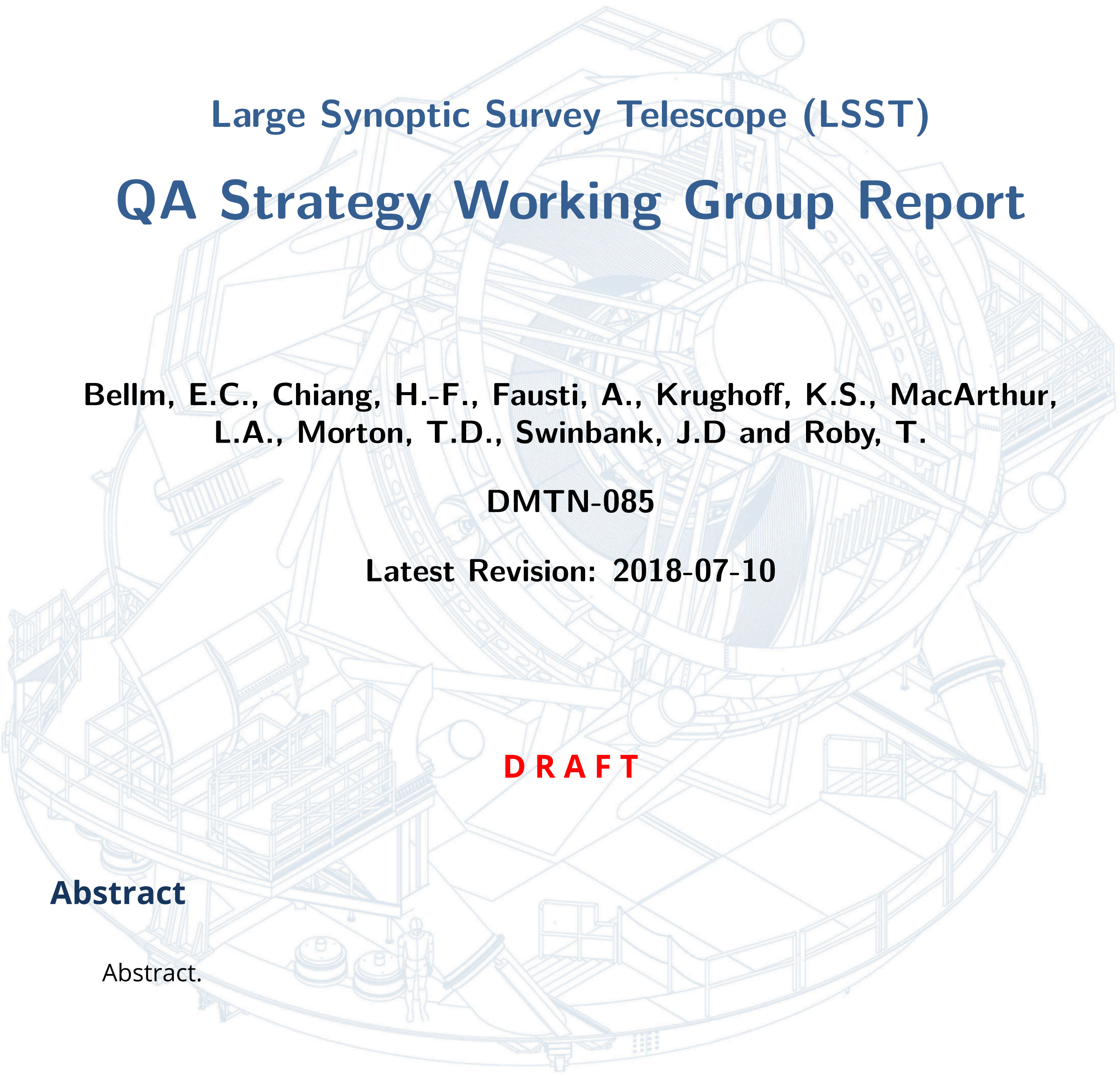
LARGE SYNOPTIC SURVEY TELESCOPE



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: 2018-07-10

D R A F T

Abstract

Abstract.

Change Record

Version	Date	Description	Owner name
0478737	2018-07-10	Unreleased draft.	Bellm et al.

Contents

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.

The complete scope of "Quality Assurance (QA) within Data Management (DM)" is too large to be coherently addressed by any group on a limited timescale. Per its charge, then, the QAWG has constrained itself to considering only those aspects of QA which are most directly relevant to the construction of the LSST Science Pipelines. In particular, we have considered the tools which developers need to construct and debug individual algorithms; tools which can be used to investigate the execution of at-scale pipeline runs; and tools which can be used to demonstrate that the overall system meets its requirements (to "verify" the system). This deliberately excludes broader requirements of Commissioning, Science Validation, or run-time Science Data Quality Assurance (SDQA)[1].

This report consists broadly of three parts. In §2, we describe the approach that the QAWG took to addressing its charge. In §3, we present a high-level overview of the systems that we envisage the future DM comprising. Finally, In §4 we identify specific components — which may be pieces of software, documentation, procedures, or other artifacts — that should be developed to enable the capabilities we regard as necessary. In some cases, these components may be entirely new developments; in others, existing tools developed by the DM subsystem may already be fit for purpose, or can be adapted with some effort. We have noted when this is the case.

Finall,y in Appendix A we define a number of key terms which are used throughout this report and which we we suggest be adopted throughout DM to provide an unambigous vocabulary for referring to QA topics.

[1]Effectively, code executed during prompt or data release production processing to demonstrate that the data being both ingested and released is of adequate quality.

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.

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[2]?
- How can those metrics be displayed on a dashboard? Is a simple time-series adequate, or do we need other types of plotting?
- 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]Scalars, vectors, spatially binned quantities, 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

The group considering the requirements of Science Pipelines developers for debugging explored a number of avenues to make developers lives easier. In doing, so they identified a number of areas in which current systems could be improved to boost both productivity and developer morale.

In particular, they considered three separate scenarios in which developers will require support.

3.1.1 A pipeline is segfaulting

We considered the following:

- A pipeline is failing with a segfault;
- A developer recompiles the failing code with no optimization;
- The unoptimized code is run through a memory analysis tool[3];
- This gives some possible locations where arrays are being overrun;
- The unoptimized code is run under GDB;
- This includes the need to start in PDB and attach GDB when the process enters the compiled C++ code;
- Using the debugging utilities, the dev finds where the array is being overrun.

3.1.2 A pipeline is throwing an exception

3.1.3 A CI run shows a regression in a metric value

3.2 Drill down

Assignee

Tim

Drill-down workflows center on the need to quickly and efficiently identify data processing problems. Typically these will be identified from discrepancies identified at higher levels of summary and aggregation. We emphasize the importance of ease-of-use for the QA analyst in order to shorten debugging cycles. Key capabilities include:

- Rapid retrieval of relevant quantities (metric values, image cutouts, catalog overlays, etc.),
- Readily-(re)configurable interactive plotting tools supporting a developer-oriented browser-based dashboard,
- Automated regression testing.

The system implementing these capabilities must be able to handle large datasets easily, such as, e.g., an entire HSC public data release.

[3]e.g. Valgind.

3.2.1 Metric computation and persistence

Recommendation

The computation, selection, and aggregation steps that define a metric should be well compartmentalized.

Many metrics are defined as statistical aggregates of a per-source computed quantity over a defined selection of sources. In order to enable a low-latency drill-down workflow and analyst flexibility, we recommend that for such metrics, the per-source computations be calculated and stored for *all* sources, and that the selection and aggregation steps be logically seperate. This might be implemented, for example, by giving a `Metric` object separate `.compute()`, `.select()` and `.aggregate()` methods. For metrics computed in this way, the maximum storage granularity (§3.2.1) should be considered to be at the source level.

Other metrics may not be well-defined at the source level, such as the slope of the stellar locus in a given color-color space, or other similar metrics that require fitting a model to many sources at once. Such metrics cannot follow this same `.compute()-.select()-.aggregate()` model, and will have a different maximum granularity (e.g., patch, tract, ccd, visit, or dataset). This maximum granularity should be explictly defined in the definition of the metric.

Recommendation

Metric values should be stored with full granularity (source, CCD, patch, dataset).

Re-running pipelines to recompute metrics imposes a significant overhead to the analyst. We therefore recommend that in general all computed metric values should be stored on disk at the highest relevant granularity. In some cases these are per-source (e.g., source FWHM or shape measurements) and may be included in the relevant object catalogs or in postprocessed tables; in others, the minimum granularity may be at the CCD, patch, or even dataset levels (§3.2.1). This procedure provides the analyst the ability to filter metric values using arbitrary metadata (night, airmass, focal plane position, moon phase...) and re-aggregate to any level desired with any aggregation function (mean, median, percentiles, standard deviation, outliers...). Supplemental storage of higher-level aggregates (e.g., mean FWHM by CCD) is discouraged because of duplication and loss of information, except where speed of visualization of *key* quantities would be impaired due to the need to load large datasets.

Recommendation

Metric values should be stored as "tidy data" in columnar data stores (e.g., Parquet) on disk with the output data repository, in such a way that the data can be loaded quickly enough for interactive work on ~100s of tracts.

"Tidy data" (Wickham, 2014) is recommended as a best practice for data analysis workflows, as it simplifies filter-groupby-aggregate workflows.

We expect that most analysts will be primarily interested in all values of a handful of metric columns at once, which suggests a column-store format will be optimal. The potentially large number of metrics and metric values argues for storage on-disk with the output repository itself. This isolation also facilitates ad-hoc processing & QA workflows by individual analysts. We also recommend a centralized system for tracking performance regressions at a high level (§3.2.3).

Additionally, we also note that in order for interactive tools to be useful for visualizing results of large data repositories, the persistence model must allow for loading and aggregating metric values at the scale of 10s to 100s of tracts with low latency. For example, if parquet files are used to store tables of metric values, it would be preferable for such tables to be stored at the per-tract level instead of the per-patch level, to minimize lots of I/O of small files when loading multiple tracts of data at once.

Recommendation

Metric values should have Butler dataIds.

In order to facilitate joining and filtering metric values by other metadata, metric values should have appropriate Butler dataIds.

Recommendation

We should be able to use the Butler interface to persist and retrieve metric values.

3.2.2 Drill-down workflows and display

Currently, pipeline developers rely on a variety of ad-hoc visualization tools and custom workflows to debug processing problems and investigate the effects of new algorithms. The QAWG recommends development of a browser-based dashboard QA system that is designed to cater to the workflow of the debugging developer, and is informed and guided by current usage of existing tools (e.g. `pipe_analysis` and `qa_explorer`). This is separate from, and in addition to, the SQuaSH dashboard (§4.11).

Recommendation

The QA system should supply a browser-based interactive dashboard that can run on any pipeline output repository (or comparison of two repositories) to quickly diagnose the quality of the data processing. This dashboard should have two levels of detail: a high-level dashboard summarizing top-level global metrics, and and a metric dashboard showing more information on a selected metric (or set of metrics). The more detailed metric dashboard should be able to explore both coadds and inidivual visits.

A developer should be able to navigate a web browser to a URL, enter the path to a data repository at the LDF on which a metric-computing postprocessing task has been run (e.g., on-disk at NCSA in `\scratch` or `\project`), and see an dashboard summarizing the data processing. In addition to computing and persisting the values of these metrics, this task also will have persisted enough information about the metric computations such that the dashboard can read the following from the repository:

- Which metrics were computed?
- What selection was done on the source catalog to compute each metric?
- What is an acceptable value for each metric for this particular data set?

The developer should have a choice to explore either the metrics in a single repository, or the differences in metrics between two data repositories by entring a “comparison repository” in addition to the primary one (that necessarily would need to have the same metrics computed as the primary).

The high-level landing page of this dashboard should allow for at-a-glance summary of the data and identification of any problems. The overall data summary could be in a header displaying the some numbers of interest, such as number of tracts, numbers of visits per filter, total numbers of sources, etc. At-a-glance identification of problems could be accomplished by displaying fully-rolled-up metric summary values in a minimalistic but informative format, such as an array of color-coded buttons or a color-coded table (e.g., green for good, red for bad). This top-level dashboard page should also have minimal but useful visualizations, such as an RA/dec plot of a single metric value (perhaps switchable via drop-down menu).

Clicking on any metric from the top-level page should load up a coadd metric-detail dashboard, which should allow for more detailed exploration of an individual metric. The layout and function of this page will depend on the type of metric. As an example, for metrics derived directly from individual source measurements, it might display a scatter plot of the metric values vs. source magnitude, as well as an RA/Dec scatter plot colormapped by metric value. This could by default show the data for all tracts, but tract-aggregated information could be available for each tract on hovering over the sky map. Upon clicking upon a specific tract, then only the points for that tract will be selected (both in sky plot and scatter plot), and then the sky plot would then display patch-summarized data. Similarly, from here, clicking on a patch would load up just the data in that patch.

This metric dashboard should also allow simulataneous visualization of different metrics, to allow cross-filtering. This might be accomplished, for example, by having a sidebar listing the metrics, and being able to use it to either switch between metrics or to select multiple metrics.

There should be an analogous drill-down mode for exploring visit-level data, with the drill-down levels being visit->ccd rather than tract->patch. There should be seamless integration between coadd and visit mode, to match the typical workflow of a debugging developer, who will first see that there is a problem with a particular metric in a particular tract at the coadd processing level, and then will want to see what visits went into that coadd tract. To enable this, from the “coadd mode” metric dashboard, there could be a way to toggle on/off visit outlines, and to jump to “visit mode” for a selected visit. In “visit mode,” in addition to the scatter/sky plots that “coadd mode” has, there could be an additional figure showing the aggregated metric value as a function of visit number, where outlier visits would be clearly visible. This system would enable a developer to quickly identify bad visits for a particular metric, in just a few clicks.

Recommendation

The dashboard should enable the analyst to start a Jupyter notebook session with the relevant datasets already loaded.

From the metric dashboard, there could be an "explore in Jupyter" button that would load up the dataset in the Jupyterlab environment, which would provide all the tools to make the dashboard visualizations, with the additional flexibility for ad-hoc exploration that the notebook provides.

3.2.3 Automated regression testing

Recommendation

In addition to the repository-level dashboard QA system heretofore discussed, a centralized service should store and plot high-level aggregations of key performance metrics on several datasets that are regularly reprocessed in order to identify performance regressions or improvements due to pipeline changes.

This is essentially the SQuaSH system, discussed in more detail in 4.11. The high-level aggregates that SQuaSH uses may be constructed and submitted by an afterburner task that uses the per-repository metric storage.

Recommendation

An analyst should be able to start an interactive drill-down session exploring the output repository in question in "one click" from any given aggregate displayed by the SQuaSH system.

This defines the relationship between the QA dashboard and SQuaSH. The QA dashboard can point to any data repository on disk in which metrics have been calculated, and does not have to relate to the centralized SQuaSH system in any way. But a developer must be able to access this dashboard in two equivalent ways: either by navigating straight to the dashboard URL and entering the path to a repository, or by beginning at the SQuaSH dashboard, and clicking on a point representing a given pipeline run.

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 exeptions.

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 aggreed 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[4].

4.5 Image viewer

Assignee
Trey

Derived from §3.1.

[4]`https://confluence.lsstcorp.org/display/DM/Pipeline+Debugging+Design`

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:

- 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.

It is important for developers to be able to easily, and interactively, visualize large quantities of catalog data. The PyViz ecosystem (HoloViews/Datashader/Dask/Pandas) is designed for exactly this purpose, and so the QAWG recommends the following:

- Including up-to-date PyViz packages in the LSST stack,
- Developing basic tools using this ecosystem that mesh with existing stack infrastructure (e.g., `Butler`, `AfwTable`),
- Providing training to developers on how to use it,
- Enabling users to use Dask to work with larger-than-memory data, either with the ability to spin up Dask clusters on demand, or (perhaps preferred) to have a central Dask cluster at the LDF to which users can connect.

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 acually 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

To date, SQuaSH has been used to follow a subset of KPMs computed by validate_drp for tracking performance regressions due to pipeline changes by regularly reprocessing test datasets in Jenkins/CI.

The following recommendations would enhance SQuaSH capabilities for DM developers.

Recommendation

SQuaSH should be used by developers for tracking metrics on their particular projects.

Developers can instrument their science pipeline Tasks using `lsst.verify` and create new verification packages to be tracked in SQuaSH (see e.g. `jointcal`). It would be interesting to send results to SQuaSH when testing development branches, so that developers can compare the new metric values with the previous values *before* merging to master. Any metric defined in `lsst.verify.metrics` should be uploaded to SQuaSH including, for example, computational metrics like code execution time.

Recommendation

SQuaSH should provide automated notification of regressions.

Metric specifications in `lsst.verify` include thresholds that can be used to automatically detect and notify regressions. The notifications could be presented to developers by Slack, for example.

Recommendation

SQuaSH should provide a metric summary display.

Verification packages might have specialized visualizations for displaying metric summary information in addition to the current time series plot. DM developers should be able to extend SQuaSH by creating new visualizations following developer documentation provided in the

SQuaSH Documentation (https://squash.lsst.io/)

Recommendation

SQuaSH should support the LDF execution environment in addition to Jenkins/CI.

Pipeline runs on larger datasets (e.g. HSC RC2 weekly reprocessing) require more computation than can be provided in the Jenkins/CI environment. SQuaSH should be flexible to support other environments like the LDF environment.

Recommendation

SQuaSH should be able to store and display metric values per DataIds (e.g. CCD, visit, patch, tract, filter).

Pipeline runs on larger datasets (e.g. HSC RC2 weekly reprocessing) also require to store and display metric values per `DataIds` as opposed to the entire dataset (e.g. test datasets in Jenkins/CI). The ability to identify metric values per filter name or spatially by CCD in a visit or per patch in a tract, would enhance SQuaSH display and monitoring capabilities, turning SQuaSH or its successor into a richer metric dashboard (see also §3.2).

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 facilitiate 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 evolvement in the software stack, such as handling of calibration data and reference catalog, can also make a once-working repository incompatible. Therefore, maintenance is occassionally needed to ensure the usability of a dataset package. **The QAWG recommends having a per-dataset product owner.**[5] 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 implmentations, 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 pacakge 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 [6] 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.

[5] 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/ctio0m9, /datasets/lsstCam, /datasets/decam /datasets/des_sn, /datasets/hsc, /datasets/refcats, /datasets/sdss, /datasets/gapon

[6] `https://developer.lsst.io/services/datasets.html`

- **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);
  - 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.

CI Continuous Integration.

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).

DM Data Management.

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 an source detected on a difference image.

QA Quality Assurance.

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.

SDQA Science Data Quality Assurance.

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