



LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)  
**QA Strategy Working Group Report**

Bellm, E.C., Chiang, H.-F., Fausti, A., Krughoff, K.S., MacArthur,  
L.A., Morton, T.D., Swinbank, J.D and Roby, T.

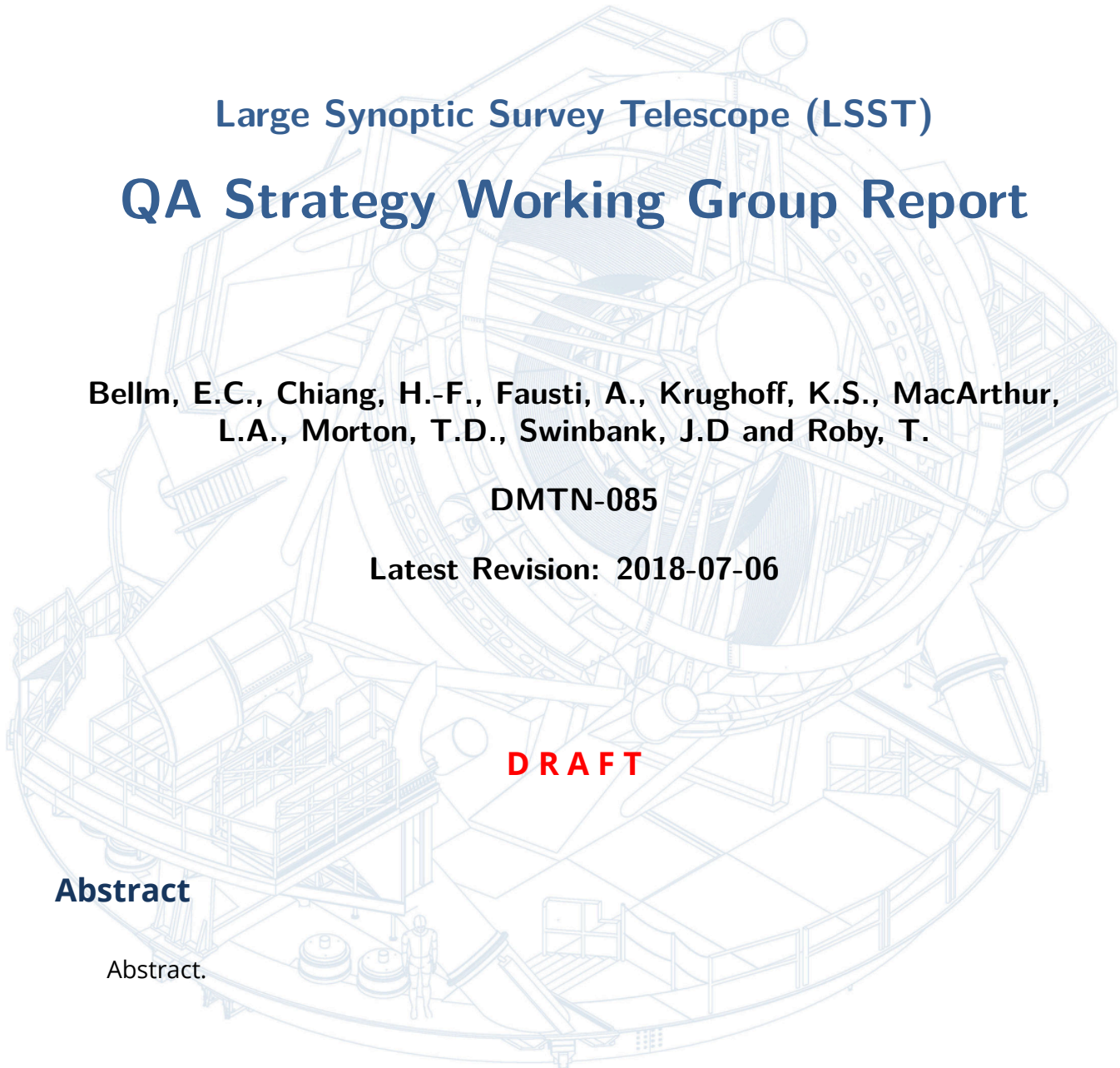
DMTN-085

Latest Revision: 2018-07-06

**DRAFT**

**Abstract**

Abstract.





## Change Record

Version	Date	Description	Owner name
8375aa1	2018-07-06	Unreleased draft.	Bellm et al.

Draft

## Contents

<b>1 Introduction</b>	<b>1</b>
<b>2 Approach to the Problem</b>	<b>1</b>
2.1 Pipeline debugging . . . . .	2
2.2 Drill down . . . . .	2
2.3 Datasets and test infrastructure . . . . .	3
<b>3 Design sketch</b>	<b>3</b>
3.1 Pipeline debugging . . . . .	3
3.2 Drill down . . . . .	4
3.3 Datasets and test infrastructure . . . . .	4
<b>4 Core components</b>	<b>4</b>
4.1 Updated pipeline debugging system . . . . .	4
4.2 Logging . . . . .	5
4.3 Capability for developers to run pipelines at scale . . . . .	5
4.4 Guidance on visualization . . . . .	6
4.5 Image viewer . . . . .	7
4.6 Catalog visualization tools . . . . .	8
4.7 Provenance . . . . .	8
4.8 Documentation content updates . . . . .	9
4.9 Testing for documentation . . . . .	10
4.10 CI system updates . . . . .	10
4.11 Metrics Dashboard / SquaSH . . . . .	11
4.12 Standard format dataset package . . . . .	11
4.13 Standard test package design . . . . .	13
<b>A Glossary</b>	<b>14</b>

# QA Strategy Working Group Report

## 1 Introduction

Assignee

John

This report constitutes the primary artefact produced by the DM QA Strategy Working Group (QAWG), addressing its charge as defined in LDM-622.

## 2 Approach to the Problem

Assignee

John

The QAWG addressed its charge by sub-dividing the problem space into three separate areas:

- Addressing the needs of developers writing and debugging algorithms on the small scale;
- Developing tooling to address the *drill down* use case;
- Providing the infrastructure needed to support automatic testing and verification.

Each of these areas were assigned to a separate sub-group within the WG for brainstorming and developing approaches, with each sub-group regularly reporting progress to overall working group meetings.

When each sub-group had developed a strong concept for the tooling needed to address their particular part of the charge, the whole working group reviewed each design in detail, identifying and developing specifications for common components or activities that enable one or more of the designs.

In §§2.1, 2.2 and 2.3, we provide details about the charge provided to each sub-group. In §§3.1, 3.2 and we provide details about the charge provided to each

## 2.1 Pipeline debugging

Assignee

John

What tools do we need to help pipeline developers with their every-day work? Specifically:

- How do you go about debugging a Task that is crashing?
- Is `lsstDebug` adequate?
- Do we need an `afwFigure`, for generating plots, to go alongside `afwDisplay`, for showing images?
- What additional capabilities are needed for developers running and debugging at scale, e.g. log collection, identification of failed jobs, etc.
- What's needed from an image viewer for pipeline developers? Is DS9 or Firefly adequate? Is there value to the `afwDisplay` abstraction layer, or does it simply make it harder for us to use Firefly's advanced features?
- How do we view images which don't fit in memory on a single node?
- How do we handle fake sources? Is this a provenance issue?

## 2.2 Drill down

Assignee

John

How can we provide developers and testers with the ability to “drill down” from high level aggregated metrics to explore the source data and intermediate data products that contributed to them? Specifically:

- What sort of metrics should be extracted from running pipelines<sup>1</sup>?
- How can those metrics be displayed on a dashboard? Is a simple time-series adequate, or do we need other types of plotting?

<sup>1</sup>Scalars, vectors, spatially binned quantities, etc.

- By what mechanism can the user drill-down from those aggregated metrics to identify the sources of problems? Do they click through pre-generated plots, or jump straight into a notebook environment?
- Assuming the user ends up in an interactive environment, what are its capabilities?
- What do the above tell us about the data products that pipelines need to persist (both in terms of metrics that are posted to SQuaSH, and regular pipeline outputs, Parquet tables, HDF5 files, etc)?

## 2.3 Datasets and test infrastructure

Assignee

John

What infrastructure must we make available to enable testing and verification of the DM system? Specifically:

- Are any changes needed to the way that DM currently handles unit testing?
- How are datasets made available to developers? Git LFS repositories? GPFS?
- What is the appropriate cadence for running small/medium/large scale integration tests and reprocessing of known data?
- How is the system for tracking metrics managed? — how are the metric calculation jobs run? By whom? How often?
- How run-time performance of the science algorithms be tracked?

## 3 Design sketch

### 3.1 Pipeline debugging

Assignee

John

## 3.2 Drill down

Assignee

Tim

## 3.3 Datasets and test infrastructure

Assignee

John

# 4 Core components

## 4.1 Updated pipeline debugging system

Assignee

Simon

The debugging system must be both reasonably powerful and easy to use. It should also be obvious from help/doc strings how to turn on debugging. There is a tradeoff between granularity and usability.

The recommendation is that the debugging system be simplified to be configurable at the task level. Debugging is turned on via a config parameter. This allows for single sub-tasks to turn on debugging independently. In practice this means that statements like `if lsstDebug:` would turn into `if self.config.debug`.

This brings up the obvious issue that free functions/non-task methods called in the run method of a task do not necessarily have the debugging flag passed into them. It becomes the responsibility of the implementer to pass `self.config.debug` into methods that have debug functionality.

Derived from §3.1.

i.e. redesigned `lsstDebug`.

## 4.2 Logging

Assignee

Simon

Logging is an important aspect of running large data processing. It is also integral to quality assessment as the logging information provides important contextual information when inspecting data for quality issues. Specifically, log messages presenting information about the processing: e.g. PSF width, number of stars used in a model fit, can indicate problems with the algorithmic behavior or input data. Logging also provides information about potential causes for unexpected termination including exit codes and exceptions.

While logs can be straightforward to parse when only a small number of concurrent processes are being used, they quickly become harder to understand as the number of processes increases.

The logging component has the following attributes:

- Configurable for logging granularity. For example, INFO, WARN, ERROR.
- Trivial to retrieve time ordered logs for a thread.
- Possible to retrieve logs, exit status, or exceptions for threads ending in an unexpected state.
- Possible to retrieve enough information to rerun the data units that were unprocessed because of processes ending in unexpected state. KSK: perhaps this is solved by the provenance system.
- Searchable per process for regexp and timestamp.

Derived from §3.1.

## 4.3 Capability for developers to run pipelines at scale

Assignee

Simon



I believe we agreed that if the logging system and provenance system are sufficient to meet recommendations, this aspect is essentially a solved problem.

I do think that it also depends on the orchestration layer, but I don't think that can be in the scope of this group.

Derived from §3.1.

#### 4.4 Guidance on visualization

Assignee

Simon

We decided that this

section should first be seeded with contexts for visualization. We can then go through and give specific guidance.

Contexts:

- Notebooks in-line: matplotlib-ish
- Dashboard like visualization: health status (grafana?)
- Pipelines debugging visualization: both persisted PNGs and pop up vis
- Purpose built QA visualization: Like the multisky plot from qa\_explorer
- Exploratory: Single plots with interactivity, including zoom and pan
- Linked plots: Interaction in one frame causes chnge of state in another plot
- Image visualization: (covered elsewhere)
- Publication plotting
- Static plots for reports
- Plotting from a non-notebook interactive setting
- Rendering of cached time series: SQuaSH

Derived from §3.1.

We're requesting a set of guidelines for developers here, not a new framework — but that's still a concrete deliverable (it's just documentation, rather than code). We might suggest that these guidelines be developed by a new WG, per Simon's suggestion<sup>2</sup>.

## 4.5 Image viewer

Assignee

Trey

Derived from §3.1.

As of 2018-06-12 we haven't converged on a solid recommendation here.

Key considerations:

- Firefly is the annointed solution being provided by DM to external stakeholders (commissioning, operations, etc). It feels right to everybody that we should be dogfooding it, and also benefitting from development being carried out for those stakeholders.
- Currently, Firefly is unappealing to developers (primarily, I think, because of slowness of user interface, and perhaps also due to installation issues). Can we resolve these issues?
- We'd want to support visualization in a number of different environments, for e.g.:
  - Inside a Jupyter notebook;
  - As a standalone tool, à la DS9;
  - Embedded in a dashboard, à la JS9, Aladin-Lite, etc.
- Do we lose flexibility by mandating the use of a backend-agnostic API (`afwDisplay`) rather than going "all-in" on e.g. a custom Firefly interface?
- We'll need to do full focal plane visualization, which none(?) of the current tools support well.

Options include:

<sup>2</sup><https://confluence.lsstcorp.org/display/DM/Pipeline+Debugging+Design>

- Do nothing; continue as we are, which means most people will use DS9 and a few will drift to Firefly as commissioning ramps up.
- Issue some sort of edict that pipelines developers have to use Firefly.
- Encourage the use of some other tool (Ginga?) instead of or as well as some of the above.
- Probably others.

Sounds like we need somebody from the QAWG to actually write some requirements — or a wishlist set of features we want — here.

#### 4.6 Catalog visualization tools

Assignee

Lauren

Derived from §3.1.

For visualizing bigger-than-memory catalogs. May include e.g. the capability to spin up Dask clusters on demand, combined with Holoviews/Datashader/whatever. Somebody who knows about this stuff needs to write a summary...

#### 4.7 Provenance

Assignee

Hsin-Fang

The QAWG recognizes the importance of provenance and the implementation of the provenance system will impact QA work significantly. We recommend high priority to finalize the design and implementation of the provenance system.

The QAWG believes that the requirements on provenance tracking are adequate as described in the Data Management Middleware Requirements (LDM-556) and Data Management Data Backbone Services Requirements (LDM-635). QA use cases provide no further requirements.

With the current design of the Gen 3 Middleware, each dataset can be linked to provenance information such as input datasets, pipeline definition, configurations, and software version

(e.g. DMS-MWST-REQ-0024 and DMS-MWBT-REQ-0096 in LDM-556). QC metric values will be Butler datasets so they will be associated with specific data IDs, and their provenance can be traced like other datasets. In the Gen 3 Middleware design, Butler/SuperTask framework is responsible of producing provenance. For production runs, the Data Backbone Services may store additional provenance, for example from the Batch Production Services, besides the Butler/SuperTask generated provenance. In the QA use cases, the primary source of provenance will be the Gen3 Butler/SuperTask generated provenance.

Regarding provenance of the database records, the QAWG does not place a strong requirement on per-source provenance. The current design is that each database record in the production database is either ingested from a file, of which the full provenance is traced, or from an uniquely identifiable execution. This design does not directly provide detailed per-source provenance, such as what exact input images actually contribute to the measurement of a particular source, but the full inputs that can contribute to the source. The provenance tracking of synthetic sources is also unclear. The QAWG thinks most low-level information can be uncovered through the drill down system (§3.2). Diagnostic information can also be computed in the pipeline codes and stored as additional columns. We recommend the tooling to investigate the composition of coadds be made in the drill down system and does not put this requirement in the provenance system.

The QAWG notes that some high level aggregate data products that are derived from provenance data can be useful in QA work. For example, the number of images that contribute to each coadd patch can be obtained from provenance data. However, the QAWG are not immediately convinced that it's worth spending significant time investigating what aggregate products need to be considered.

Derived from §3.1.

This section should note:

- That provenance is an immediate issue impacting QA work, so a solution is a priority;
- Some requirements as to the granularity at which provenance tracking is necessary for QA.

## 4.8 Documentation content updates

Derived from §3.3.

**Assignee**

John

- Clearer guidance on unit tests.
- Clearer guidance on code review, with requirements for test coverage etc.

#### 4.9 Testing for documentation

Derived from §3.3.

**Assignee**

John

- Examples.

#### 4.10 CI system updates

Derived from §3.3.

**Assignee**

John

- Test coverage.
- Tighter control of the environment.
- Better notifications.
- Better descriptions of which jobs do what.
- Clear description of what Developers are required to do before merging to master (see also §4.8).

## 4.11 Metrics Dashboard / SQuaSH

Derived from §3.3.

Assignee
Angelo

Broadly as current SQuaSH, but to track:

- Code execution time.
- Test coverage?
- Notifications of regressions.

## 4.12 Standard format dataset package

Derived from §3.3.

Assignee
Hsin-Fang

The key considerations and motivations to standardize test datasets in the construction phase include:

- Developers would like low-friction access to test datasets. A central location where developers can look up what has been curated is desired.
- Developers would like datasets at multiple scales and representative of various data quality and observing conditions. The properties of these datasets much be well understood.
- Besides raw (unprocessed) data, intermediate and final data products from various stages of pipeline processing are wanted. They facilitate testing of algorithms which are only relevant to later parts of the pipeline or analysis codes without the need of regenerating the products.
- Datasets require continuing maintenance. Data products based on a recent software release are usually wanted.

The standard format of a dataset package is a ready-to-use Butler repository and follows the format of a Butler repository as defined in its corresponding obs package. The format is configurable by design, however, it is tied to the codes in the stack, so can change from a software stack version to another. Besides implementations in the obs packages and Butler, other evolution in the software stack, such as handling of calibration data and reference catalog, can also make a once-working repository incompatible. Therefore, maintenance is occasionally needed to ensure the usability of a dataset package. **The QAWG recommends having a per-dataset product owner.**<sup>3</sup> Product owners are responsible for ensuring that the content and use cases of the datasets are well described and compatible with recent stack versions. The owner of a dataset can typically be the team with immediate use cases and knowledge of its camera package.

The Obs Pkg WG (RFC-393) is charged to re-design and refactor the obs packages for maintainability and extensibility. We suggest the Obs Pkg WG take into considerations in their design to mitigate the close tie between a Butler repository and its obs package implementations, as well as adopt a common structure across different cameras when possible. After the refactoring, the obs packages shall rarely change so the dataset format will be more stable. The QAWG recommends prioritise the Obs Pkg WG.

In some cases, a dataset package may contain additional data that are not tenable in the format of a Butler repository. We recommend following the format as described in DM Developer Guide Common Dataset Organization and Policy<sup>4</sup> and update the policy as needed.

As for the storage of test datasets, we consider any test dataset package being either small or large, based on its size and use cases. The QAWG's recommendations are as follows.

- **Storage of small datasets**

We consider small datasets as those smaller than around 100 GB and comfortably operatable as a Git LFS repository. They are carefully selected to meet specific use cases. The use cases of small datasets include

- as input test data in CI jobs in the DM Jenkins system (§4.10);

<sup>3</sup>At the time of writing, our test datasets include the following: afwdata, ap\_verify\_hits2015, testdata\_cfht, testdata\_deblender, testdata\_decam, testdata\_jointcal, testdata\_lsstSim, testdata\_subaru, qserv\_testdata, validation\_data\_cfht, validation\_data\_decam, validation\_data\_hsc, /datasets/auxTel, /datasets/comCam, /datasets/c-tio0m9, /datasets/lsstCam, /datasets/decam /datasets/des\_sn, /datasets/hsc, /datasets/refcats, /datasets/sdss, /datasets/gapon

<sup>4</sup><https://developer.lsst.io/services/datasets.html>

- as example data in documentations, demos, and tutorials.

To meet their needs, we recommend them

- packaged as EUPS products;
- made publicly available on GitHub;
- stored as Git LFS repositories as needed;
- tagged their versions with the DM software releases;
- documented clearly its use cases and named product owner for each dataset;

- **Storage of large datasets**

We consider large datasets as those larger than around 100 GB and hence transferring over network takes longer than an hour typically. They can contain edge cases that have not been identified to form specific small test datasets, or for use cases in which data volume is important. We recommend them

- made available on LSST development machines (currently on GPFS);
- usable and shared by team members;
- protected under a disaster recovery policy;
- documented clearly its use cases and named product owner for each dataset;

### 4.13 Standard test package design

Derived from §3.3.

Assignee

Hsin-Fang

Currently, automatic continuous integration tests are performed via multipla packages under two designs: (1) Scons-based execution, including `ci_hsc` and `ci_ctio0m9`, and (2) exeuction through shell scripts in `validate_drp`. Both `ci_hsc` and `validate_drp` are run in Jenkins and triggered by timers every night (§4.10).

It's QAWG's understanding that the `validate_drp` scripts will eventually replace `ci_hsc` and `ci_ctio0m9`, and a set of test scripts will be run in a meta-package named `lsst_ci`. However, at the time of writing, the `validate_drp` scripts test only the single frame processing step, while



ci\_hsc exercises almost the entire end-to-end DRP pipelines. There has not been sufficient resources in implementing further processing in `validate_drp`. Similarly, the `lsst_dm_stack_demo` repository should be converted into an EUPS product and a test script added to `lsst_ci` for execution (DM-14806).

The QAWG recommends priority to unify the CI test package design and finish the transition to `validate_drp`. If such effort cannot be allocated, documentations should be added to clearly describe the status quo, and recommendations for developers during the transition should be added to the Developer Guide (similar to §4.8 and §4.10). Before `validate_drp` can replace `ci_hsc` and `ci_ctio0m9`, the packages should be maintained.

Should address the union of `lsst_dm_stack_demo`, `ci_hsc`, `validate_drp` use cases.

## A Glossary

**aggregate metric** An aggregation of multiple point metrics. For example, the overall photometric repeatability for a particular tract given multiple observations of each star.

**aggregation** A single result—e.g., a metric value—computed from a collection of input values. For example, we can sum or average a metric computed over patches to produce an aggregate metric at tract level.

**dashboard** A visual display of the most important information needed to achieve one or more objectives, consolidated and arranged on a single screen so that the information can be monitored at a glance (Few, 2013).

**drill down** Move from a higher level aggregation of data to its inputs. For example, given data describing a tract, we might drill down to constituent patches and then to objects; given a visit, we might drill down to CCD and then source. In the context of this document, it refers to the act of identifying an issue in a high-level summary of the data (e.g. an aberrant metric value) and interactively investigating its inputs to find the source of the problem.

**GPFS** IBM's General Parallel File System; now known as IBM Spectrum Scale. In DM use, this is taken to mean bulk data storage provided through a POSIX filesystem interface at the LSST Data Facility.

**KPM** Key Performance Metric.

**metric** We follow the SQR-019 definition of a metric as a measurable quantities which may be tracked. A metric has a name, description, unit, references, and tags (which are used for grouping). A metric is a scalar by definition. We consider multiple types of metric in this document; see aggregate metric, model metric, point metric.

**metric value** The result of computing a particular metric on some given data. Note that we *compute*, rather than measure, metric values.

**model metric** A metric describing a model related to the data. For example, the coefficients of a 2D polynomial fit to the background of a single CCD exposure.

**monitoring** The process of collecting, storing, aggregating and visualizing metrics.

**point metric** A metric that is associated with a single entry in a catalog. Examples include the shape of a source, the standard deviation of the flux of an object detected on a coadd, the flux of a source detected on a difference image.

**QAWG** QA Strategy Working Group.

**releaseable product** A software package or other component of the DM system which is expected to be included in the next tagged release of the system. At time of writing, this implies inclusion in a standard top-level package (e.g. `lsst_distrib`), but we note that future changes to the release procedure may render that definition obsolete.

**SQuaSH** Science Quality Analysis Harness; SQR-009; <https://squash.lsst.codes>.

**tidy data** Tidy datasets have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table (Wickham, 2014).

## References

[LDM-556], Dubois-Felsmann, G., Jenness, T., Bosch, J., et al., 2018, *Data Management Middleware Requirements*, LDM-556, URL <https://ls.st/LDM-556>

[SQR-009], Fausti, A., 2017, *The SQuaSH metrics dashboard*, SQR-009, URL <https://sqr-009.lsst.io>

Few, S., 2013, *Information Dashboard Design*, Analytics Press, 2 edn.

[LDM-635], Gower, M., Butler, M., Lim, K.T., 2018, *Data Management Data Backbone Services Requirements*, LDM-635, URL <https://ls.st/LDM-635>

[SQR-019], Sick, J., Fausti, A., 2018, *LSST Verification Framework API Demonstration*, SQR-019, URL <https://sqr-019.lsst.io>

[LDM-622], Swinbank, J., 2018, *Data Management QA Strategy Working Group Charge*, LDM-622, URL <https://ls.st/LDM-622>

Wickham, H., 2014, *Journal of Statistical Software*, Articles, 59, 1, URL <https://www.jstatsoft.org/v059/i10>, doi:10.18637/jss.v059.i10