LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)

DM QA Status & Plans

Simon Krughoff and John Swinbank

DMTN-074

Latest Revision: 2018-06-14

DRAFT

Abstract

This document will:

- Describe the current status of "Quality Assurance (QA)" tools, in the broadest sense, currently provided by Data Management;
- Sketch out a set of common use cases and requirements for future QA tool and service development across the subsystem.

It is intended to serve as input to planning for QA currently being undertaken by the DM Leadership Team, the DM System Science Team, and the DM QA Strategy Working Group (LDM-622).

Change Record

Version	Date	Description	Owner name
1.0 (2d17eae)	2018-06-14	First release.	Krughoff & Swinbank

Contents

DM QA Status & Plans

1 Introduction

Across the Data Management subsystem, we (ab)use the term "QA" to refer to various aspects of ensuring that things are "working properly". This spans a wide gamut of applications, including, for example:

- Does our code correctly compile and pass its unit tests?
- Can we demonstrate that the DM system meets Key Performance Metrics (KPMs) and satisfies other aspects of our requirements documentation (LSE-29; LSE-30; LSE-61)?
- Do we properly understand the operation of our scientific algorithms, both individually and operating in concert? Can we identify when the results they produce are scientifically lacking?
- Do we provide tools to developers, scientists and other members of the DM team to help them understand and debug the code and systems they are constructing or using as part of their work?
- How do we track computational performance across the system, from execution times of scientific algorithms to the scaling properties of database queries?
- Can we monitor systems within the LSST Data Facility to ensure that they are operational and performing correctly?
- Can we identify problems which stem from bad data, as distinct from bad software or services?

To date, the DM team has built a number of tools which address parts of these problems. However, a coherent, unified vision for how they fit together remains lacking. This document will[1] catalogue the tools that are currently available, will identify use-cases and the requirements arising from them, will determine to what extent the existing tools satisfy those requirements, and will suggest directions for future development.

[1]Ultimately; the current draft does not yet address all aspects of this scope!

2 Current Tooling

We begin by cataloguing the tooling which has been developed to date, or which will be deployed in the short term[2]. These tools are listed by the team within DM which originated or leads the development of them.

2.1 Alert Production (AP)

2.1.1 ap_pipe and ap_verify

Development within the Alert Production group has focused on the construction of an instrumented "end-to-end" alert production pipeline.

The pipeline code itself lives in the `lsst-dm/ap_pipe` package in GitHub. This provides prototype implementations of major alert production pipeline components (single frame processing, image differencing, source association), and a control script to string them together[3]

The `lsst-dm/ap_verify` package is a companion to ap_pipe. ap_verify effectively wraps pipeline functionality in a form that is intended to be appropriate for regular testing in CI (§2.6.1). As such, it provides:

- A standardized way of defining "dataset" packages, each of which provide a curated test dataset;
- The facility to ingest data into Butler repositories suitable for processing with LSST stack tools;
- Facilities for instrumenting and calculating metrics based upon running pipeline code;
- Submission of calculated metrics to SQuaSH using the lsst.verify system (§§2.6.5 & 2.6.4).

Based largely upon the experience gained in building ap_verify, the AP team has put considerable thought into the appropriate mechanisms for extracting metrics from running pipeline `Tasks`. This has resulted in DMTN-057, which describes a number of possibilites for how this might be standardized. At time of writing, none of these proposals have been formally adopted by the project.

[2]That is, which we can count on becoming available shortly, regardless of any future course corrections which result from this document or other discussion.
[3]At time of writing, this system is being migrated to use the DM-standard `CmdLineTask` framework.

2.2 Data Release Production (DRP)

2.2.1 `afwDisplay`

The `lsst.afw.display` framework provides a Python API for displaying and manipulating images and overlaying information source lists based on LSST stack primitives. The code is backend-agnostic — that is, the developer writes to a device-independent common API, and the display can appear on one of a number of image viewers. Back-end implementations are currently available for *SAOImage DS9*[4], *Matplotlib*[5], *Firefly*[6] (§2.3.1) and *Ginga*[7]. Adding more backends is relatively straightforward. However, the goal of being backend-agnostic may be in conflict with taking advantage of special capabilities or functionalities offered by particular tools.

It is worth noting that there is no `afwDisplay` equivalent for working with figures: pipeline developers instead tend to access *Matplotlib* directly.

The `afwDisplay` code does not appear in the work breakdown structure; no team has formal responsibility for it. However, the DRP team are the primary maintainers. The teams within science pipelines all make use of `afwDisplay`.

2.2.2 ci_hsc

The ci_hsc package provides a curated set of HSC data and a SCons[8]-based system for processing it through a DRP-like workflow. The resulting data products are automatically sanity-checked: that is, we establish that the expected outputs have been produced and contain a plausible number of objects, reasonable selection of objects used for PSF determination, and so on, but we do not do detailed analysis of source measurements or astrophyisical plausibility. This means that ci_hsc provides an excellent way to catch regressions in pipeline machinery and integration, but is not sensitive to more subtle algorithmic issues.

ci_hsc may be run standalone by individual developers — it takes around three hours — and is periodically run by the CI system (§2.6.1).

[4]`https://github.com/lsst/display_ds9`
[5]`https://github.com/lsst/display_matplotlib`
[6]`https://github.com/lsst/display_firefly`
[7]`https://github.com/lsst/display_ginga`
[8]`http://www.scons.org/`; note that the choice of SCons here is ingenious — it provides many desirable features in a workflow system — but also completely non-standard across the rest of the codebase, and can make this package hard for developers to engage with.

2.2.3 Automated static plots with pipe_analysis

The pipe_analysis package provides scripts which inspect a repository of processed data and generate static plots of the distributions of scientifically relevant quantities. These may be manually inspected to demonstrate the internal consistency and fidelity of photometric and astrometric measurements on the visit-stage and coadd-stage catalog outputs. They also can be used to compare catalog outputs from two different reruns

The plots generated by pipe_analysis have been refined, and new plots added to the collection, based on several years of investigating issues encountered by the DRP group in processing HSC data.

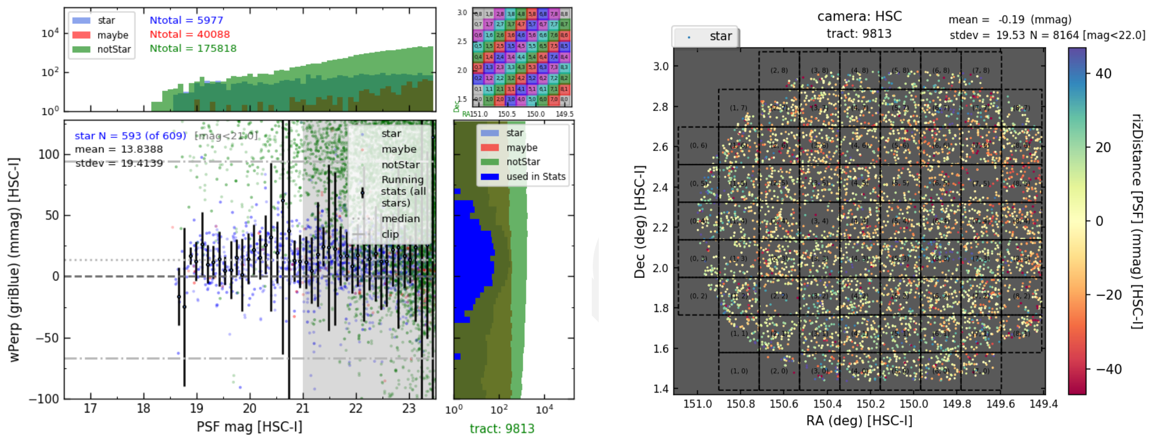

FIGURE 1: Examples of static plots generated by pipe_analysis, courtesty of Yusra AlSayyad.

Examples of plots produced by pipe_analysis are shown in Figure 1. Note that this figure shows but two of many plots generated.

2.2.4 Dynamic "drill-down" plots

Tim Morton (Princeton) is currently building a toolset that offers the ability to create plots of tracts-worth of catalog data to find patterns and pathologies. These interactive and linked visualizations, inspired by the pipe_analysis plots (§2.2.3), are created and explored live in a Jupyter notebook environment.

The current system plotting of multiple quantities of interest, linked to each other and to sky maps of each quantity; scanning through sky plots of a quantity from each visit contributing to a coadd; and in-notebook quick-look of images corresponding to catalog data points.

This toolset is being designed with scalability in mind, and is easily capable of handling millions of datapoints.

2.2.5 Large scale scientific analysis of HSC data

Approximately annually, a derivate of the LSST codebase is used to process the full collection of data from the HSC Strategic Survey Program. This is coupled with an intensive QA effort (using several of the tools listed above, in addition to pipeline production scientists in Japan looking at the data) to verify that new desired features are working as expected and that the scientific performance of the pipeline has not regressed, before the results are released to the HSC community. These data releases form the basis of scientific analyses by the HSC collaboration, which in turn identify issues that may go unnoticed during pipeline processing.

Processing the entire survey dataset in this way enables the identification of edge and corner cases that may otherwise go unnoticed. Unfortunately, the pipeline outputs are proprietary until they are released to the world a couple of years later, but members of the DRP team at Princeton can access them and use the results to trigger development and bug-fixes using public data.

2.3 Science User Interface & Tools (SUIT)

2.3.1 Firefly

Firefly is the IPAC's "advanced astronomy web UI framework". It provides data browsing, image viewing, and plotting functionality, and will be the focus of the "portal aspect" of the Science Platform.

Firefly is a client-server application; a Java-based server component is colocated with the data, and is accessed through a Javascript based UI in the user's browser. At various times, installations of Firefly have been made available to DM developers on project-provided compute hardware[9], but these are not regularly maintained or accessed by pipelines developers. A firefly server is deployed with every instance of the JupyterLab environment (§2.6.7).

Firefly interfaces to the standard image display system used by the LSST pipelines (§2.2.1) are

[9] i.e. `lsst-dev01.ncsa.illinois.edu`

available.

The functionality available from Firefly may be useful to other DM teams, in particular providing visualization and plotting services in support of algorithm development and investigating the properties of data releases. However, the SUIT team is not scoped to deliver capabilities to the rest of DM: they are focused on developing Firefly to meet the requirements of the Science Platform. Other parts of DM may, of course, benefit from the tools developed for this purpose.

2.4 Science Data Archive & Application Services (DAX)

2.4.1 Database intgration testing

Qserv has a multi-node integration test which spins up several shard servers and a master via containers, loads a test dataset, and issues a suite of queries checking against known/expected results. This is run automatically using the Travis CI service on all changes.

Unfortunately, the nature of this test and the Travis CI service renders it complicated and brittle; these tests are often broken for reasons unrelated to a developer's changes. The same test can, however, be run manually by individual developers before merging.

Larger-scale QServ integration tests are carried out periodically at CC-IN2P3 and the LSST Data Facility. These consist of deploying the containerized Qserv system across a large cluster and executing queries against large-scale test datasets. These tests are currently controlled by a collection of scripts, but work is underway to move to a Kubernetes[10]-based infrastructure.

2.4.2 Database performance testing

Performance testing for qserv is performed annually, on synthetic data sets that are on a glide path to the scale of LSST DR1. The test procedure is described in LDM-552; examples of the reports produced include DMTR-13 and DMTR-16.

Performance constraints are particualrly an issue for the Level 1 (Prompt) database prototype. A simulator was developed to test its performance at various scales. This simualtor is described in DM-6365. It is not regularly run as part of Continuous Integration (CI).

[10]https://kubernetes.io

2.4.3 DAX services

The current development version of the imgserv service is shipped with an integrated test suite of several dozen queries, which can be quickly executed against the service via a built-in CLI, running against either a small test dataset (included with the code) or the larger data sets currently deployed in the Prototype Data Access Center (PDAC). This same test suite can also be executed directly over HTTP[11] to validate the top-end web service layers.

The test suite is not yet integrated with the CI (§2.6.1) system, but in principle there is no reason why it could not be.

This testing strategy has been found very successful for imgserv, and it will be extended to other DAX services as development continues.

2.4.4 `lsstDebug`

The author of an algorithmic task knows which intermediate data products and diagnostic plots are useful for answering questions about its behavior. The `lsstDebug` framework allows developers to insert code into their `Tasks` that displays images (using standard DM primitives) and makes plots that are only run when the `CmdLineTask` is invoked with the `--debug` parameter.

Historically, the `lsstDebug` system has been poorly documented and the subject of some confusion. There are no published guidelines about appropriate ways to use it or expectations of how developers should instrument their `Tasks`.

Note that the `lsstDebug` system did not originate with the DAX team (and, indeed, it's quite likely that no member of DAX has ever used it), but it falls within their remit as part of the "task framework".

2.5 Data Facility (LDF)

[11]Using a client such as *cURL*; `https://curl.haxx.se/`.

2.5.1 Regular manual reprocessing of HSC data

Every two weeks, members[12] of the LDF team reprocess some three tracts of HSC data, comprising the "RC2" dataset[13], through the current weekly release of the DRP pipeline[14], including post-processing with the pipe_analysis system (§2.2.3). Any substantive changes in processing outputs[15] are flagged for the attention of the DRP team. The results of the four most recent processing campaigns are retained for reference.

In addition, Data Facility staff periodically, on request from the DRP team, reprocess the whole of the first HSC public data release ("PDR1") on the Verification Cluster.

2.6 Science Quality and Reliability Engineering (SQuaRE)

2.6.1 Continuous Integration services (Jenkins)

SQuaRE provides and maintains the CI system which regularly builds and tests much of the DM Science Pipelines and Qserv codebase. The Systems Engineering simulations team also take advantage of the same system; it is not used by the other teams within DM.

The CI system is widely used for a number of related purposes, including:

- Testing of code by developers before it is merged. Primarily this runs all unit tests in all packages, but can also run more comprehensive integration tests;
- Executing metric collection systems (e.g. validate_drp, §2.6.6);
- Building and preparing code for public and internal release. This includes packaging both source and binary releases and building Docker images for use in the JupyterLab environment §2.6.7.

[12] Principally Hsin-Fang Chiang
[13] The RC2 dataset is fully defined in DM-11345.
[14] Currently, this means `makeSkyMap.py`, `singleFrameDriver.py`, `mosaic.py`, `skyCorrection.py`, `coaddDriver.py` and `multiBandDriver.py`.
[15] For example, changes to CCDs logged as failing certain pipeline steps, or new errors or exceptions logged during pipeline operation; we do not consider changes to the contents of the science data products "substantive" for this purpose, as long as those data products are generated as expected.

2.6.2 Unit test framework

All code written for DM is expected to be accompanied by appropriate unit tests[16].

The ideal unit test demonstrates that an individual "unit" of software (a class, a function, etc) is operating correctly. In practice, many tests used by DM go well beyond this, for obvious reasons: it's hard to demonstrate that the `ProcessCcdTask` class is operating correctly without relying on classes for loading data, performing instrument signature removal, source detection and measurement, calibration, and so on. As a natural result of this, DM's unit tests are often (ab)used to provide mini-integration tests. While this conflation of integration and unit testing provides convenient test capabilities which are not easily available elsewhere, it leads to fragility in the test suite, and makes it impossible, in general, to test modules in a standalone way: any package needs all of its dependencies available to execute its test suite.

Although the creation of unit tests is not a responsibility of SQuaRE, the technology used to implement them, and their regular execution as part of the CI system, is (although we note that much of the current test harness has been developed in close cooperation with the Architecture team).

2.6.3 Stack Demo

The Stack Demo, `lsst/lsst_dm_stack_demo`, contains a curated selection of SDSS data, a script which executes single frame measurement on that data, and a set of precomputed expected outputs. The results of measurement are compared with the expected values, and an error is thrown if any values (including number of detected sources, their positions, and assorted measurement algorithm results) are different.

Note that the expected values are the results of some previous processing run. There is no ground truth for this data, and these results have not been rigorously analysed to demonstrate that they are correct. In short, this procedure can verify that changes being made do not substantially change the results of processing, but cannot verify that the results are intrinsically correct.

The Stack Demo is available for end users to run standalone, but it is also automatically run as part of most CI (§2.6.1) jobs.

[16]Refer to the Unit Test Policy in the Developer Guide, but note that our implementation does not follow the terminology it uses

2.6.4 lsst.verify

lsst.verify is a framework for packages that measure software and data quality metrics. A metric can be any measurable scalar quantity; some examples are in the LSST Science Requirements Document (LPM-17), though packages can also define ad hoc metrics. Measurements made through lsst.verify can be uploaded to LSST's SQuaSH monitoring dashboard (§2.6.5). The intention behind the development of `lsst.verify` is that it serve as the primary mechanism for verifying LSST pipelines[17].

2.6.5 SQuaSH

SQuaSH is a metric aggregation and display service. It provides time-series visualization of selected metrics derived from LDM-502. Metics may be submitted to SQuaSH using lsst.verify (§2.6.4). These metrics are stored and tracked, making it possible regression alerts to be issued via several mechanisms including Slack notifications[18]. All metrics measurements are accompanied by the associated code changes to aid in identification of relevant packages in the event of a regression.

To date, the primary use of SQuaSH has been to follow a selection of KPMs. However, the framework is not restricted to only KPMs: any metric defined in `lsst.verify.metrics` can be uploaded to SQuaSH. SQuaSH then serves as both a database and search engine for metric measurements as well as a visualization platform for those measurements.

For a selection of the measuremnts corresponding to select KPMs, specialized visualiztions have been developed, but it is also possible to query the metric database[19] for other measurements to feed to ad hoc visualizations. The intention is that SQuaSH will be used by developers to track low level metrics on their particular project as well as higher level Subsystem and Project level metrics.

SQuaSH is described in SQR-009.

2.6.6 Validation packages

The validate_drp package will automatically run single-frame processing on a dataset, calculate a subset of metrics derived from the Science Requirements Document (LPM-17), upload

[17]The process of validating the pipelines in a scientific context are left to the DM validation team.
[18]Note that this capability does not seem to be widely advertised or understood.
[19]Using GraphQL.

the results to SQuaSH[20]

(§2.6.5), and evaluate expected analytic models for photometric and astrometric performance following Ivezic et al. (2008).

It is worth noting that, although these packages are referred to as "validation" packages, in fact they form part of the *verification* process, and will likely be renamed in future.

Currently, validate_drp supports the following metrics[21]:

- Relative astrometry
  - AM1, AF1, AD1
  - AM2, AF2, AD2
  - AM3, AF3, AD3
- Photometric repeatability
  - PA1, PA2, PF1
- Residual PSF ellipticity correlations
  - TE1, TE2

The focus, to date, has been on testing data release algorithms, but the SQuaRE team intends to extend the system to address prompt processing and some lower level systems (e.g. Jointcal).

In addition, three curated datasets for use with validate_drp are provided. These are based on CFHT (validation_data_cfht), DECam (validation_data_decam) and HSC (validation_data_hsc). These datasets are reguarly reprocessed by the CI system (§2.6.1) and the results uploaded to SQuaSH.

Note that the validation_data packages contain both "raw" data and processed data which has been ingested to a Butler repository. There is no regular routine for updating that processed data in light of changes to the software; developers have occasionally found it unclear or confusing whether this data is actually supposed to be usable or useful[22].

[20]The `master` branch of validate_drp currently uses an older system for calculating and transmitting metric measurements, there is currently a complete port to the `lsst.verify` system on the `verify_port` branch.

[21]From `etc/metrics.yaml` of validate_drp `w.2018.07`

[22]In this context, note RFC-243 which requests the regular generation of "a pipeline output dataset".

2.6.7 Hosted Jupyter notebooks

The JupyterLab environment is a complete, containerized system for spinning up Jupyter Notebooks in a hosted environment. The environment comes complete with pre-compiled recent versions of the stack; currently, these are the three most recent weekly releases, the two most recent daily releases, and the most recent major release. The JupyterLab environment provides persistent personal storage, which may be used to customise the user's personal environment, and shared storage, which can be used for exchanging data, etc. It also provides commonly expected utilites like a shell prompt as well as a text editor.

Currently there are three deployments; all of these target a focused user community, rather than widespread. These are:

- The DM team developing the JupyterLab system itself;
- The Systems Engineering team, as they begin commissioning exercises;
- The Education & Public Outreach team.

Since these are currently hosted in the on commercial "cloud" systems, since they are relatively small; they are all trivially re-deployable to a project hosted cluster resource running Kubernetes when such a resource becomes available.

3 Requested Functionality

In this section, we briefly enumerate functionality that has been requested from across the project under the banner of "QA". We will consider four broadly separate "regimes" in which QA procedures of one sort or another are relevant:

Developer support Provide developers with the tools they need to work quicky and effectively when developing scientific algorithms and/or other core pieces of the system.

Code quality Ensure that our code is buildable, executable, and maintainable.

Metric verification Demonstrate that we can reliably execute code at scale and track its performance against predefined targets.

Science validation Check that the results of processing are scientifically useful, and characterize all the data products produced.

This brief sketch is not intended to be comprehensive: for a detailed consideration of DM's sdesires, requirements and future plans for QA, refer to forthcoming report from the QA Working Group (DMTN-085).

3.1 Developer Support

3.1.1 Test datasets

When writing code, it is essential that developers have low-friction access to test datasets from a variety of instruments.

These datasets must be well understood, in terms of their origin and properties.

It is convenient that such datasets should be available as raw (unprocessed) data, but also that intermediate and final data products from various stages of pipeline processing be made available. This facilitiates testing of algorithms which are only relevant to later parts of the pipeline or analysis codes (e.g. Jointcal, the Science Platform) without the need for time or expertise to rerun basic reduction steps.

These processed data products must be kept up-to-date: they should always correspond to the products that would be produced by the current state-of-the-art pipeline.

3.1.2 Visualization, plotting and debugging frameworks

Although `afwDisplay` (§2.2.1) goes some way towards addressing needs for image display, there is no equivalent for plotting or displaying catalogue data.

The `lsstDebug` system for debugging `Tasks` is poorly documented and badly understood, and — worse — expectations for how debugging information should be collected are unclear.

3.1.3 Support for running and debugging at scale

Developers request more support with running at scale on e.g. the Verification Cluster, and, crucially, understanding the results of those runs. Current debugging techniques consist of little in the way of status information, reams of unstructured logs, and extensive use of `grep`.

3.1.4 Notifications and dashboards

Many of the systems being described or requested rely on some analysis of the performance of code being proposed by some particular developer against standards for code quality, algorithmic correctness, performance, etc. Wherever possible, the results of these checks should be presented to developers automatically (e.g. by Slack and/or e-mail notification), and delivered in a clear and consistent way (rather than requiring developers to navigate and integrate information from multitudinous chatbots, websites, and so on).

3.2 Code quality

3.2.1 Unit tests

Extending unit test coverage and usability protects all developers from code being broken by changes elsewhere in the stack.

Writing test code that covers complex systems (e.g. `Tasks`, display frameworks) can be awkward: better guidance, standards or frameworks for doing this would reduce the workload on the developer.

The Developer Guide should reflect current best practice and expectations in respect of testing.

The overloading of unit tests to act as integration tests (as discussed in §2.6.2) may contribute to making tests slow and hard to work with. Stricter guidelines for making unit tests more focused, together with better integration testing facilities, may help to address this.

3.2.2 Integration tests

Integration tests for major DM components should be run regularly. These would likely include (at a minimum) separate, "end-to-end" tests for each of the alert production and data release production systems, together with other major components of the system (e.g. alert distribution, databases). Coverage should be tracked, so that we understand which aspects of the system are being tested and which are not[23].

Integration tests are not responsible for scientific validation of data products. However, they

[23] This will include e.g. which `Tasks` are not executed by any integration tests because they are disabled by pipeline configuration.

should track all other aspects of execution. This will likely include:

- Successful execution (the code does not abort or fail);
- Generation of all expected data products;
- Significant changes in log messages (e.g. new warnings or errors being logged);
- Changes in data processed, rejected, etc (e.g. some CCDs that previously passed now fail);
- Calculation of metrics describing the results.

Note that integration tests may be executed in (at least) two situations:

- Individual developers wish to check their code before a merge;
- The current `master` branch, and any maintained release branches, should be regularly exercised to ensure that no regressions have been merged.

Note that integration tests which developers are expected to run before merging to `master` must be (relatively) fast to avoid increasing developer friction.

Checks for regression on `master` should be treated with the appropriate gravity: they should result in notifications (§3.1.4) to responsible individuals, and ultimately be escaled to e.g. the DM Project Manager if they are not addressed.

3.2.3 Static analysis, code linters, etc

A variety of other tools may be applied to identify potential issues or coding standards violations. While these may often be effective in improving quality, care must be taken to ensure both that they don't introduce excessive churn (making e.g. Version Control System (VCS) history hard to follow) and that the benefits gained are worth the implementation costs.

3.3 Metric verification

3.3.1 Performance analysis

Execution time, under controlled conditions[24], should be tracked for all time-sensitive areas of the codebase.

This will obviously include high-level summaries of the time taken to execute major pipeline components or database queries (which must be tracked in order to satisfy high-level requirements). However, it must also make it easy to track the impact of changes to individual algorithms or other components, and to provide feedback to developers if their changes introduce performance regressions.

3.3.2 High-level metric tracking

KPMs derived directly from high-level requirements documents (LPM-17; LSE-29; LSE-30, etc) are not directly applicable to the DM codebase, since they characterize the performance of the LSST system *as a whole*. However, the Subsystem (presumably under the guidance of the System Science Team (SST)) should define, and *rigorously* track our performance against, equivalent quantities that can be well-defined in the context of DM (for example, measurements made on simulated or precursor datasets).

Actively tracking performance is key. This could take the form of notifications (§3.1.4) to key individuals (e.g. the DM Project Manager, DM Subsystem Scientist, DM Pipelines Scientist, T/CAMs of relevant groups) on any regression, or a weekly meeting of key personnel to review collected measurements.

Ultimately, successful verification and delivery of the DM Subsystem is predicated on hitting predefined targets in all of these metrics, and the DM System Requirements (LSE-61) should be updated to reflect this.

3.3.3 Ad-hoc metric calculation and tracking

While some metrics, as described in §3.3.2, should be regularly tracked under controlled conditions on specific datasets, there is also a need for other metrics to be calculated and — potentially — tracked.

There are a number of considerations here:

[24]We might consider whether AWS instances, or other shared hardware, qualify.

- Developers will wish to execute code against arbitrary datasets and obtain a high-level summary of its performance. Such metrics might be tracked in the short term for developer convenience, but varying compute platforms, datasets and algorithmic implementations will render them of little value for long-term monitoring.
- Some ad-hoc metrics should be tracked indefinitely. However, they will then require that the platforms and datasets upon which they are being tracked are held constant to ensure that the values being recorded are comparable with earlier measurements.
- Interfaces for defining metrics should be made available to developers.
- The types of metric being collected should be considered. From use cases collected to date, it's not immediately obvious whether the combination of scalar summary values plus the ability to "drill down" to processed data products, is adequate, or if more complex datatypes are required[25].

3.4 Science validation

3.4.1 Drill-down

It should be possible to display catalog data or summary metrics of interest as a function of positio on the sky for (at least) tracts worth of data (and preferably all sky). When patterns or pathologies are observed, it should be possible for the user to interactively "drill down" to investigate their source. For example, this includes plotting multiple (user selected) quantities of interest, enabling brushing & linking between them, and the ability to round-trip data to a Jupyter notebook for further analysis. The tooling should enable the user to investigate questions such as:

- Why are the colors in this region of the sky systematically offset from the stellar locus?
- Why is the difference between CModel Mags and Psf Mags multimodal?
- ...

[25]Perhaps equivalent: should the metric-tracking system store pipe_analysis (§2.2.3-style plots, or other intermediate data products, from which summary metric values might be derived, in addition to the metric measurements themselves, or should it be possible to generate these plots on the fly from the output dataset? If the latter, must the output dataset be preserved indefinitely?

4 Glossary

CI Continuous Integration. See §2.6.1.

KPM Key Performance Metric. KPMs are a subset of the high-level LSST requirements which were chosen in LDM-240 to track progress of the DM system towards completion. The current set as they pertain to the LST Science Pipelines are described in LDM-502. Note that, being derived from high-level requirements, KPMs in general describe the performance of the *complete* LSST system, and not that of the DM subsystem; it follows that it is impossible to verify all KPMs purely based on code and services provided by DM.

metric A measureable quantity which is used to characterize the performance of some aspect of the DM system. For example, metrics might include photometric repeatability or alert production latency.

PDAC Prototype Data Access Center.

QA Quality Assurance.

SST System Science Team.

VCS Version Control System.

References

[DMTR-13], Becla, J., 2015, *Qserv Summer 15 Large Scale Tests*, DMTR-13, URL `https://ls.st/DMTR-13`

[DMTN-085], Bellm, E., Chiang, H.F., Fausti, A., et al., 2018, *QA Strategy Working Group Report*, DMTN-085, URL `https://dmtn-085.lsst.io`,
LSST Data Management Technical Note

[LSE-29], Claver, C.F., The LSST Systems Engineering Integrated Project Team, 2017, *LSST System Requirements (LSR)*, LSE-29, URL `https://ls.st/LSE-29`

[LSE-30], Claver, C.F., The LSST Systems Engineering Integrated Project Team, 2018, *Observatory System Specifications (OSS)*, LSE-30, URL `https://ls.st/LSE-30`

[LSE-61], Dubois-Felsmann, G., Jenness, T., 2017, *LSST Data Management Subsystem Requirements*, LSE-61, URL `https://ls.st/LSE-61`

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

[DMTN-057], Findeisen, K., 2017, *Integrating Verification Metrics into the LSST DM Stack*, DMTN-057, URL `https://dmtn-057.lsst.io`,
LSST Data Management Technical Note

[LPM-17], Ivezić, Ž., The LSST Science Collaboration, 2011, *LSST Science Requirements Document*, LPM-17, URL `https://ls.st/LPM-17`

Ivezic, Z., et al., 2008, ArXiv e-prints (`arXiv:0805.2366`), ADS Link

[LDM-240], Kantor, J., Jurić, M., Lim, K.T., 2016, *Data Management Releases*, LDM-240, URL `https://ls.st/LDM-240`

[LDM-552], Mueller, F., 2017, *Qserv Software Test Specification*, LDM-552, URL `https://ls.st/LDM-552`

[LDM-502], Nidever, D., Economou, F., 2016, *The Measurement and Verification of DM Key Performance Metrics*, LDM-502, URL `https://ls.st/LDM-502`

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

[DMTR-16], Thukral, V., 2017, *Qserv Fall 16 Large Scale Tests/KPMs*, DMTR-16, URL `https://ls.st/DMTR-16`