K., et al., "The LSST Data Management System," *ArXiv e-prints* **arXiv:1512.07914** (2015).

[2] Ivezic, Z. et al., "LSST: from Science Drivers to Reference Design and Anticipated Data Products," *ArXiv e-prints* **arXiv:0805.2366** (2008).

[3] Axelrod, T. et al., "The LSST Data Processing Pipeline," *BAAS* **36**, 1529 (2004).

[4] Gill, R., Gracia, G., Lupton, R. H., and O'Mullane, W., "Novel technique for tracking manpower and work packages: a useful tool for the team and management," in [*Modeling, Systems Engineering, and Project Management for Astronomy VI*], *Proc. SPIE* **9150**, 91501E (2014).

[5] Kantor, J. et al., "Agile software development in an earned value world: a survival guide," in [*Modeling, Systems Engineering, and Project Management for Astronomy VI*], *Proc. SPIE* **9911**, 99110N (2016).

[6] Becla, J. et al., "Project Management Guide." LSST DM Tech Note DMTN-20, LSST https://dmtn-020.lsst.io (2016).

[7] O'Mullane, W. et al., "Data Management Organization and Management." LDM-294, LSST https://ls.st/LDM-294 (2017).

[8] Lim, K.-T. et al., "Data Management System Design." LDM-148, LSST, https://ls.st/LDM-148 (2017).

[9] O'Mullane, W., Jurić, M., and Economou, F., "Data Management Test Plan." LDM-503, LSST https://ls.st/ldm-503 (2017).

[10] Claver, C. F. et al., "Systems engineering in the Large Synoptic Survey Telescope project: an application of model based systems engineering," in [*Modeling, Systems Engineering, and Project Management for Astronomy VI*], *Proc. SPIE* **9150**, 91500M (2014).

[11] Ivezić, Ž. and The LSST Science Collaboration, "LSST Science Requirements Document." LPM-17, LSST https://ls.st/LPM-17 (2018).

[12] Claver, C. F. et al., "LSST System Requirements." LSE-29, LSST https://ls.st/LSE-29 (2017).

[13] Claver, C. F. et al., "Observatory System Specifications." LSE-30, LSST https://ls.st/LSE-30 (2018).

[14] Juric, M. et al., "Data Product Definition Document." LSE-163, LSST https://ls.st/LSE-163 (2018).

[15] Dubois-Felsmann, G. and Jenness, T., "Data Management System (DMS) Requirements." LSE-61, LSST https://ls.st/LSE-61 (2018).

[16] Economou, F. and Lim, K.-T., "Github migration plan." Document-17187, LSST https://ls.st/Document-17187 (2014).

[17] Wang, D. L., Monkewitz, S. M., Lim, K.-T., and Becla, J., "Qserv: a distributed shared-nothing database for the LSST catalog," in [*State of the Practice Reports*], *SC '11*, 12:1–12:11, ACM (2011).

[18] Roby, W. et al., "Firefly: embracing future web technologies," in [*Software and Cyberinfrastructure for Astronomy IV*], *Proc. SPIE* **9913**, 99130Y (2016).

[19] Bosch, J. et al., "The Hyper Suprime-Cam software pipeline," *PASJ* **70**, S5 (2018).

[20] "LSST DM Developer Guide." LSST, March 2018 https://developer.lsst.io. (Accessed: 4 April 2018).

[21] "numpydoc - Numpy's Sphinx extensions." 4 April 2018 https://numpydoc.readthedocs.io. (Accessed: 5 April 2018).

[22] van Heesch, D., "Doxygen Manual." 2017 http://www.doxygen.org/manual/. (Accessed: 5 April 2018).

[23] "@use JSDoc." JSDoc 3 documentation project, 2018 http://usejsdoc.org. (Accessed: 5 April 2018).

[24] van Rossum, G., "PEP 8 – Style Guide for Python Code." Python Software Foundation, 1 August 2013, https://www.python.org/dev/peps/pep-0008/. (Accessed: 5 April 2018).

[25] "Google C++ Style Guide." 2017 https://google.github.io/styleguide/cppguide.html. (Accessed: 5 April 2018).

[26] The Clang Team, "ClangFormat." 2018 http://clang.llvm.org/docs/ClangFormat.html. (Accessed: 5 April 2018).

[27] The Clang Team, "ClangTidy." 2018 http://clang.llvm.org/extra/clang-tidy. (Accessed: 5 April 2018).

[28] "ISO/IEC 14882:2014 – Programming Languages – C++." International Organization for Standardization, 2014 (ISO) https://www.iso.org/standard/64029.html.

[29] Jenness, T., "Porting the LSST Data Management Pipeline Software to Python 3," *ArXiv e-prints* **arXiv:1611.00751** (2016).
[30] Jenness, T., "Modern Python at the Large Synoptic Survey Telescope," *ArXiv e-prints* **arXiv:1712.00461** (2017).
[31] Krekel, H., "pytest: helps you write better programs." 2017 https://docs.pytest.org. (Accessed: 5 April 2018).
[32] Fausti, A., "The SQuaSH dashboard." LSST SQuaRE Tech Note SQR-009, LSST https://sqr-009.lsst.io (2018).
[33] Padmanabhan, N., Lupton, R., and Loomis, C., "EUPS — a Tool to Manage Software Dependencies." https://github.com/RobertLuptonTheGood/eups (2015).
[34] Jenness, T., "Porting the stack to OS X El Capitan." LSST DM Tech Note DMTN-001, LSST https://dmtn-001.lsst.io (2015).
[35] Oliphant, T. E., [*A guide to NumPy*], Trelgol Publishing (2006).
[36] Hunter, J. D., "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering* **9**, 90 (2007).
[37] The Astropy Collaboration, "The Astropy Project: Building an inclusive, open-science project and status of the v2.0 core package," *ArXiv e-prints* **arXiv:1801.02634** (2018).
[38] Berry, D. S., Warren-Smith, R. F., and Jenness, T., "AST: A library for modelling and manipulating coordinate systems," *Astronomy and Computing* **15**, 33–49 (2016).
[39] Knight, S., "Building software with SCons," *Computing in Science Engineering* **7**(1), 79–88 (2005).
[40] Preston-Werner, T., "Semantic Versioning." 2013 https://semver.org. (Accessed: 5 April 2018).
[41] Jenness, T. et al., "Investigating interoperability of the LSST data management software stack with Astropy," in [*Software and Cyberinfrastructure for Astronomy IV*], *Proc. SPIE* **9913**, 99130G (2016).
[42] McKercher, R., "Document Management Plan." LPM-51, LSST https://ls.st/LPM-51 (2017).
[43] "LSST Science Pipelines User Guide." LSST, April 2018 https://pipelines.lsst.io. (Accessed: 9 April 2018).
[44] Jurić, M., Ciardi, D., and Dubois-Felsmann, G., "LSST Science Platform Vision Document." LSE-319, LSST https://ls.st/LSE-319 (2017).
[45] Baker, M., [*Every Page Is Page One: Topic-Based Writing for Technical Communication and the Web*], XML Press (2013).
[46] Procida, D., "What nobody tells you about documentation." 2017 https://www.divio.com/en/blog/documentation/. (Accessed: 9 April 2018).
[47] Sick, J., Gill, M. S., Krughoff, S., and Swinbank, J., "Science Pipelines Documentation Design." LSST DM Tech Note DMTN-030, LSST https://dmtn-030.lsst.io (2017).
[48] Gentle, A., [*Docs like code*], Just Write Click (2013).
[49] Sick, J., "The LSST the Docs Platform for Continuous Documentation Delivery." LSST SQuaRE Tech Note SQR-006, LSST https://sqr-006.lsst.io (2016).
[50] Jenness, T., "lsst-texmf: The LSST LaTeX classes." LSST https://lsst-texmf.lsst.io (2017). (Accessed: 9 April 2018).
[51] Economou, F., Hoblitt, J. C., and Norris, P., "Your data is your dogfood: DevOps in the astronomical observatory," *ArXiv e-prints* **arXiv:1407.6463** (July 2014).
[52] Hoblitt, J., "Puppet vs. Jenkins: A Tale of Types and Providers." https://puppet.com/presentations/puppet-vs-jenkins-tale-types-and-providers (2015). (Accessed: 11 April 2018).
[53] Economou, F. et al., "We're all programmers now: the JupyterLab notebook environment of the LSST science platform," in [*Software and Cyberinfrastructure for Astronomy V*], *Proc. SPIE* **10707**, in press (2018).
[54] Blomer, J., Aguado-Sánchez, C., Buncic, P., and Harutyunyan, A., "Distributing LHC application software and conditions databases using the CernVM file system," *Journal of Physics: Conference Series* **331**(4), 042003 (2011).
[55] Chotard, N., "stackyter." 2017 http://stackyter.readthedocs.io. (Accessed: 11 April 2018).