DMTN-030: Science Pipelines Documentation Design

Jonathan Sick, Mandeep S. S. Gill, Simon Krughoff and John Swinbank

Latest Revision: 2017-01-02

DOI: 10.5281/zenodo.228174

1 Introduction

The LSST Science Pipelines are a key project in the LSST Data Management System. DM developers, data release production operators, and the LSST astronomy community at large all have a stake in using and extending the Science Pipelines. Comprehensive and usable documentation is prerequisite for the success of these stakeholders.

This technical note describes a design for the LSST Science Pipeline's user guide. The initial design was developed by the authors at a design sprint held in Tucson, AZ, over December 7 to 9, 2016.[1] As the documentation design is implemented and improved, this technical note will be updated and serve as a reference for the project's design philosophy and direction. The LSST Science Pipelines documentation is implemented consistently with LDM-493 Data Management Documentation Architecture's *user guide* project class.

[1] The meeting notes are archived with this technical note as a PDF, and also available online at http://ls.st/507.

Meeting **Notes (PDF)**

The design is presented as follows. In Section 2 we define the scope of the LSST Science Pipelines as a software and documentation project. Section 3 describes the audience of this documentation project. Section 4 lays out the *docs-as-code* technical framework within which the documentation is produced and delivered.

In later sections we present the information architecture of the Science Pipelines documentation. Section 5 introduces *topic-based* documentation as a core design philosophy. Topic-based documentation motivates us to build standardized documentation *types* that guide authors and also establish patterns that benefit readers. In Section 6 we frame these topic types by mocking up the homepage of the Science Pipelines documentation website. Following this we present designs for topic types:

- Section 7: Processing topic type.
- Section 8: Framework topic type.
- Section 9: Module topic type.
- Section 10: Task topic type.
- Section 11: API reference topic type, including a discussion of prototypes.

2 Scope of the Science Pipelines documentation project

One of the basic questions our design sprint needed to address was *what is the scope of pipelines.lsst.io?* Typically this question is defined by a product marketing team, or by a source code repository (for instance, https://docs.astropy.org is documentation for the https://github.com/astropy/astropy software project). In our case, the situation is complicated by the fact that Data Management software is being built by multiple teams across many coupled repositories. Taken together, the software repositories of the Data Management System are typically called *the Stack*, but not all parts of the Stack are used together. There are server-side database and display components, as well as pipelines algorithms bound together with middleware.

In this documentation design, our stance is to label all Data Management client-side code that is imported as the `lsst` Python package as *The LSST Science Pipelines*. Specifically:

- Core frameworks, like `afw`, are included.
- The client-side Butler is included.
- Middleware that is integral to the `lsst` python package, like `logging`, is included. The backend counterparts, like an ELK stack to receive log messages is not included.
- Tasks and other algorithmic components are included.
- Display packages are included, like `display_firefly`, though Firefly itself is not.
- Observatory interfaces maintained by DM are included.

- Despite being in the `lsst` namespace, LSST Simulations software **is not included** since it is managed and versioned separately. This could be redressed in the future, given how tightly coupled the DM and Simulations software is.

This package collection crosses DM team lines, but reflects how the software is used, and how it can be maintained as an open source project into the future.

> **Note**
>
> A related issue is the name of the software product itself. This segment of the Data Management System does not have name; while the name "Stack" is often used, that refers to all software in the DMS.
>
> This technote uses the branding "LSST Science Pipelines" and "pipelines.lsst.io," though technically this is an appropriation of the collective name of work by the University of Washington and Princeton teams. The Science Pipelines product and documentation site also includes elements from the Data Access, Science User Interface Toolkit, Middleware and SQuaRE teams.
>
> Overall, the product naming is an unresolved issue and we acknowledge that the pipelines.lsst.io branding may change.

2.1 Developer documentation

A related issue is how developer documentation for the Science Pipelines should be managed. Currently, the separate DM Developer Guide contains a mixture of organizational policy documents and technical documentation needed to build DM software—including the LSST Science Pipelines. We propose to move developer documentation tightly coupled to the LSST Science Pipelines from the DM Developer Guide project to pipelines.lsst.io, such as:

- Build system information (`lsst-build`, `lsstsw`).
- How to create packages in `lsst`.
- How to write tests in the `lsst`.
- Patterns for using EUPS.

Some information in the DM Developer Guide is applicable beyond the Science Pipelines project. In this case, information can be merely linked from pipelines.lsst.io to developer.lsst.io, such as:

- Code style guides.
- General documentation writing patterns.

Keeping policies like these in developer.lsst.io allows them to be centrally referenced by other DM projects.

2.2 Relationship to other DM documentation projects

The tight focus of pipelines.lsst.io necessitates other DM user guide projects. This is anticipated by LDM-493.

We expect pipelines.lsst.io to be joined by other software documentation projects:

- Firefly/SUIT.
- Qserv.
- Webserv/DAX.
- Various projects by SQuaRE.

There will also be "data" documentation in the operations era:

- Science Platform user guide.
- Alert Production user guide (for services and scientists that consume the alert stream).
- Data Release documentation projects, archived for each data release (DR1, DR2, and so on).

Note that the alert and data release documentation will consume the Science Pipelines documentation: those projects will describe pipelines built *from* the LSST Science Pipelines and use the algorithmic descriptions published with the pipelines code on pipelines.lsst.io.

Finally, there will be operations guides that, like developer.lsst.io, document internal processes for operating LSST.

3 The audiences of pipelines.lsst.io

Audience needs drive design. In this documentation design exercise, we attempted to identify who the core users of the LSST Science Pipelines are, and what their documentation needs are.

We identified the following groups:

- **DM Developers.** From a construction standpoint, this is the key audience. It is also the most active audience. DM developers need:
  - Complete and accurate API references. Currently developers learn by introspecting APIs and reading other code and tests that consume an API. The current Doxygen site is not useful.
  - Descriptions of how tasks fit together; both API-wise and from a high-level algorithmic perspective.
  - Examples.
  - Tutorials and other documentation for running pipelines on validation clusters.
- **Construction-era science collaborations.** We find that most astronomers using the LSST Science Pipelines are not processing data, but rather are using it as a core library for the Simulations stack (such as the Metrics Analysis Framework to motivate the Observing Strategy Whitepaper).
- **DESC.** This collaboration is exceptional in that it wants to process data using the Science Pipelines, contribute packages (Twinkles), and provide algorithmic feedback. DESC needs:
  - Developer documentation (to support their own package development).
  - Algorithm background (to comment on).
  - Documentation on how to run pipelines on their own infrastructure.
- **LSST operators and scientists in operations.**
  - The Science Directorate will have very similar needs to DM developers now.
  - DRP will need documentation on how the Science Pipelines work, but will augment this with internal operations guides (which are out of scope).
- **Astronomers using the LSST Science Platform in operations.**
  - To some extent, LSST usage will be similar to that seen for SDSS: small queries to subset data; complex queries to get objects of interest.
  - Astronomers will want to run pipeline tasks on a subset of data with customized algorithms.
  - Astronomers will use the Butler to get and put datasets within their storage

quota.
    - Astronomers will develop and test algorithms that may be proposed for incorporation in DRP (though an atypical scenario).
- **Other observatories and surveys.**

We conclude that DM is the most active user group, and also has the greatest documentation needs. The needs of other groups are consistent with DM's. From a user research perspective this is useful. If we build pipelines.lsst.io with DM's own needs in mind, the documentation will also be immediately useful for other groups. As the project matures, we can add tutorial documentation to help other user groups.

4 Documentation as code

The Science Pipelines documentation uses a *documentation-as-code* architecture. Documentation is stored and versioned in pipelines package repositories, and built by DM's standard continuous integration system with Sphinx into a static HTML site that is published to the web with LSST the Docs. This section outlines the basic technical design of the Science Pipelines documentation.

4.1 Documentation in packages

Each EUPS-managed LSST Science Pipelines package contains a `doc/` directory. All documentation specific to that package is contained within the `doc/` directory. A minimal `doc/` directory looks like:

```
doc/
  index.rst
  tasks/
  _static/
```

Here, `index.rst` is a reStructuredText document we refer to as a module topic, described later. The `tasks/` directory hosts task topic pages for any pipeline tasks implemented in that module. The `_static/` directory is a space for non-reStructuredText content that is copied verbatim to the built site.

Most Science Pipelines packages host a single module, but there are exceptions like `afw`. In this case, there is an additional level of directories:

```
doc/
  index.rst
  cameraGeom/
    index.rst
    tasks/
    _static/
  coord/
  detection/
  display/
  fits/
  geom/
  gpu/
  image/
  math/
  table/
```

Each `afw` module subdirectory contains the `index.rst`, `tasks/` and `_static/` structure. The root `doc/index.rst` file is a minimal file containing a toctree that is only used for local, single package documentation builds.

Note

The `doc/` directory was already used by the previous Doxygen-based documentation build system. However, during the transition from Doxygen to Sphinx-based builds, we do not expect any conflicts since content for the two system reside in non-overlapping files (`.dox` versus `.rst` files for Doxygen and Sphinx, respectively). It should be possible to continue to build a Doxygen version of the documentation while the new Sphinx site is being prepared.

4.2 Per-package documentation builds

Developers can build documentation for individual cloned packages by running `scons sphinx` from the command line. This matches the workflow already used for code development. Developers will build documentation for individual packages in development environments to preview changes to module documentation, including conceptual topics, examples, tasks, and API references.

Note

The Doxygen-based build system uses a `scons doc` build command. This command (notwithstanding a likely rename to `scons doxygen`) will remain to support Doxygen generation of C++ API metadata.

Internally, the `scons sphinx` command replaces the `make html` and `sphinx-build` drivers normally used for Sphinx documentation. By integrating with Sphinx's internal Python APIs, rather than using `sphinx-build`, we avoid putting `conf.py` Sphinx project configuration information in each package's `doc/` directory. Instead, Sphinx configuration is centrally managed in SQuaRE's documenteer package, through sconsUtils.

Note

The single package documentation builds omit content from related packages, but will generate warnings about links to non-existent content. This is an acceptable trade-off for a development environment. In the continuous integration environment, where all documentation content is available, documentation builds can be configured to fail on broken links.

4.3 Integrated documentation: the pipelines_lsst_io repository

Besides documentation embedded in Science Pipelines packages, there is a core documentation repository: https://github.com/lsst/pipelines_lsst_io. This repository hosts higher-level documentation that crosses modules, including: installation guides, release notes, getting-started tutorials, processing and framework documentation.

When pipelines_lsst_io is built, the `doc/` directories of each package is linked into the cloned pipelines_lsst_io repository:

```
pipelines_lsst_io/
   index.rst
   _static
   ...
   afw/ -> linked or copied from afw/doc/
   pipe_base -> linked or copied from pipe_base/doc/
```

With these linked package `doc` directories, the Sphinx build for `pipelines_lsst_io` is able to build all documentation simultaneously, and resolve all links within the project.

The LTD Mason tool (see SQR-006) was designed to make the package documentation links, assuming that lsstsw was being used (as it is in the Jenkins environment). However, it may be more appropriate to make pipelines_lsst_io agnostic of lsstsw, which implies that pipelines_lsst_io should itself be an EUPS-managed package, and that its build logic should also be hosted in `sconsUtils`.

5 Topic-based documentation

The Science Pipelines documentation, following the proposed LDM-493 Data Management Documentation Architecture, will use *topic-based* content organization. In this approach, content is organized into self-contained, well-structured pieces that support a reader's current task, and that link the reader to other topics for related information. Topic-based content differs from books, research articles, and indeed, paper manuals, where a reader is expected to gradually accumulate knowledge and context through a preset narrative. In topic-based documentation, readers drop into specific pages to support a specific objective, but can follow links to establish context, and in effect, build a personalized curriculum for learning and using the software.

The topic-based documentation approach is widely employed by the technical writing community. Help sites from companies like GitHub and Slack use topic-based documentation. Open source projects like Astropy also use elements of topic-based documentation philosophy. Wikipedia is an excellent example of organically-evolving topic-based documentation. The (currently proposed) LDM-493 Data Management Documentation Architecture also identifies topic-based documentation as the information architecture of DM's user guides. Every Page is Page One by Mark Baker,[2]

hereafter referred to as *EPPO*, describes the motivation and principles of topic-based documentation. This section is intended to brief the Data Management team on topic-based documentation concepts, following EPPO.

[2] Baker, Mark. *Every page is page one: topic-based writing for technical communication and the web*. Laguna Hills, CA: XML Press, 2013.

In topic-based documentation, a topic *often* maps to an individual HTML page or reStructuredText source file. Baker, in EPPO §6.8, identified the following design principles of topics:

1. Self-contained.
2. Specific and limited purpose.
3. Conform to type.
4. Establish context.
5. Assume the reader is qualified.
5. Stay on one level.
6. Link richly.

5.1 Topic types

In this planning exercise, principle #3 is immediately relevant. Topic types are predefined structures for documentation, and every implemented topic page inherits from one of these types. For writers, topic types are valuable since provide a strong template into which the domain expert's knowledge can be filled. Topic types are also valuable to readers since they provide systematic patterns that can be learned and used as wayfinding. For example, Numpydoc Python API references are built on a topic type (perhaps a few, for classes and for function and for indices). Authors immediately know how to write a Python API references, and readers immediately know how to use a Python API reference page when they see one.

Defining topic types requires us to identify every kind of documentation we will write (alternatively, each kind of thing that must be documented). Then the topic type can be implemented as a template given to authors. The template itself imbues topic instances with many of the characteristics of an EPPO topic, but should be backed up with authoring instructions. In the 6 Designing the homepage section of this technote we identify the Science Pipeline's primary topic types, and those types are designed in later sections.

5.2 Topic scope

Another aspect critical for this planning exercise is that topics are self-contained (principle #1), and have a specific and limited purpose (#2). This means that each documentation page can be planned with an outline:

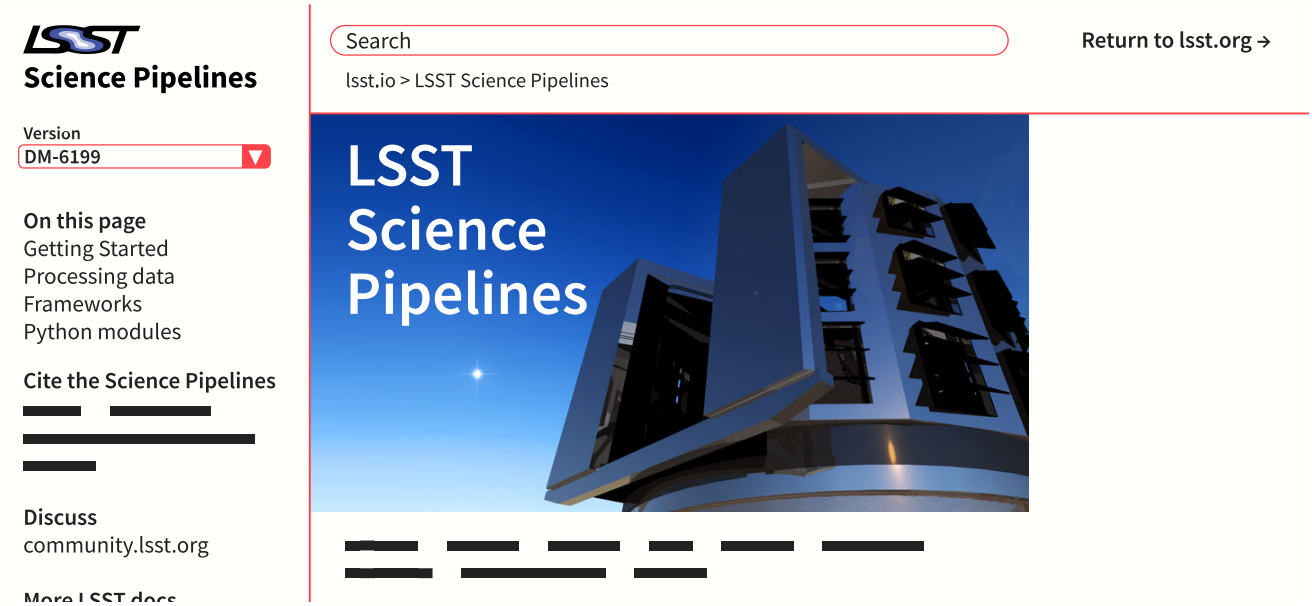
- A title.
- A purpose and scope.
- Known related documents that the page can link to, rather than duplicate.

Thus many authors can work on separate topics in parallel, knowing that each topic conforms to a uniform type pattern, and is supported by other topics. In software architecture parlance, *topics have well-defined interfaces*. Topic-based documentation can be much more effectively planned and managed than traditional narrative writing.

6 Designing the homepage

Identifying an inventory of topic types requires foresight of what the Science Pipelines documentation will become. We approached this by first designing the homepage for the Science Pipelines documentation. The homepage provides a unique, overarching view into the inventory of content needed to document the full domain of the Science Pipelines for all expected audiences.

We envision four distinct areas on the homepage: preliminaries, processing, frameworks, and modules. Each area organizes and links to topics of distinct types. The following sub-sections motivate each area of the homepage, while later sections describe the design of each topic type in greater detail.



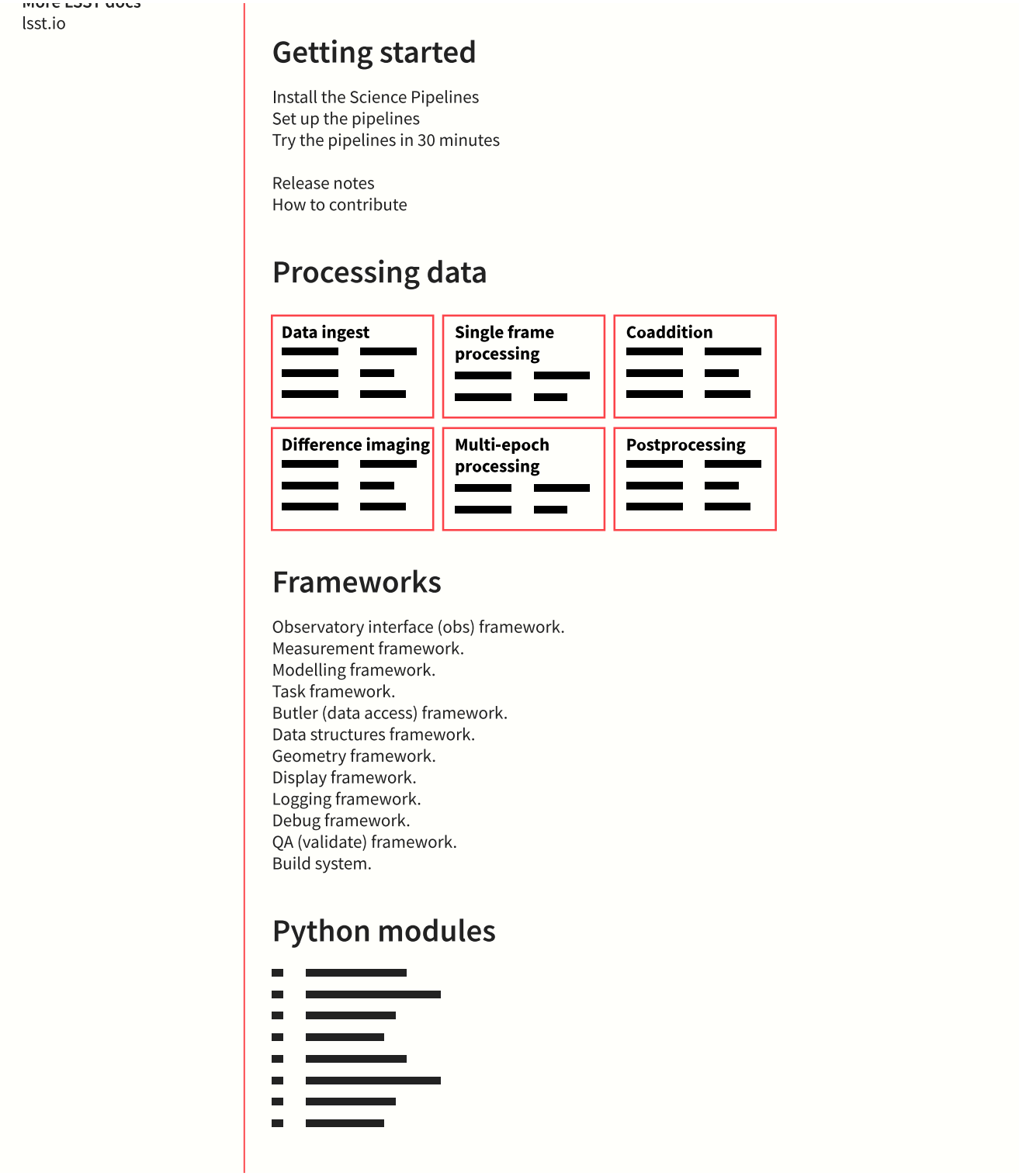


Figure 1 Mockup of the Science Pipelines documentation homepage layout.

6.1 Preliminaries section

Above the fold, the first task of the documentation site is to address readers who have little or no knowledge with the LSST Science Pipelines. This area contains:

- A blurb that introduces readers to the Science Pipelines.

- Links to installation topics, and EUPS usage topics.
- Links to release note topics.
- An invitation to try a quick tutorial that helps a reader understand what the Science Pipelines feel like to use.
- Links to topics describing how to contribute to the LSST Science Pipelines.
- An explanation of how to get help with the LSST Science Pipelines (that is, link to https://community.lsst.org).

In summary, this section establishes the LSST Science Pipelines as an open source software project.

6.2 Processing section

A common task for readers (both community end users, and indeed, the DM team itself) is to *use* the Science Pipelines. This is a distinct viewpoint from documenting the Science Pipelines's implementation (modules, and frameworks). Rather, the processing section focuses on how to organize data, use command line tasks to process data, and consume those outputs for scientific investigations. And while topics in the processing section defer to task topics as definitive self-contained scientific descriptions of algorithms, the processing section is used to frame these command line tasks and help astronomers make judgments about how they are used for science.

Based on the Twinkles pipeline, which uses the LSST Science Pipelines, we recognized that processing can be organized into *contexts*. Each context has a well-defined type of input, well-defined types of outputs, and well-defined types of measurements. A core set of contexts is:

1. Data ingest. *Organizing Butler repositories for different observatories.*
2. Single frame processing. *This is primarily an introduction to ProcessCcdTask.*
3. Coaddition processing.
4. Difference image processing.
5. Multi-epoch measurement.
6. Postprocessing. *This is a discussion of output catalogs, and may need more fine-grained topical organization.*

On the homepage, each listed context is a link to a processing topic.

Note

LDM-151 §5 follows a similar pattern:

- §5.1: Image Characterization and Calibration
- §5.2: Image Coaddition and Differencing
- §5.3: Coadd Processing
- §5.4: Overlap Resolution
- §5.5: Multi-Epoch Object Characterization
- §5.6: Postprocessing

While LDM-151's sectioning makes sense for motivating algorithm development (LDM-151's purpose), we believe that real-world usage warrants our sugggested organization of the processing section.

6.3 Frameworks section

To bridge high level usage documentation to low-level API references, we realized that frameworks are an ideal platform for introducing and framing implementation documentation. Frameworks are collections of modules (possibly crossing EUPS packages) that implement functionality. Examples of frameworks are:

- Observatory interface (obs) framework.
- Measurement framework.
- Modelling framework.
- Task framework.
- Butler (data access) framework.
- Data structures framework.
- Geometry framework.
- Display framework.
- Logging framework.
- Debug framework.
- QA (validate) framework.
- Build system.

By organizing topics around frameworks, we have a platform to discuss their functionality for both end users (how to use the framework's features) and developers (patterns for developing in and with the framework) in a way that's not constrained by

implementation details (module organization).

The frameworks section of the homepage lists each framework's name, along with a descriptive subtitle. Each item is a link to a corresponding framework topic.

6.4 Modules section

The final section of the homepage is a comprehensive listing of modules in the LSST Science Pipelines. Each item is a link to a corresponding module topic. This listing will be heavily used by developers seeking API references for the modules they are using on a day-to-day basis.

These module topics are imported from the doc/ directories of each Science Pipelines EUPS package. The homepage's module listing can be automatically compiled in a custom reStructuredText directive.

7 Processing topic type

Processing topics are a practical platform that discuss how to calibrate, process, and measure astronomy datasets with the LSST Science Pipelines. As discussed in 6 Designing the homepage, processing is organized around *contexts*, such as single frames, coaddition, or difference imaging. Each processing context has a main page that conforms to the processing topic type.

Processing topics consist of the following components:

- Title.
- Context.
- In depth.
- Tutorials.
- Command line tasks.

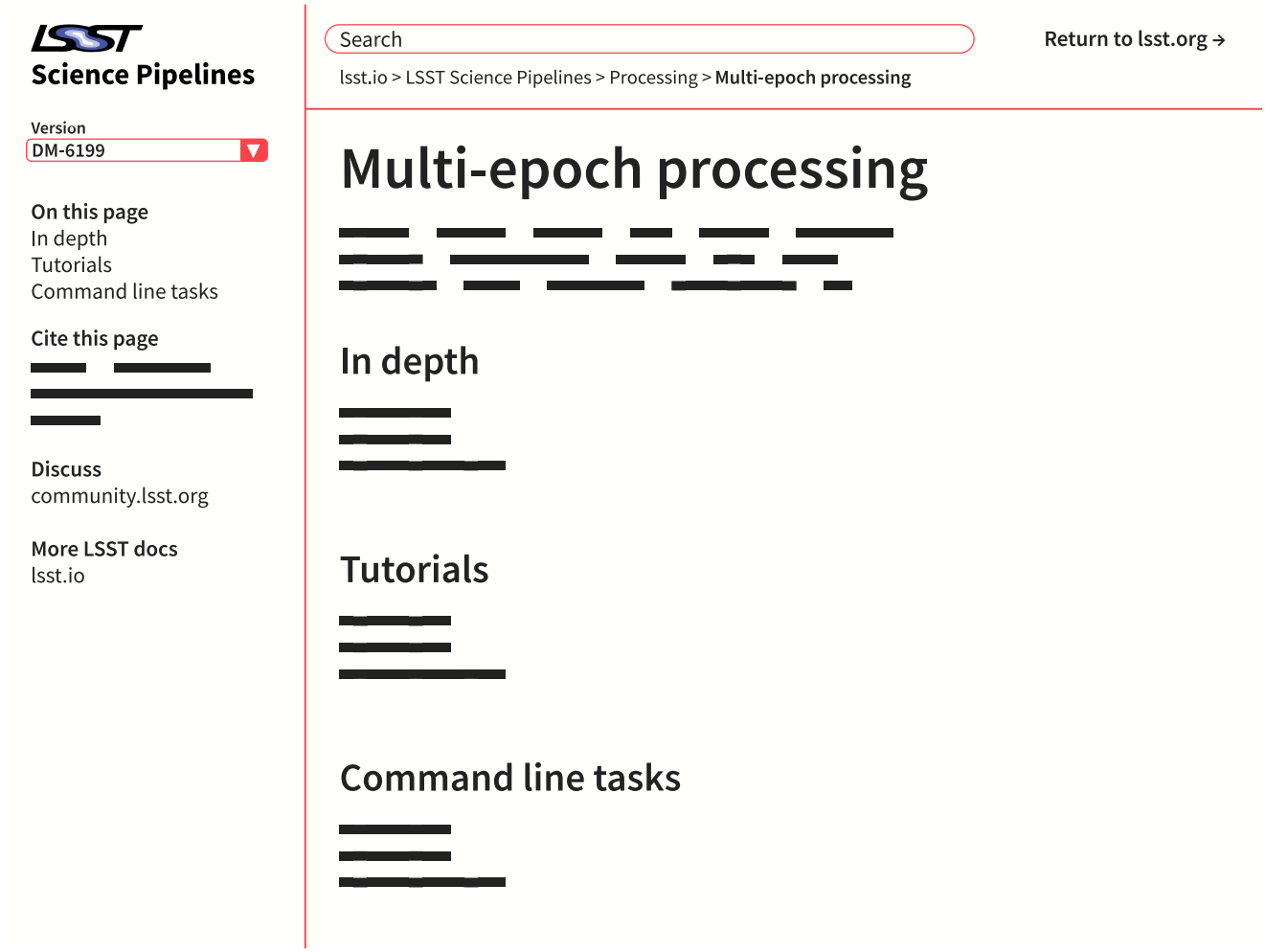


Figure 2 Mockup of processing topics.

7.1 Title

The name of a processing topic is the name of the processing context. For example, “Single frame processing” or “Multi-epoch processing.”

7.2 Context

Within a couple of short paragraphs below the title, this component establishes the topic’s context:

- Explain what the processing context *means* in non-jargon language. What data goes in? What data comes out?
- Link to adjacent processing contexts. For example, a single frame processing topic should mention and link to the data ingest topic.
- Mention and link to the main command line tasks used in this context.
- Suggest and link to an introductory tutorial for this processing context.

7.3 In depth

This section lists and links (as a `toctree`) to separate topic pages. Each of these self-contained topics provide in-depth background into aspects of processing in this context. They should primarily be written as narrative glue to other types of documentation, including frameworks, tasks, and modules. That is, these topics are guides into understanding the Science Pipelines from a practical data processing perspective. The first in depth topic should be an 'Overview' that describes the processing context itself, and introduces other in-depth topics and tutorials.

Based on experience from Twinkles, many of these topics can be divided into two halves: processing data in this context, and measuring objects from the products of that processing. Processing topic pages have the flexibility to organize in depth topics (and tutorials, below) around themes like this.

7.4 Tutorials

The Tutorials section links (as a `toctree`) to tutorial topic pages that demonstrate processing real datasets in this context. These tutorials should be easily reproduced and run by readers; necessary example datasets should be provided.

These tutorials might be designed to be run as a series across several processing contexts. For example, a tutorial on ingesting a dataset in the "ingest" context may be a prerequisite for a `processCcd` tutorial in a "single frame processing" context.

7.5 Command line tasks

Command line tasks are the primary interface for processing data with the Science Pipelines. This final section in a processing topic lists all command line tasks associated with that processing context. Links in this `toctree` are to task topics.

Note that only *command line* tasks associated with a context are listed here. Processing topics are designed to be approachable for end users of the Science Pipelines. Command line tasks are immediately usable, while sub-tasks are only details for configuration (that is, re-targettable sub tasks) or for developers of new pipelines. Thus mentioning only command line tasks gives users a curated list of runnable tasks. As a user gains

experience with command line tasks like `processCcd.py`, they will gradually learn about sub-tasks through links built into the task topic design. This pathway graduates a person from being a new to experienced user and even potentially a developer.

8 Framework topic type

Frameworks in the LSST Science Pipelines are collections of modules that provide coherent functionality. Rather than only documenting modules in isolation, the framework topic type is a platform for documenting overall concepts, design, and usage patterns for these frameworks that cross module bounds. Examples of frameworks were listed earlier in the homepage design section.

8.1 Topic type components

Each framework has a homepage that conforms to the *framework topic type*, which has the following components:

- Title.
- Context.
- In depth.
- Tutorials.
- Modules.

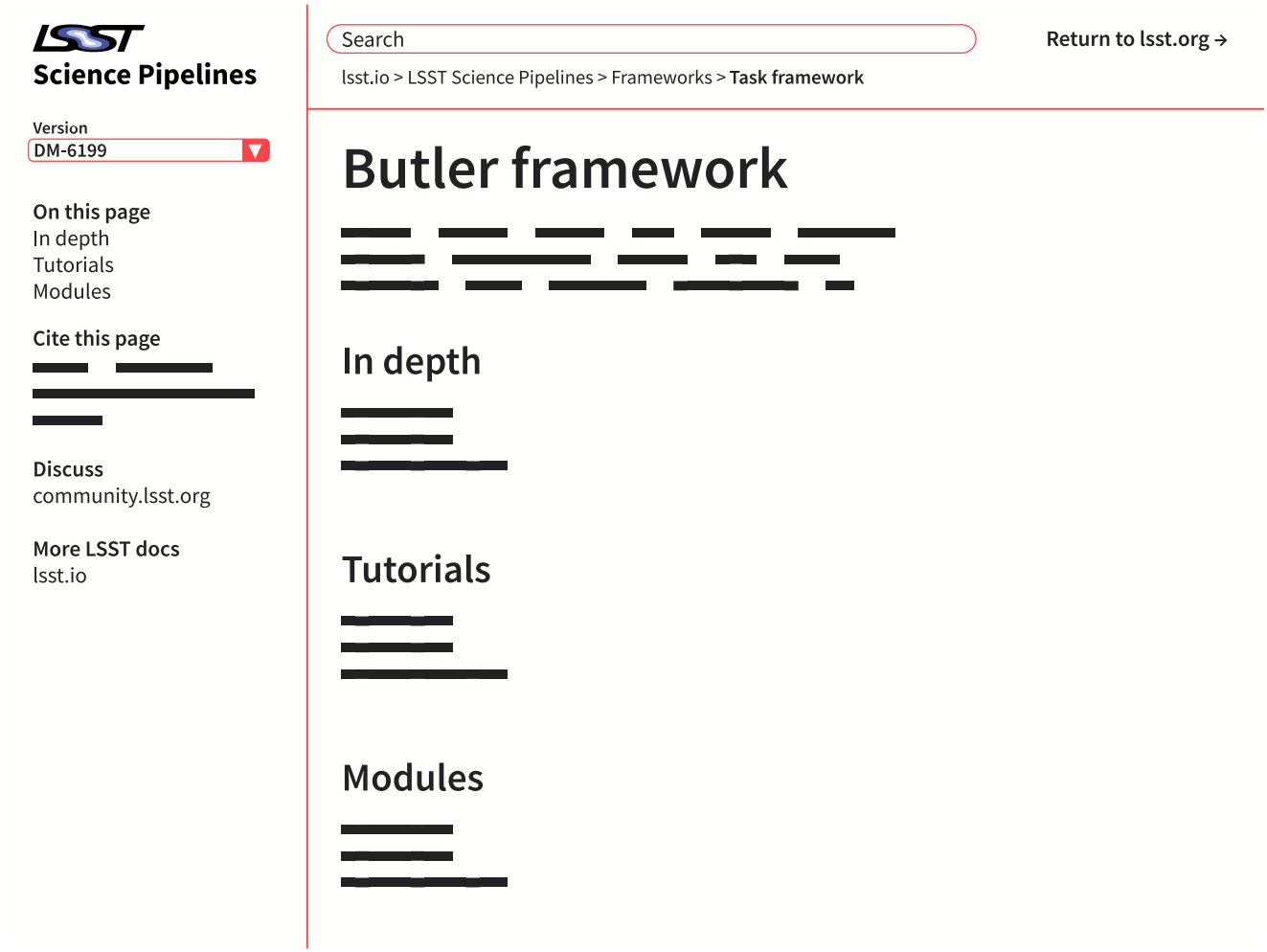


Figure 3 Mockup of the framework topic type.

8.1.1 Title

The title of the framework's topic is simply the name of the framework itself.

8.1.2 Context

Following the title, the initial few paragraphs of the topic should establish context. A context paragraph establishes what the framework is for, and what the framework's primary features or capabilities are.

8.1.3 In depth

This section provides a table of contents (`toctree`) for additional topics that cover individual framework concepts. Concept topics can include guides for developing against the framework, and descriptions of the basic ideas implemented by the framework. '*Concept*' is purposefully ambiguous but we require that concept topic pages follow the design principles of topic-based documentation.

Generally, the first topic should be an overview. The overview topic's narrative introduces and links to other framework topics.

8.1.4 Tutorials

The Tutorials section provides a table of contents (`toctree`) linking to separate tutorial topic pages. These tutorials demonstrate and teach how to use and develop in the framework.

> **Note**
>
> Additional design work is required for tutorial topic types.

8.1.5 Modules

This section lists and links to the module topic of all modules included in a framework. These links establish a connection between the high-level ideas in a framework's documentation with lower-level developer-oriented details in a module's documentation.

8.2 Framework topic type extensibility

The components described above are a *minimum* set used by each framework topic. Some frameworks may add additional components. For example, the measurement framework might include an index of all measurement plugins. The task framework might include an index of all tasks.

9 Module topic type

The module topic type comprehensively documents a *module* as an entrypoint to task and API references associated with a specific part of the codebase.

The module topic type consists of the following components:

- Title.
- Summary paragraph.
- See also.
- In depth.

- Tasks.
- Python API reference.
- C++ API reference.
- Packaging.
- Related documentation.

lsst.afw.table — Table data structures

See also

In depth

Tasks

Python API reference

C++ API reference

Packaging

lsst.afw.table is part of the afw EUPS package. Fork on GitHub.

Top-level packages

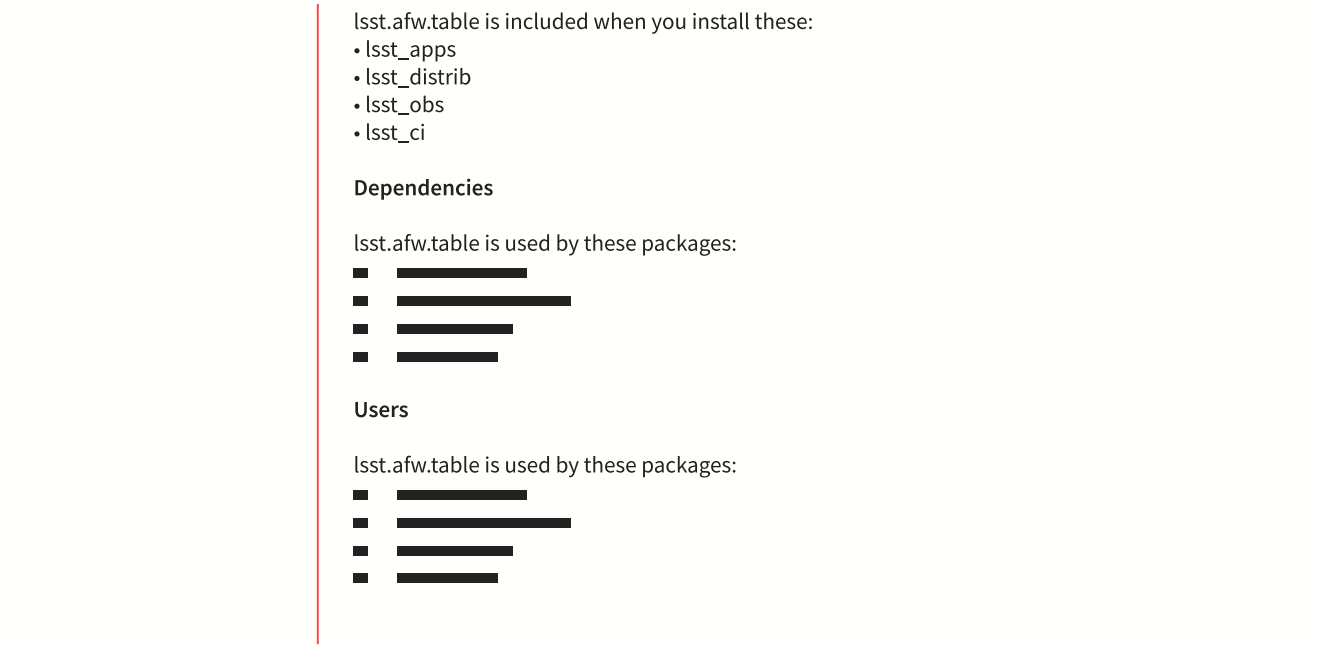


Figure 4 Mockup of module topic types.

9.1 Title

Since "module" is a Python-oriented term, the title should be formatted as: "python module name — Short description." For example:

`lsst.afw.table` — Table data structures.

9.2 Summary paragraph

This paragraph establishes the context of this module and lists key features. This section is intended to help a reader determine whether this module is relevant to their task.

9.3 See also

Right after the summary paragraph, and within a `seealso` directive, this component links to other parts of the documentation that do not otherwise follow from the topic type design. For example, if the module is part of a framework, that framework's page is linked from here. This component can also be used to disambiguate commonly-confused modules.

9.4 In depth

This section lists and links to conceptual documentation pages for the module. Each conceptual documentation page focuses on a specific part of the API and dives into features while providing usage examples. The topics can also document architectural decisions. These pages are similar to the conceptual documentation provided in the "Using" sections of Astropy sub-packages (see Using table for examples). The `lsst.validate.base` prototype documentation (currently available at https://validate-base.lsst.io) includes examples of such conceptual documentation pages as well.

9.5 Tasks

This section lists and links to task topics for any tasks implemented by this module. The task topic type is discussed in 10 Task topic type.

Minimally, this section should be a simple list where the task name is included first as a link, followed by a short summary sentence.

> **Note**
>
> It may be useful to distinguish tasks usable as command line tasks from plain tasks. Perhaps the two types could be listed separately, with command line tasks appearing first.

9.6 Python and C++ API reference

These sections list and link to reference pages for all Python and C++ API objects. Individual functions and classes are documented on separate pages. See 11 API Reference Documentation Prototypes for a discussion of API reference pages.

9.7 Packaging

Modules exist inside EUPS packages. This section is designed to help a user understand how to access a module, and understand how this module's package relates to other packages in the Science Pipelines documentation by:

- Stating what package a module is part of.
- Linking to that package's GitHub repository.
- Stating what top-level packages include this module's package. This help readers

understand what package to install.
- Stating what packages depend on this module's package, distinguishing between direct and in-direct dependencies. This will help developers.
- Stating what packages in the LSST Stack dependent on this package. Again, this will primarily help developers.

The package dependencies can be expressed as both lists and graph diagrams.

9.8 Related documentation

Modules will be documented and discussed elsewhere. This section consists of a listing of other documents related to this module, including:

- Design documentation.
- Technotes.
- RFCs.
- Community forum conversations.

For the last item, we envision a service that can monitor https://community.lsst.org forum conversations for mentions of pre-defined keywords and automatically populate a list of related forum posts. Linking documentation to the Community forum will help make the documentation interactive. With minimal overhead, a reader can begin to discuss and ask questions about documentation and the LSST Science Pipelines.

10 Task topic type

The *task* topic type defines how tasks in the LSST Science Pipelines are documented. Tasks are basic algorithmic units that are assembled into processing pipelines. Astronomers will use task topics to understand these algorithms and identify implications for their science. Users will refer to task topics to learn how to configure and run pipelines. Developers will use task topics to learn how to connect tasks into pipelines. Thus these topics are important for both astronomy end users and developers.

Currently the Science Pipelines have two flavors of tasks: tasks, and *command line* tasks. Though command line tasks have additional capabilities over plain tasks, those capabilities are strict supersets over the regular task framework. In other words, a command line task is also a task. The topic type design reflects this by making no

significant distinction between tasks and command line tasks with two design principles. First, command line task topics will have additional sections. Second, tasks and counterpart command line tasks are documented as the same identity.

Soon, a new SuperTask framework will replace command line tasks (though like command line tasks, they are still subclasses of a base `Task` class). SuperTasks will allow a task to be activated from a variety of contexts, from command line to cluster workflows. By documenting the core task and extending that documentation with additional 'activation' details, the task topic type should gracefully evolve with the SuperTask framework's introduction.

A task topic consists of the following components:

- Title.
- Summary sentence.
- Processing sketch.
- Module membership.
- See also.
- Configuration.
- Entrypoint.
- Butler inputs.
- Butler outputs.
- Examples.
- Debugging variables.
- Algorithm details.

Note

This topic design replaces earlier patterns for documenting tasks. Archives of documentation for the previous system are included in this technote.

- How **to Document a Task** (Confluence; September 23, 2014).
- AstrometryTask: example of task documentation implemented in Doxygen.

ProcessCcdTask

On this page
Configuration
Entrypoint
Butler inputs
Butler outputs
Examples
Debugging variables
Algorithm

Cite this page

Discuss
community.lsst.org

More LSST docs
lsst.io

ProcessCcdTask is part of the lsst.pipe.tasks module.

See also

Configuration

Entrypoint

Butler inputs

Butler outputs

Examples

Debugging variables

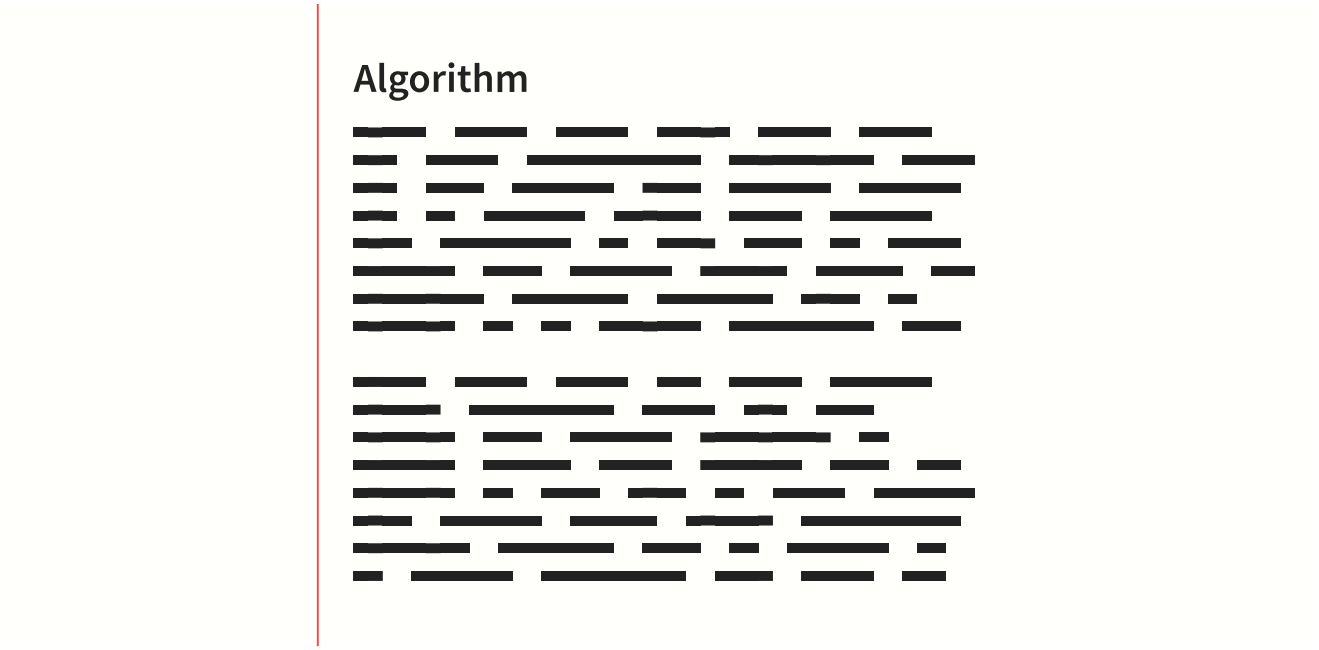


Figure 5 Mockup of the task topic type.

10.1 Title

A task topic's title is the name of the task's class. For example,

> ProcessCcdTask

Note

Following the design principle that command line tasks be documented with the underlying task itself, the title should *not* be the command line script's name, such as "processCcd.py."

We should monitor how the SuperTask command line activator refers to tasks; it may make sense for SuperTasks to always use the task's class name rather than use an alternate form.

Note

An alternative to forming the title from *only* the task's class name is to add a description, for example:

> ProcessCcdTask — Calibrate and measure a single exposure

A summary is already provided with the 10.2 Summary sentence component, but including a summary in the title may improve the usability of the task listing from module topics and processing topic pages.

10.2 Summary sentence

This sentence, appearing directly below the title, has two goals: indicate that the page is for a task, and succinctly describe what the task does. This sentence is important for establishing *context*; readers should be able to use this sentence to quickly determine if the page is relevant to their task.

10.3 Processing sketch

Appearing after the summary sentence as one or more distinct paragraphs and lists, this component provides additional details about what the task does. A processing sketch might list the methods and sub-tasks called, in order or execution. Mentions of methods and sub-tasks should be linked to the API reference and task topic pages, respectively, for those objects.

Like the summary sentence, this component is intended to quickly establish the task's context. This sketch should not be extensive; detailed academic discussion of an algorithm and technical implementation should be deferred to the 10.12 Algorithm notes component.

10.4 Module membership

In a separate paragraph after the processing sketch, this component states what module implemented the task:

AstrometryTask is part of the `lsst.meas.astrom.astrometry` module.
The module mention is a link to the module's topic.

This component establishes the task's code context, which is useful for developers.

10.5 See also

Wrapped inside a `seealso` directive, this component links to related content, such as:

- Tasks that commonly use this task (this helps a reader landing on a “sub task’s” page find the appropriate driver task).
- Tasks that can be used *instead of* this task (to link families of sub tasks).
- Pages in the Processing and Frameworks sections of the Science Pipelines documentation.

10.6 Configuration

This section describes the task’s configurations defined in the task class’s associated configuration class. Configuration parameters are displayed similarly to attributes in Numpydoc with the following fields per configuration:

- Parameter name.
- Parameter type. Ideally the parameter type links to a documentation topic for that type (such as a class’s API reference).
- A description sentence or paragraph. The description should mention default values, caveats, and possibly an example.

We anticipate that a reStructuredText directive can be built to automatically generate this topic component.

10.7 Entrypoint

The entrypoint section documents the task’s ‘run’ method. Note that task run methods are not necessarily named ‘run,’ nor do they necessarily share a uniform interface.

Initially this section will only contain the namespace of the run method, such as

```
lsst.meas.astrom.astrometry.AstrometryTask.run
```

(with the namespace linked to the method’s API reference).

Later, a custom directive may automatically replicate information from the method’s API reference and insert it into the Entrypoint section (recall that topics should be self-contained).

Todo

We may also need to add a section on Task class initialization.

10.8 Butler inputs

This section documents datasets that this task (as a command line task) consumes from the Butler.

For each `Butler.get()`, this section lists standardized entries with:

- Dataset type (linked to the dataset type's class documentation).
- A free-form description.

We anticipate that the SuperTask framework will provide hooks for auto-documenting this.

10.9 Butler outputs

This section documents datasets that this task (as a command line task) consumes from the Butler.

For each `Butler.put()`, this section lists standardized entries with:

- Dataset type (linked to the dataset type's class documentation).
- A free-form description.

Again, we anticipate that the SuperTask framework will provide hooks for auto-documenting this.

10.10 Examples

The section provides one or more runnable examples that demonstrate both the task's usage within Python, and from teh command line.

More design work is needed to implement examples. The examples should fulfill the following criteria:

- Test data sets to run the example should be documented and made accessible to the reader.

- The example should be runnable by a reader within minimal work. That is, the example includes all surrounding boilerplate.
- The example should also be runnable from a continuous integrated context, with verifiable outputs.
- Where an example includes a large amount of boilerplate, it should be possible to highlight the parts most relevant to the task itself.

Many tasks already have associated examples in the host package's `examples/` directory. As an early implementation, these examples can be copied into the documentation build and linked from this section. For example:

> **Examples**
>
> - `exampleModule.py` — Description of the example.

10.11 Debugging variables

This section documents all variables available in the task for the debugging framework. Like Numpydoc 'Arguments' fields, for each debug variable the following fields are documented:

- Variable name.
- Variable type (linking to the type's API reference).
- Free-form description. The description should indicate default values, and if the variable is a complex type, include an example.

This section also includes a link to the debug framework's topic page so that the debug framework itself isn't re-documented in every task page.

10.12 Algorithm notes

This section can contain extended discussion about an algorithm. Mathematical derivations, figures, algorithm workflow diagrams, and literature citations can all be included in the Algorithm notes section.

Note that this section is the definitive scientific description of an algorithm. Docstrings of methods and functions that (at least partially) implement an algorithm can defer to this section. This design makes it easier for scientific users to understand algorithms without

following method call paths, while allowing method and function docstrings to focus on technical implementation details (such as arguments, returns, exceptions, and so forth).

11 API Reference Documentation Prototypes

In the Science Pipelines documentation project, API references are collections of topics that document, in a highly structured format, how to program against the Python and C++ codebase. Following the docs-as-code pattern, documentation content is written in, and extracted from the codebase itself. We use Numpydoc format for Python API documentation, with a toolchain extended from Astropy's astropy-helpers. For C++, Doxygen inspects the API and code comments to generate an XML file that is processed with breathe into reStructuredText content that is built by Sphinx.

This section describes the prototype implementations of the API documentation infrastructure with two packages: `daf_base`, and `validate_base`. Section 11.1 Prototype Python API reference implementation for validate_base describes the case of `validate_base`, a pure-Python package with extensive documentation written in Numpydoc. Section 11.2 Prototype SWIG-wrapped Python API reference for daf_base describes Python documentation in `daf_base`, where many of the Python APIs are implemented in C++ and wrapped with SWIG. Finally, Section 11.3 Prototype C++ API reference in daf_base describes C++ API documentation in `daf_base` generated with Doxygen and Breathe. This prototype work was performed in tickets DM-7094 and DM-7095.

11.1 Prototype Python API reference implementation for validate_base

`lsst.validate.base` is a useful prototype for Python API reference documentation since it is pure-Python and was originally written with well-formatted Numpydoc docstrings.

11.1.1 Module homepage

In `lsst.validate.base`'s module topic, this reStructuredText generates the Python API reference:

```
Python API reference
====================

.. automodapi:: lsst.validate.base

.. automodapi:: lsst.validate.base.jsonmixin
   :no-inheritance-diagram:

.. automodapi:: lsst.validate.base.datummixin
   :no-inheritance-diagram:
```

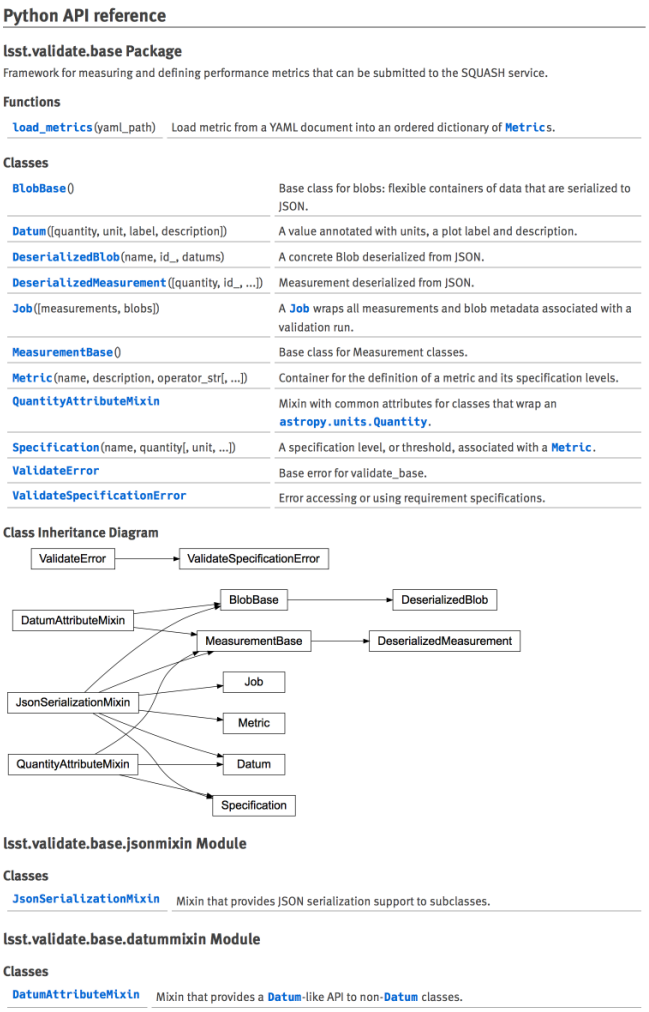
For each Python module namespace exported by `lsst.validate.base`, we include a corresponding `automodapi` directive on the module topic page. `automodapi`, obtained from Astropy's astropy-helpers through the `documenteer.sphinxconfig.stackconfig`, makes Python API references efficient to build since the `automodapi` directive does the following things:

- Collects all functions and classes in a module and generates API references pages for them.
- Builds a table of contents in situ.
- Builds a class inheritance diagram.

To take advantage of `automodapi`'s semi-automated API reference generation, LSST's Python modules need clean namespaces. There are two techniques for achieving this:

1. Modules should only export public APIs, using `__all__`.
2. Sub-package `__init__` files should import the APIs of modules to create a cohesive API organization. In `validate_base`, most APIs are imported into the `__init__.py` module of `lsst.validate.base`. The `lsst.validate.base.jsonmixin` and `lsst.validate.base.datummixin` modules were not imported into `lsst.validate.base` because they are private APIs, yet still need to be documented.

Figure 6 shows the visual layout of a Python API table built by `automodapi`:



Python API reference

lsst.validate.base Package

Framework for measuring and defining performance metrics that can be submitted to the SQUASH service.

Functions

load_metrics(yaml_path)	Load metric from a YAML document into an ordered dictionary of Metrics.

Classes

BlobBase()	Base class for blobs: flexible containers of data that are serialized to JSON.
Datum([quantity, unit, label, description])	A value annotated with units, a plot label and description.
DeserializedBlob(name, id_, datums)	A concrete Blob deserialized from JSON.
DeserializedMeasurement([quantity, id_, ...])	Measurement deserialized from JSON.
Job([measurements, blobs])	A Job wraps all measurements and blob metadata associated with a validation run.
MeasurementBase()	Base class for Measurement classes.
Metric(name, description, operator_str[, ...])	Container for the definition of a metric and its specification levels.
QuantityAttributeMixin	Mixin with common attributes for classes that wrap an astropy.units.Quantity.
Specification(name, quantity[, unit, ...])	A specification level, or threshold, associated with a Metric.
ValidateError	Base error for validate_base.
ValidateSpecificationError	Error accessing or using requirement specifications.

Class Inheritance Diagram

lsst.validate.base.jsonmixin Module

Classes

JsonSerializationMixin	Mixin that provides JSON serialization support to subclasses.

lsst.validate.base.datummixin Module

Classes

DatumAttributeMixin	Mixin that provides a Datum-like API to non-Datum classes.

Figure 6 `lsst.validate.base` API contents on the module topic page.

11.1.2 Python API reference pages

As mentioned, `automodapi` also creates API reference pages. Each function and class is documented on a separate page. Figure 7 shows the reference page for the `lsst.validate.base.Metric` class.

Metric

class lsst.validate.base. **Metric** (*name*, *description*, *operator_str*, *specs=None*, *parameters=None*, *reference_doc=None*, *reference_url=None*, *reference_page=None*) [source]

Bases: lsst.validate.base.jsonmixin.JsonSerializationMixin

Container for the definition of a metric and its specification levels.

Metrics can either be instantiated programatically, or from a metric YAML file with the from_yaml class method.

See also

See the Using metrics and specifications in Python page for usage details.

Parameters:

name : str

Name of the metric (e.g., 'PA1').

description : str

Short description about the metric.

operator_str : str

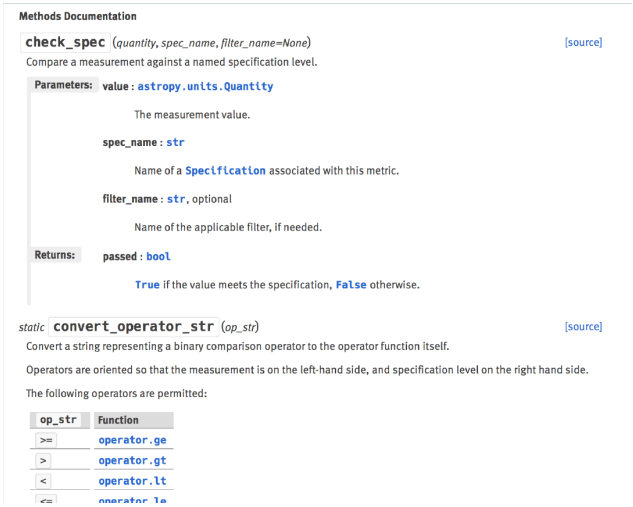
A string, such as '<=', that defines a success test for a measurement (on the left hand side) against the metric specification level (right hand side).

specs : list, optional

A list of Specification objects that define various specification levels for this metric.

parameters : dict, optional

A dict of named Datum values that must be known when measuring a metric.



Methods Documentation

check_spec (*quantity*, *spec_name*, *filter_name=None*) [source]

Compare a measurement against a named specification level.

Parameters:

value : astropy.units.Quantity

The measurement value.

spec_name : str

Name of a Specification associated with this metric.

filter_name : str, optional

Name of the applicable filter, if needed.

Returns:

passed : bool

True if the value meets the specification, False otherwise.

static **convert_operator_str** (*op_str*) [source]

Convert a string representing a binary comparison operator to the operator function itself.

Operators are oriented so that the measurement is on the left-hand side, and specification level on the right hand side.

The following operators are permitted:

op_str	Function
>=	operator.ge
>	operator.gt
<	operator.lt
<=	operator.le

reference_doc : `str`, optional

The document handle that originally defined the metric (e.g., `'LPM-17'`).

reference_url : `str`, optional

The document's URL.

reference_page : `str`, optional

Page where metric in defined in the reference document.

Attributes Summary

`description`	Short description of the metric (`str`).
`json`	`dict` that can be serialized as semantic JSON, compatible with
`name`	Name of the metric (`str`).
`operator`	Binary comparision operator that tests success of a measurement fulfilling a specification of this metric.
`operator_str`	String representation of comparison operator.
`parameters`	`dict` of named `Datum` values that must be known when measuring
`reference`	Documentation reference as human-readable text (`str`, read-only).
`reference_doc`	Name of the document that specifies this metric (`str`).
`reference_page`	Page number in the document that specifies this metric (`int`).
`reference_url`	URL of the document that specifies this metric (`str`).

Methods Summary

`check_spec`(quantity, spec_name[, filter_name])	Compare a measurement against a named specification level.
`convert_operator_str`(op_str)	Convert a string representing a binary comparison operator to the operator function itself.
`from_json`(json_data)	Construct a Metric from a JSON dataset.
`from_yaml`(metric_name[, yaml_doc, ...])	Create a `Metric` instance from a YAML document that defines metrics.
`get_spec`(name[, filter_name])	Get a specification by name and other qualifications.
`get_spec_dependency`(spec_name, dep_name[, ...])	Get the `Datum` of a specification's dependency.
`get_spec_names`([filter_name])	List names of all specification levels defined for this metric; optionally filtering by attributes such as filter name.
`jsonify_dict`(d)	Recursively build JSON-renderable objects on all values in a dict.
`write_json`(filepath)	Write JSON to a file.

Attributes Documentation

`description` = *None*

Short description of the metric (`str`).

`json`

`dict` that can be serialized as semantic JSON, compatible with the SQUASH metric service.

`name` = *None*

Name of the metric (`str`).

`operator`

Binary comparision operator that tests success of a measurement fulfilling a specification of this metric.

Measured value is on left side of comparison and specification level is on right side.

`operator_str`

String representation of comparison operator.

The comparison is oriented with the measurement on the left-hand side and the specification level on the right-hand side.

`parameters` = *{}*

`dict` of named `Datum` values that must be known when measuring a metric.

Parameters can also be accessed as attributes of the metric. Attribute names are the same as key names in `parameters`.

`reference`

Documentation reference as human-readable text (`str`, read-only).

Uses `reference_doc`, `reference_page`, and `reference_url`, as available.

`reference_doc` = *None*

Name of the document that specifies this metric (`str`).

`reference_page` = *None*

Page number in the document that specifies this metric (`int`).

`reference_url` = *None*

URL of the document that specifies this metric (`str`).

`==`	`operator.eq`
`!=`	`operator.ne`

Parameters: **op_str** : `str`

A string representing a binary operator.

Returns: **op_func** : obj

An operator function from the `operator` standard library module.

classmethod `from_json`(*json_data*) [source]

Construct a Metric from a JSON dataset.

Parameters: **json_data** : `dict`

Metric JSON object.

Returns: **metric** : `Metric`

Metric from JSON.

classmethod `from_yaml`(*metric_name*, *yaml_doc=None*, *yaml_path=None*, *resolve_dependencies=True*) [source]

Create a `Metric` instance from a YAML document that defines metrics.

See also

See Defining metrics and specifications in YAML for details on the metric YAML schema.

Parameters: **metric_name** : `str`

Name of the metric (e.g., `'PA1'`).

yaml_doc : `dict`, optional

A full metric YAML document loaded as a `dict`. Use this option to increase performance by eliminating redundant reads of a common metric YAML file. Alternatively, set `yaml_path`.

yaml_path : `str`, optional

The full file path to a metric YAML file. Alternatively, set `yaml_doc`.

resolve_dependencies : `bool`, optional

API users should always set this to `True`. The opposite is used only used internally.

Raises: **RuntimeError**

Raised when neither `yaml_doc` or `yaml_path` are set.

`get_spec`(*name*, *filter_name=None*) [source]

Get a specification by name and other qualifications.

Parameters: **name** : `str`

Name of a specification level (e.g., `'design'`, `'minimum'`, `'stretch'`).

filter_name : `str`, optional

The name of the optical filter to qualify a filter-dependent specification level.

Returns: **spec** : `Specification`

The `Specification` that matches the name and other qualifications.

Raises: **RuntimeError**

If a specification cannot be found.

`get_spec_dependency`(*spec_name*, *dep_name*, *filter_name=None*) [source]

Get the `Datum` of a specification's dependency.

If the dependency is a metric, this method resolves the value of the dependent metric's specification level `spec_name`. In other words, `spec_name` refers to the specification level of both *this* metric and of the dependency metric.

Parameters: **spec_name** : `str`

`Specification` name.

dep_name : `str`

Name of the dependency.

filter_name : `str`, optional

Name of the optical filter, if this metric's specifications are optical filter dependent.

Returns: **datum** : `Datum`

The dependency resolved for the metric's specification.

`get_spec_names`(*filter_name=None*) [source]

List names of all specification levels defined for this metric; optionally filtering by attributes such as filter name.

Parameters: **filter_name** : `str`, optional

Name of the applicable filter, if needed.

Returns: **spec_names** : `list`

Specification names as a list of strings, e.g. `['design', 'minimum', 'stretch']`.

`jsonify_dict`(*d*)

Recursively build JSON-renderable objects on all values in a dict.

Parameters: **d** : `dict`

Dictionary ito convert into a JSON-serializable object. Values are recursively JSON-ified.

Returns: **json_dict** : `dict`

Dictionary that can be serialized to JSON.

Examples

Subclasses can use this method to prepare output in their `json`-method implementation. For example:

```
def json(self):
    return JsonSerializationMixin.jsonify_dict({
    'value': self.value,
    })
```

`write_json`(*filepath*)

Write JSON to a file.

Parameters: **filepath** : `str`

Figure 7 `lsst.validate.base.Metric` class API page, generated by Numpydoc.

Compared to Doxygen's HTML output that builds unified references for all API objects in a module, `automodapi` documents each function and class in separate pages. This choice by Astropy is effective because well-documented classes tend to already have long pages (see lsst.validate.base.Metric class API page, generated by Numpydoc.); mixing several classes on the same page would create confusion about an API object's class membership as a user scrolls.

11.1.3 Numpydoc implementation notes

The means of crafting API reference content for individual classes and functions is already well resolved. `automodapi` uses Numpydoc to extract docstrings from code and build well-formatted reference pages. DM's usage of Numpydoc is already well-documented in the Developer Guide, and approved for use in RFC-214.

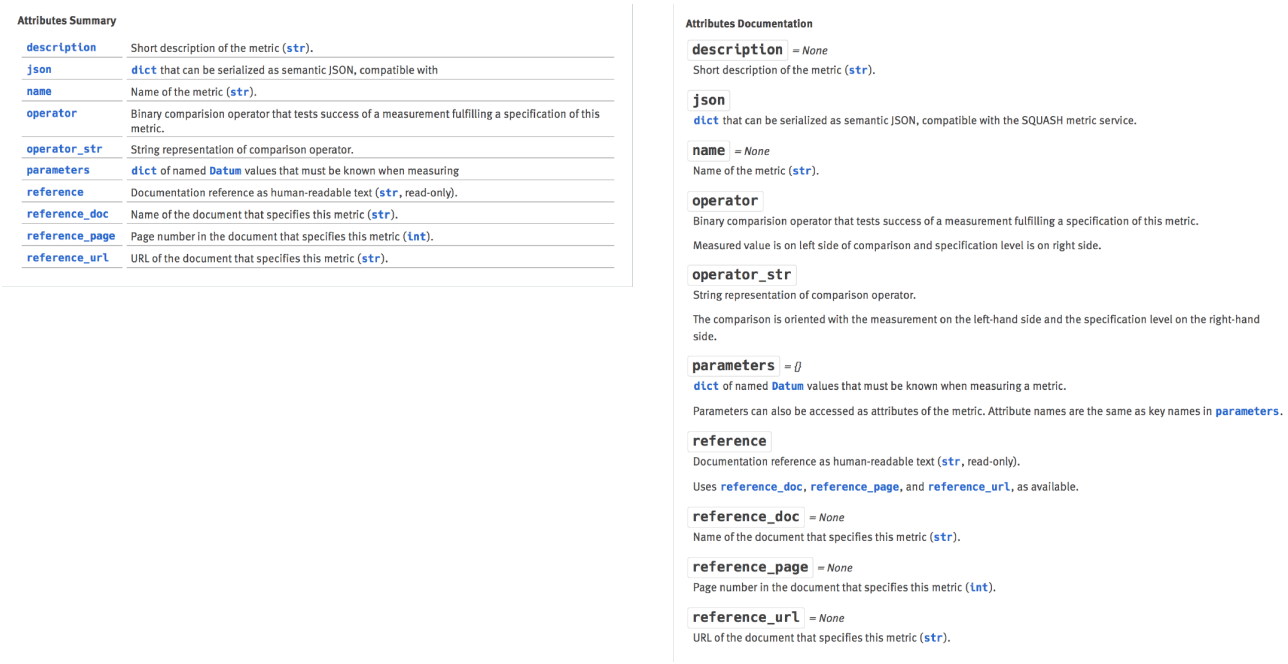
The only unexpected difficulty encountered in the `lsst.validate.base` prototype documentation was with class attribute documentation. Publicly accessible class attributes should be documented, since they form the API along with methods. Numpydoc's documentation recommends that attributes should be documented in a special "Attributes" section of the class docstring. For example:

```
class Example(object):
    """An example class.

    ...

    Attributes
    ----------
    name : `str`
       The instance name.
    """

    def __init__(self, name):
        self.name = name
```

However, this attributes section was separate from another attributes section that documented a class's properties. The correct format, used by Astropy, is to associate a docstring with each attribute where it is declared:

```
class Example(object):
    """An example class.

    ...
    """

    hello = None
    """The instance name (`str`)."""

    def __init__(self, name):
        self.name = name
```

This documentation approach may alter some classes by requiring attribute declarations at the class scope, rather than only during `__init__`. However, the outcome is highly useful documentation where all attributes, even class properties are documented together, as in Figure 8.



Attributes Summary

description	Short description of the metric (str).
json	dict that can be serialized as semantic JSON, compatible with
name	Name of the metric (str).
operator	Binary comparision operator that tests success of a measurement fulfilling a specification of this metric.
operator_str	String representation of comparison operator.
parameters	dict of named Datum values that must be known when measuring
reference	Documentation reference as human-readable text (str, read-only).
reference_doc	Name of the document that specifies this metric (str).
reference_page	Page number in the document that specifies this metric (int).
reference_url	URL of the document that specifies this metric (str).

Figure 8 Attributes table (left) and expanded documentation (right) in `lsst.validate.base.Metric`. Note that regular attributes, like `name`, are documented alongside attributes implemented as properties, like `json`. See also Figure 6 where attribute documentation is shown in the context of the full `lsst.validate.base.Metric` reference page.

Another lesson learned from the `lsst.validate.base` prototype is that *module docstrings* should not be comprehensive. In the topic framework, modules are better documented in module topic pages, rather than in code. If anything, the module docstring

may contain a one-sentence summary of the module's functionality. This summary appears as a subtitle of the module name in `automodapi` output. For example, in Figure 6 the module docstring reads: "Framework for measuring and defining performance metrics that can be submitted to the SQUASH service."

In summary, the Python API reference infrastructure created by the Numpy (Numpydoc) and Astropy (`automodapi`) projects is eminently usable for the Science Pipelines documentation without any modifications.

11.2 Prototype SWIG-wrapped Python API reference for daf_base

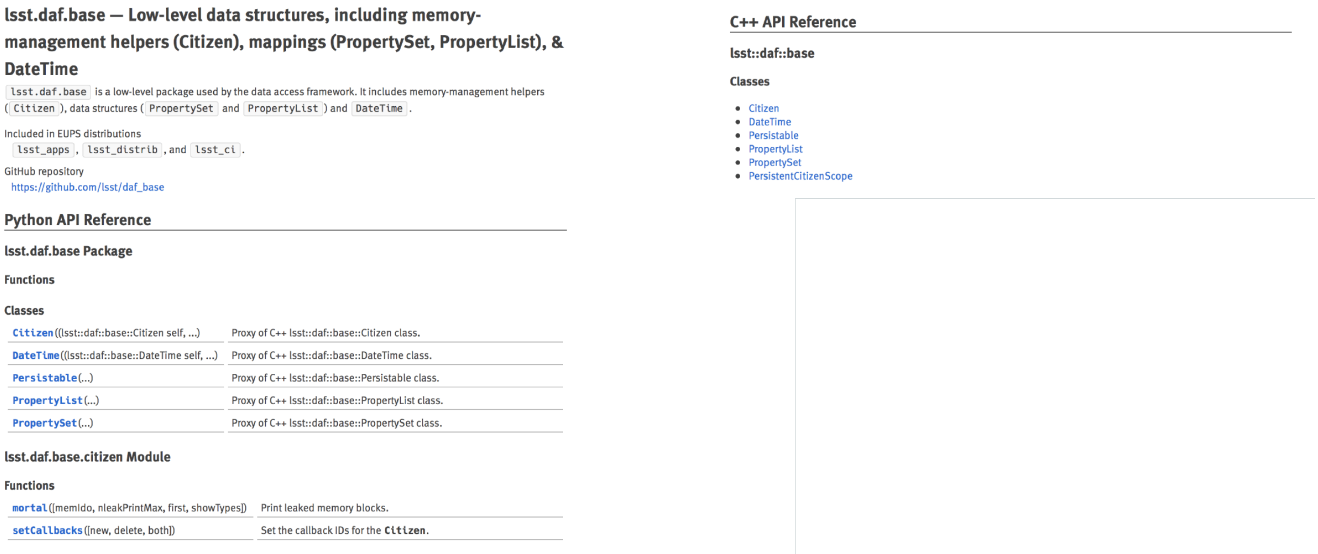
The Python APIs of `lsst.daf.base` are more difficult to generate documentation for since they are not implemented in Python, but rather wrapped with SWIG. This means that the API will generally be non idiomatic, and docstrings do not conform to Numpydoc.

In the `lsst.daf.base` prototype (DM-7095) the wrapped `lsst.daf.base` and native `lsst.daf.base.citizen` modules are documented with this reStructuredText:

```
Python API reference
====================

.. automodapi:: lsst.daf.base
   :no-inheritance-diagram:
   :skip: long, Citizen_census, Citizen_getNextMemId,
Citizen_hasBeenCorrupted, Citizen_init, Citizen_setCorruptionCallback,
Citizen_setDeleteCallback, Citizen_setDeleteCallbackId,
Citizen_setNewCallback, Citizen_setNewCallbackId, Citizen_swigregister,
DateTime_initializeLeapSeconds, DateTime_now, DateTime_swigregister,
Persistable_swigregister, PropertyList_cast, PropertyList_swigConvert,
PropertyList_swigregister, PropertySet_swigConvert,
PropertySet_swigregister, SwigPyIterator_swigregister,
VectorBool_swigregister, VectorDateTime_swigregister,
VectorDouble_swigregister, VectorFloat_swigregister,
VectorInt_swigregister, VectorLongLong_swigregister,
VectorLong_swigregister, VectorShort_swigregister,
VectorString_swigregister, endl, ends, flush, ios_base_swigregister,
ios_base_sync_with_stdio, ios_base_xalloc, ios_swigregister,
iostream_swigregister, istream_swigregister, ostream_swigregister,
type_info_swigregister, vectorCitizen_swigregister, SwigPyIterator,
VectorBool, VectorDateTime, VectorDouble, VectorFloat, VectorInt,
VectorLong, VectorLongLong, VectorShort, VectorString, ios, ios_base,
iostream, istream, ostream, type_info, vectorCitizen

.. automodapi:: lsst.daf.base.citizen
```

Note the extensive curation of the Python namespace required for the wrapped `lsst.daf.base` API. SWIG clutters the Python namespace, making it difficult to automatically identify relevant APIs to document.

Figure 9 shows the `lsst.daf.base` API reference contents on its module page.



lsst.daf.base — Low-level data structures, including memory-management helpers (Citizen), mappings (PropertySet, PropertyList), & DateTime

`lsst.daf.base` is a low-level package used by the data access framework. It includes memory-management helpers (`Citizen`), data structures (`PropertySet` and `PropertyList`) and `DateTime` .

Included in EUPS distributions
`lsst_apps` , `lsst_distrib` , and `lsst_ci` .

GitHub repository
https://github.com/lsst/daf_base

Python API Reference

lsst.daf.base Package

Functions

Classes

`Citizen`((lsst::daf::base::Citizen self, ...)	Proxy of C++ lsst::daf::base::Citizen class.
`DateTime`((lsst::daf::base::DateTime self, ...)	Proxy of C++ lsst::daf::base::DateTime class.
`Persistable`(...)	Proxy of C++ lsst::daf::base::Persistable class.
`PropertyList`(...)	Proxy of C++ lsst::daf::base::PropertyList class.
`PropertySet`(...)	Proxy of C++ lsst::daf::base::PropertySet class.

lsst.daf.base.citizen Module

Functions

`mortal`([memIdo, nleakPrintMax, first, showTypes])	Print leaked memory blocks.
`setCallbacks`([new, delete, both])	Set the callback IDs for the `Citizen`.

C++ API Reference

lsst::daf::base

Classes

- Citizen
- DateTime
- Persistable
- PropertyList
- PropertySet
- PersistentCitizenScope

Figure 9 Module page for `lsst.daf.base` in the DM-7094/DM-7095 prototyping. The Python API reference section is built with Numpydoc's `automodapi` (11.2 Prototype SWIG-wrapped Python API reference for daf_base), while the C++ section is assembled from a `toctree` of manually-built pages containing `doxygenclass` directives (11.3 Prototype C++ API reference in daf_base).

The generated API reference page for a SWIG-wrapped Python class (`lsst.daf.base.Citizen`) is shown in Figure 10. Compared to a Python API documented in Numpydoc, the docstrings generated by SWIG are not useful. Additional work, outside the scope of this document, is needed to establish how Python APIs implemented from C++ should be documented.

Meanwhile, note that Numpydoc still renders poorly-formed docstrings (albeit, with Sphinx warnings). This will be useful during early implementation of the Science Pipelines documentation site.

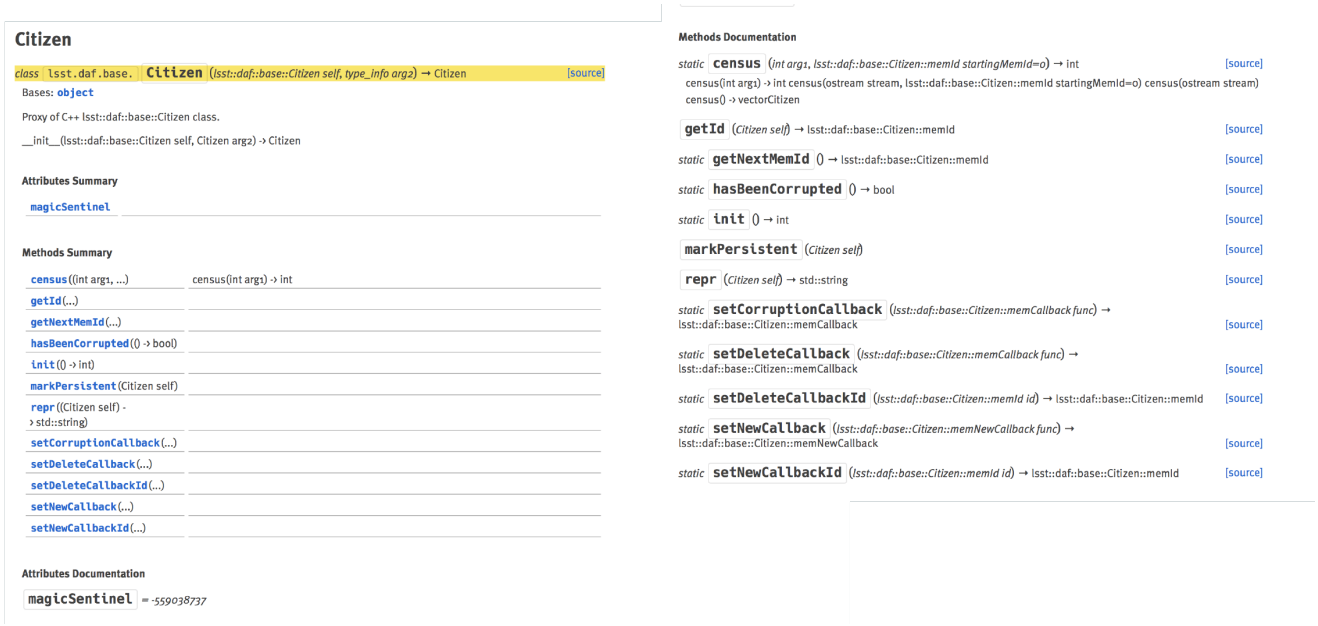


Figure 10 `lsst.daf.base.Citizen` *Python API reference page, generated from Numpydoc, of the SWIG-wrapped* `lsst::daf::base::Citizen` *C++ class.*

11.3 Prototype C++ API reference in daf_base

This section explores the prototype API reference documentation for the C++ package `lsst::daf::base`. In this case, the C++ source and Doxygen-formatted comments are processed by Doxygen, which yields XML files describing the API (XML is a Doxygen output generated by `sconsUtils` in addition to HTML). When the Sphinx project is built, Breathe uses these XML files to generate reStructuredText content. Mechanisms for configuring Breathe to find the appropriate XML are already included in `documenteer.sphinxconfig.stackconf`.

Like `automodapi`, Breathe provides a doxygennamespace directive that generates documentation for an entire C++ namespace (like `lsst::daf::base`).

```
C++ API Reference
=================

lsst::daf::base
---------------

Classes
^^^^^^^

.. doxygennamespace:: lsst::daf::base
   :project: daf_base
```

Unlike `automodapi`, though, this directive inserts all API documentation for the namespace in situ, rather than creating and linking to API reference pages for individual API objects.

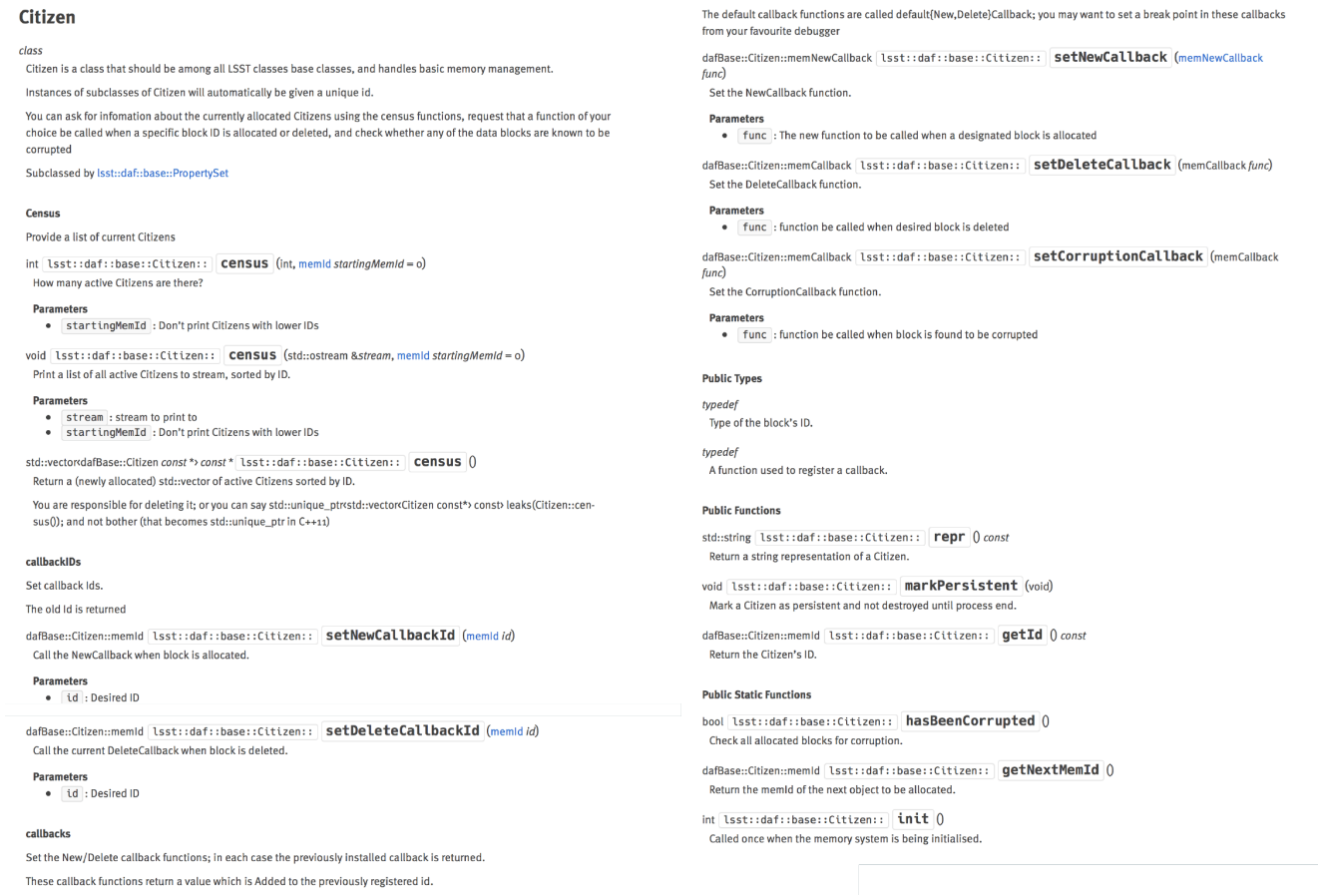
To emulate `automodapi` with standard Breathe directives, we first created a `toctree` that linked to manually-built API reference pages for each C++ class:

```
C++ API Reference
=================

lsst::daf::base
---------------

Classes
^^^^^^^

.. toctree::

   cpp/lsst_daf_base_Citizen
   cpp/lsst_daf_base_DateTime
   cpp/lsst_daf_base_Persistable
   cpp/lsst_daf_base_PropertyList
   cpp/lsst_daf_base_PropertySet
   cpp/lsst_daf_base_PersistentCitizenScope
```

The output of this `toctree` is shown in Figure 9 (right).

Each manually built class reference page uses Breathe's `doxygen` class directive. An example for `lsst::daf::base::Citizen`:

```
#######
Citizen
#######

.. doxygenclass:: lsst::daf::base::Citizen
   :project: daf_base
   :members:
```



Citizen

class

Citizen is a class that should be among all LSST classes base classes, and handles basic memory management.

Instances of subclasses of Citizen will automatically be given a unique id.

You can ask for infomation about the currently allocated Citizens using the census functions, request that a function of your choice be called when a specific block ID is allocated or deleted, and check whether any of the data blocks are known to be corrupted

Subclassed by lsst::daf::base::PropertySet

Census

Provide a list of current Citizens

int `lsst::daf::base::Citizen::` **census** (int, memId *startingMemId* = 0)

How many active Citizens are there?

Parameters
- `startingMemId` : Don't print Citizens with lower IDs

void `lsst::daf::base::Citizen::` **census** (std::ostream &*stream*, memId *startingMemId* = 0)

Print a list of all active Citizens to stream, sorted by ID.

Parameters
- `stream` : stream to print to
- `startingMemId` : Don't print Citizens with lower IDs

std::vector<dafBase::Citizen *const* *> *const* * `lsst::daf::base::Citizen::` **census** ()

Return a (newly allocated) std::vector of active Citizens sorted by ID.

You are responsible for deleting it; or you can say std::unique_ptr<std::vector<Citizen const*> const> leaks(Citizen::census()); and not bother (that becomes std::unique_ptr in C++11)

callbackIDs

Set callback Ids.

The old Id is returned

dafBase::Citizen::memId `lsst::daf::base::Citizen::` **setNewCallbackId** (memId *id*)

Call the NewCallback when block is allocated.

Parameters
- `id` : Desired ID

dafBase::Citizen::memId `lsst::daf::base::Citizen::` **setDeleteCallbackId** (memId *id*)

Call the current DeleteCallback when block is deleted.

Parameters
- `id` : Desired ID

callbacks

Set the New/Delete callback functions; in each case the previously installed callback is returned.

These callback functions return a value which is Added to the previously registered id.

The default callback functions are called default{New,Delete}Callback; you may want to set a break point in these callbacks from your favourite debugger

dafBase::Citizen::memNewCallback `lsst::daf::base::Citizen::` **setNewCallback** (memNewCallback *func*)

Set the NewCallback function.

Parameters
- `func` : The new function to be called when a designated block is allocated

dafBase::Citizen::memCallback `lsst::daf::base::Citizen::` **setDeleteCallback** (memCallback *func*)

Set the DeleteCallback function.

Parameters
- `func` : function be called when desired block is deleted

dafBase::Citizen::memCallback `lsst::daf::base::Citizen::` **setCorruptionCallback** (memCallback *func*)

Set the CorruptionCallback function.

Parameters
- `func` : function be called when block is found to be corrupted

Public Types

typedef

Type of the block's ID.

typedef

A function used to register a callback.

Public Functions

std::string `lsst::daf::base::Citizen::` **repr** () *const*

Return a string representation of a Citizen.

void `lsst::daf::base::Citizen::` **markPersistent** (void)

Mark a Citizen as persistent and not destroyed until process end.

dafBase::Citizen::memId `lsst::daf::base::Citizen::` **getId** () *const*

Return the Citizen's ID.

Public Static Functions

bool `lsst::daf::base::Citizen::` **hasBeenCorrupted** ()

Check all allocated blocks for corruption.

dafBase::Citizen::memId `lsst::daf::base::Citizen::` **getNextMemId** ()

Return the memId of the next object to be allocated.

int `lsst::daf::base::Citizen::` **init** ()

Called once when the memory system is being initialised.

Figure 11 `lsst::daf::base::Citizen` *API reference page, generated from breath's* `doxygenclass` *directive.*

As Figure 11 shows, C++ class documentation rendered this way is still not as useful as the pure-Python documentation. One reason is that the Breathe output is not typeset as strongly as Numpydoc output is. Better CSS support could help with this. Second, the comment strings written for Figure 11 itself are not as comprehensive as those for `lsst.validate.base` Python classes. Better C++ documentation standards will help improve content quality (see DM-7891 for an effort to address this).

11.4 API reference generation conclusions

In review, the Science Pipelines documentation has three distinct types of API references: pure-Python, wrapped Python, and C++ APIs. Numpydoc and `automodapi` are excellent off-the-shelf solutions for generating documentation for Python APIs. The latter two modes require additional engineering.

In principle, Doxygen and Breathe are a good toolchain for generating C++ API references. The following will improve C++ documentation:

- Improved C++ Doxygen documentation standards.
- Better CSS formatting of `doxygenclass` directive output.
- Development of a custom directive that emulates `automodapi` by automatically scraping a C++ namespace and generating individual documentation, but still uses `doxygenclass`.

Generating reference documentation for Python APIs implemented in C++ will be the most difficult challenge. We will address this separately, in conjunction with LSST's migration from SWIG to Pybind11.