LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)
Data Management

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. (chair) and Roby, T.

DMTN-085

Latest Revision: 2019-01-24

Abstract

This document describes the work undertaken by the QA Strategy Working Group and presents its conclusions as a series of recommendations to DM Project Management.

Change Record

Version	Date	Description	Owner name
1.0	2019-01-24	Released to DMLT.	Swinbank

Document source location: `https://github.com/lsst-dm/dmtn-085`

Contents

1 Introduction **1**

2 Approach to the Problem **2**

2.1 Pipeline debugging . . . 3

2.2 Drill down . . . 3

2.3 Datasets and test infrastructure . . . 4

3 Design sketch **4**

3.1 Pipeline debugging . . . 4

3.2 Drill down . . . 6

3.3 Datasets and test infrastructure . . . 7

3.3.1 Small-scale unit and documentation tests . . . 7

3.3.2 Integration tests . . . 7

3.3.3 Metric and performance tracking . . . 8

4 Core components **8**

4.1 Updated pipeline debugging system . . . 9

4.2 Logging . . . 10

4.3 Workload management system . . . 11

4.4 Visualization . . . 12

4.4.1 Catalog visualization . . . 13

4.4.2 Image visualization . . . 14

4.5 Provenance . . . 15

4.6 Code quality documentation . . . 17

4.6.1 Unit tests . . . 17

4.6.2 Code review . . . 18

4.7 Documentation and examples . . . 18

4.8 CI system updates . . . 20

4.8.1 Notifications and failures . . . 20

4.8.2 Execution environment . . . 21

4.9 Standard format dataset packages 22
4.10 Standard format test packages 25
4.11 Metric collection & tracking 26
4.11.1 Metric definition and calculation 27
4.11.2 Collecting metrics 30
4.11.3 Metric tracking: dashboard and alerts 31
4.12 Drill-down workflows 33

5 Conclusion **37**

A Recommendations **37**

B Glossary **40**

QA Strategy Working Group Report

1 Introduction

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

It is worth starting by revisiting the definition of Quality *Assurance*, or QA. In particular, note that LDM-522 defines QA as "Quality *Analysis*", a process which is undertaken by humans during commissioning and operations, and which stands in contrast to automated Quality Control (QC) systems. For the purposes of this group, we have taken a more holistic definition (following the guidance in the charge) of QA, covering all activities undertaken by the Data Management (DM) construction project to ensure the ultimate quality of its deliverables.

The complete scope of "QA within 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 pipeline runs at scales beyond those which are trivially addressable by individual developers on single compute systems; 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.

Throughout, we provide a number of recommendations which we suggest should be adopted by the DM Subsystem as a whole. These recommendations identify specific actions that

[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.

should be taken or capabilities that should be provided; in general, addressing them will require action by the Project Manager or T/CAMs to schedule appropriate activities.

Finally, in Appendix B, we define a number of key terms which are used throughout this report and which we suggest be adopted across DM to provide an unambiguous vocabulary for referring to QA topics.

Recommendation

Adopt the definitions of QA-related terms in the DMTN-085 glossary subsystem-wide.
For example, by inclusion in a subsystem-level glossary; refer to DM-9807, DM-14877, and DM-14911.

2 Approach to the Problem

The QAWG addressed its charge by sub-dividing the problem space into three separate — but overlapping — 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 automated 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.

As each sub-group 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

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 that don't fit in memory on a single node?
- How do we handle fake sources? Is this a provenance issue?

2.2 Drill down

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 types of metric 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 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?

[2] Scalars, vectors, spatially binned quantities, etc.

- 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

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 fixed datasets?
- How is the system for tracking metrics managed? — how are the metric calculation jobs run? By whom? How often?
- How should run-time performance of the science algorithms be tracked?

3 Design sketch

In this section, we summarize the issues identified and approaches suggested by the groups described in §2. From these, we synthesize a number of concrete actions — tools to be developed, documentation to be provided, etc — which we recommend to the project in §4.

3.1 Pipeline debugging

The group does not identify a single, over-arching tool or concept which would solve the problem of developer productivity and happiness. Instead, we believe that developer needs can be best addressed by making incremental improvements to a number of core pieces of DM technology and infrastructure which team members regularly interact with. In particular, we identified the following major "pain points" for developers:

- The pipeline debugging system, `lsstDebug`, is badly documented and awkward to use, and developers lack appropriate guidelines on how to embed debugging information

into algorithms or on how to use that information to most effectively debug running pipelines.

- Developers find it hard to know how to debug their code when it is running at scale. Issues include parsing logs when a large number of concurrent processes are running; inadequate documentation of the existing Slurm system; uncertainty about replacement of Slurm with a future workflow system; and difficulties in understanding the provenance of data products.
- The project has issued unclear guidance and inadequate documentation to developers about the appropriate tools to be used for visualizing data. This is most pronounced for image visualization[3], but also applies to catalog data.
- Developers struggle to identify suitable datasets for running tests — both small and large scale — in a convenient form. Large repositories exist on the project GPFS system, but it's unclear what they contain or how to effectively access them[4]; smaller repositories exist on GitHub[5], but they are inconsistent in structure, content and documentation: it's impossible for a developer to quickly identify data which is relevant to their use case, or to establish whether some particular reduction of the data is "correct". Instead, they rely on folklore and talking to peers to find data that "worked for somebody else", with (often) predictably frustrating results.

To address these issues, the working group suggests the development of a number of separate-but-related components. These include:

- An updated system for instrumenting running pipeline code; effectively, a revision of `lsstDebug`. This is developed further in §4.1.
- A revised set of tooling for generating, aggregating and analyzing logs. This is developed further in §4.2.
- Revised documentation on interacting with the workload management system. This is developed further in §4.3.

[3] Developers have the impression they "ought to" be using Firefly, but there is much uncertainty around its suitability for the task and its future development direction.

[4] A canonical example is the Hyper Suprime-Cam (HSC) public data release: the volume of data is overwhelming for a developer who simply needs some representative HSC data to test an algorithm.

[5] e.g. validation_data_hsc, afwdata, ap_verify_hits2015, etc.

- Guidelines for the structure and maintenance of data repositories. This is developed further in §4.9.
- A clear roadmap for the development of visualization tools, and, derived from that, guidelines on how to apply those in the development of pipelines. This is developed further in §4.4.

In addition, the WG suggests prioritization of developer-accessible systems for tracking and understanding the provenance of pipeline outputs. For additional comments on provenance from a QA perspective, refer to §4.5.

3.2 Drill down

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 therefore envisage a drill-down system that provides rapid retrieval of relevant quantities (metric values, image cutouts, catalog overlays, etc.) combined with readily-(re)configurable interactive plotting tools. This is provided in a browser-based tool which enables the user to rapidly access successive layers of detail on aggregated metrics (effectively "de-aggregating" them on demand), and ultimately enables the user to seamlessly transition to an interactive analysis environment[6] primed with the data under investigation. Further details about the design and capabilities of such a system are presented in §4.12.

We suggest that this system should be linked with a metric tracking system: selected aggregate metrics from successive comparable[7] processing should be tracked as a time series, with the user able to identify outliers and rapidly switch to the drill-down system to investigate. This capability is an extension of that already provided by SQuaSH, and is discussed further in §4.11.3.

The system implementing these capabilities must be able to handle large datasets — for example, an entire HSC public data release — quickly. Furthermore, we emphasize the importance of ease-of-use for the QA analyst in order to shorten debugging cycles.

We also note that an effective provenance system is fundamental to both drill-down and

[6] i.e. a Jupyter notebook
[7] i.e. using the same input data, running with the same configuration

metric-tracking use cases; we discuss this further in §4.5.

3.3 Datasets and test infrastructure

The group regards this part of its charge as qualitatively different from the other two (§§3.1 & 3.2): where those focus on servicing the needs of the individual developer or analyst, this material describes the capabilities that are required by the overall subsystem to track its progress and verify its deliverables.

We considered the following three aspects of this part of the charge:

3.3.1 Small-scale unit and documentation tests

The current unit test and Continuous Integration (CI) system functions well. However, we suggest a number of improvements. These include clearer instructions to developers on the expectations for testing (discussed in §4.6) and assorted minor upgrades to the CI system, primarily to enable more convenient failure notifications and tighter control of the test environment (§4.8).

The single highest priority we identified was the lack of a current system for performing CI on example code, documentation and (perhaps) Jupyter notebooks. This has already rendered many of the examples provided in the current codebase obsolete — and because they aren't regularly tested, we have no way to know what else is broken and will fail or otherwise confuse new (or existing) users. We regard addressing this as one of the single highest priorities for the project. It is discussed further in §4.7.

3.3.2 Integration tests

DM's integration test needs are currently served by a heterogeneous mix of approaches: from "unit" tests which effectively test the integration of multiple packages, to explicit integration jobs executed by the CI system — of which there are multiple types, and no common implementation standard[8] — to large scale data processing exercises undertaken periodically at the LSST Data Facility (LDF)[9].

We posit that a unified and documented approach to integration testing will provide lower

[8] See lsst_dm_stack_demo, ci_hsc, validate_drp, etc.
[9] e.g. `https://confluence.lsstcorp.org/display/DM/Reprocessing+of+the+HSC+RC2+dataset`.

implementation overheads, fewer surprises for developers, and more predictable coverage for the project. We expand on the considerations and design for such a system in §4.10.

Developing a standardized approach to integration testing requires a standardized approach to the management of test datasets: this is addressed in §4.9. It also impacts upon the CI system, and hence is relevant to §4.8.

Finally, we note that the way in which pipelines are defined and executed will likely evolve rapidly over the remainder of the construction period as technologies like PipelineTask and Butler Generation 3 come into use. Given the rapidly-moving nature of this work, the QAWG regards them as out of scope, except insofar as we urge that QA tasks should be implemented and kept up to date with these new frameworks as they become available.

3.3.3 Metric and performance tracking

DM already has a metric-tracking system based on SQuaSH. This effectively tracks the time evolution of a number of aggregate metrics based on data processing under controlled conditions (effectively, the results of selected integration tests, carried out as per §3.3.2). This basic machinery is excellent, but has suffered from low adoption among DM developers, and it is not clear that any regular checking of or acting upon trends in the calculated metrics is being undertaken.

We therefore propose a series of updates and refinements to SQuaSH, focusing on a series of modest enhancements to its capabilities to alert DM developers and management to regressions (or improvements) in selected performance metrics. These are developed in §4.11.3.

The primary means by which data is submitted to SQuaSH is through the `lsst.verify` framework and associated metric definitions. The mechanisms by which this framework is integrated with the Science Pipelines require further definition (refer to DMTN-057 and DMTN-098 for discussion) and it has never undergone a design review or acceptance process. Clearer ownership and direction for this part of the system are essential to drive adoption. We discuss this in more detail and provide suggestions in §4.11.2.

4 Core components

4.1 Updated pipeline debugging system

This component is derived from §3.1.

The existing pipeline debugging system, `lsstDebug`, provides useful capabilities, but — perhaps due to an idiosyncratic user interface and lack of documentation — is frequently not properly exploited. Since the total amount of code involved in the old system, both implementing it and using it, is modest, we suggest its wholesale replacement by an alternative, more approachable, system.

Recommendation

Develop a new pipeline instrumentation and debugging system, replacing `lsstDebug`.

The total effort expended here should be modest: we expect that design, implementation and documentation of a new system should take no more than one full time equivalent month. Converting existing code make take somewhat longer.

The capabilities of the new system would remain broadly the same as `lsstDebug`. Specifically, enabling debugging mode would set a flag within the codebase. Developers can check for that flag, and take appropriate actions (e.g. spawning additional plots) if it is set. However, establishing a baseline expectation of what actions are appropriate or expected within this debugging context will require input from the leadership team.

Recommendation

Guidelines for the effective use of the pipeline debugging system should be supplied to developers.

These guidelines should be supplied by the Science Pipelines Product Owners and the LSST Software Architect, and made available through the DM Developer Guide.

Developers and users rightly expect that the debugging system will provide an idiomatic user interface. We suggest that the complexity of the existing `lsstDebug` interface has been a bar to adoption. We therefore propose that simplicity be a key aim of the new system. In that

vein, we suggest that the debugging system should be controlled by a single Boolean parameter, with no further user-driven customization, and with very explicit documentation. We acknowledge that this is a direct trade-off between granularity and simplicity, and suggest that history drives us to err on the side of simplicity.

Recommendation

Debugging mode should be binary: it is either enabled or disabled, with no further configuration.
For example, within a `Task`, one might check if debugging mode is enabled with a statement like `if self.config.debug: ...`.

The natural consequence of this is that debugging of tasks invokved by some form of middleware (e.g. `CmdLineTask`, `PipelineTask` called by an executor) should receive an explicit argument enabling debugging. All other callables that support debugging must take an explicit argument that enables it.

4.2 Logging

This component is derived from §3.1.

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.

DM already provides a logging system (the "log" package), and the API documentation provided for it[10] is adequate[11].

However, the outputs of the logging system become increasingly hard to parse as the number of concurrent processes increases: a common complaint when running large-scale data processing is that it's difficult to identify failures and understand where they came from.

[10] `https://developer.lsst.io/stack/logging.html`
[11] It is, perhaps, worth noting that configuring the output of the logging system is much less straightforward and is badly documented: `http://doxygen.lsst.codes/stack/doxygen/x_masterDoxyDoc/log.html#configuration`.

Recommendation

A log aggregation and monitoring service should be provided for large-scale processing jobs at the Data Facility.
Such a service should not be a requirement for jobs to execute (in particular, when running in non-Data Facility environments, logs should continue to be generated and collected as at present).

Log aggregation should provide the following capabilities:

- Display all log messages from a given pipeline execution;
- Display messages at a selected log level (`INFO`, `WARN`, `ERROR`, etc);
- Display time-ordered logs for a given thread;
- Display logs, exit status and/or exceptions for threads ending in an unexpected state;
- Searchable per process for regexp and timestamp.

We believe that these capabilities could be provided by building atop off-the-shelf log aggregation tools such as Logstash[12]. However, we suggest that exploring synergies with existing or planned Data Facility services is likely essential. We emphasize that effective log management capabilities are of increasing importance not just in the operational era but in the immediate future, as we move towards large-scale data processing in support of commissioning: deploying a basic version of this service should be regarded as a high priority.

4.3 Workload management system

Derived from §3.1.

A commonly heard complaint from developers is that the logistics of running code at scale for test purposes is too complex and unreliable. In particular, developers are concerned that sometimes jobs fail to execute without a reason being clearly stated (the job simply disappears from the Slurm queue, without explanation) or that it can be hard to understand from the logs why a job failed or particular output was generated.

[12] `https://www.elastic.co/products/logstash`

The WG reached the conclusion that a wholesale rethinking of workload management is outside the scope of the group's charge. Instead, we suggest just three key improvements: to logging (to better identify and diagnose failures), to provenance (to better understand why certain data products have been produced) and to documentation (primarily, to avoid confusion over why certain jobs might vanish without trace). Logging and provenance are addressed in §4.2 and §4.5 respectively.

Recommendation

Tutorial and reference documentation for developers attempting to run jobs at scale should be refreshed.
In particular, revised documentation should focus on identifying and resolving common failure modes, and understanding how best to use existing resources, such as the dashboard at `https://monitor-ncsa.lsst.org/`, to rapidly diagnose and escalate issues with underlying infrastructure.

4.4 Visualization

Derived from §3.1.

Broadly, we regard "visualization" as an umbrella term covering both visualizations derived from catalogs as well as image display and manipulation. The concerns expressed by developers and others are, on the whole, common to both; the solutions may not be.

For both visualization regimes, the predominant request from developers is that the project provide them with clear guidance as to both what resources will be provided and supported by the LSST construction effort (for example, tools like Firefly or abstractions like `afwDisplay`) and which tools they are required or expected to use in the interest of maintaining a coherent and consistent codebase and set of outputs.

In this section, we concentrate on ad-hoc visualization in support of the regular pipeline developer. In §§3.2 and 4.12, we describe the design of an interactive "drill-down" tool, which will provide a number of plotting and data exploration capabilities. It is our expectation that, however comprehensive such a tool might become, there will always be a necessity for individual developers to be able to quickly investigate their data in as flexible a way as possible; conversely, wherever practical we should enable developers to exploit, and encourage them to remain consistent with, the capabilities and conventions delivered by the drill-down tool.

These sections should therefore be regarded as complementary.

4.4.1 Catalog visualization

The group notes that there are many contexts in which visualizations derived from catalogs might be required (for example, in-line display in a Jupyter notebook, persisting plots from a debugging session, as described in §4.1, preparing plots for publication, etc), which may all have substantially different requirements. We also note that there exists a diverse infrastructure of scientific plotting and data exploration tools in the Python community, a comprehensive selection of which has been collated by the PyViz project[13], which also provides documentation on effectively using them in conjunction with each other. Given that, we regard it as unnecessary, and indeed undesirable, for LSST to attempt to standardize on any particular plotting tool: we should rather encourage developers to exploit community resources with minimal overhead.

Recommendation

DM should formally adopt the PyViz ecosystem.
This adoption would include, for example, including PyViz tools in a regular installation of the LSST Stack; providing training and documentation for developers and — crucially — developing interfaces which enable LSST conventions (afw tables, the Data Butler) to be used in the PyViz context.

Many visualization use cases involve manipulating data at a larger scale than can conveniently be done on a single compute node. Within the PyViz ecosystem, Dask[14] is the preferred approach. We note that there many be some redundancy between Dask and the LSST middleware (Butler, PipelineTask and appropriate executors, etc). However, Dask provides a convenient, easy to install and use solution which can immediately address developer needs.

[13] http://pyviz.org
[14] http://dask.pydata.org/en/latest/

Recommendation

DM should adopt Dask to enable users to work with larger than memory data. This might be achieved by providing users with the ability to spin up Dask clusters on demand using (say) Kubernetes, or by providing a Dask cluster at the LSST Data Facility to which users can connect. If ongoing middleware development renders this obsolete, then it can be retired.

4.4.2 Image visualization

The concern among developers about what they "ought" to be doing is particularly acute when considering image visualization: developers are universally familiar with SAOImage DS9[15], but also aware of other `afwDisplay` back-ends (Ginga[16], Matplotlib[17]), and aware that project resources are being spent on Firefly. This uncertainty is compounded because most developers are unaware of long-term plans for Firefly, and even among system management there is uncertainty about the Firefly development timeline and the extent to which Firefly development is able to accommodate emergent work and requests in response to ongoing development.

Recommendation

DM should provide clear, written guidance to developers about the availability, status and expected usage of image display tools.

We identify two separate regimes of image visualization:

1. A laptop with a small amount of data, where images and other results are inspected with local tools;
2. A server-based environment, like the LSST Science Platform's Notebook Aspect, where centrally provided services have access to large amounts of data.

In the first case, a desktop application like DS9 may be convenient. However, in the second regime, a server-side visualization tool like Firefly provides more convenient access to the

[15] `http://ds9.si.edu/`
[16] `https://ginga.readthedocs.io/`
[17] `https://matplotlib.org`

data: it enables us to bring image display tools to the data, rather than image data to the display.

In addition to the above, we identify three separate ways in which users may wish to use a visualization tool:

1. As a standalone tool (albeit controllable via a Python API);
2. Embedded within a Jupyter Notebook;
3. As a separate "tab" within the JupyterLab environment.

We note that there are a number of LSST-specific requirements on image visualization. These include display of LSST-style image masks, `Footprints`, and other specialized classes, as well as a mechanism for conveniently visualizing the full focal plane.

Previous drafts of this document identified Firefly as the only viable choice for addressing LSST-specific requirements and operating in all of the environments required by LSST. Unfortunately, following the recent changes to the scope of the DM Science User Interface and Tools group (DMTN-096) it no longer seems practical to go "all in" on Firefly: we do not expect Firefly to receive the concentrated attention on usability, responsiveness and LSST specific features which it would require to become a subsystem-wide standard.

With some reluctance, therefore, the working group concludes that for the remainder of construction DM developers will continue to operate with a heterogeneous mixture of visualization tools. We regard this as a significant threat to the overall efficiency of the construction effort, but are unable to suggest an alternative approach within the established schedule, scope and budgetary constraints on the subsystem. We are further concerned about the lack of high-quality visualization tooling available to the commissioning and science validation efforts — and, indeed, about the knock-on effects on what tooling will be available to analyse LSST data during the operational era. However, these concerns are out of scope for this group.

4.5 Provenance

Derived from §§3.1 & 3.2.

An effective provenance system is key to any form of QA work: it is clearly necessary to understand where a particular result came from in order to investigate issues it raises. This is necessary for stand-alone quality analysis, but is fundamental to the proper operation of drill-down and metric-tracking systems. Furthermore, some high level aggregate data products that are derived from provenance data — for example, the number of images that contribute to each coadd patch — are important in QA work.

The QAWG notes that provenance has long been discussed within DM, but detailed plans and timelines have historically been fragmentary[18]. We are concerned that the lack of an effective provenance system is a major barrier to productivity.

Recommendation

The design and implementation of the provenance system should have high priority in the project scheduling.

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.

In the Generation 3 Middleware design, Butler/PipelineTask framework is responsible for recording provenance. For production runs, the Data Backbone Services may collect and store additional provenance information — for example from the Batch Production Service — in addition to that generated by Butler and/or PipelineTask. In the QA use cases, though, we expect the primary source of provenance will be Butler and PipelineTask.

In the Generation 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). Assuming that the recommendation of §4.11.1 is adopted, metric values will be stored as Butler datasets and will have associated Data IDs. This will enable their provenance to be traced in the same way as other datasets.

The QAWG does not place strong requirements on per-source provenance of catalogs and database records. 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 —

[18] We hope the ongoing middleware development effort is remedying this!

such as which input images actually contribute to the measurement of a particular source — but rather enables us to trace the full set of inputs that *could have* contributed to the source. We suggest that this is adequate; if necessary, additional tooling to e.g. investigate the composition of coadds can be added to the drill-down system (§3.2).

4.6 Code quality documentation

Derived from §3.3.

DM values code quality, with an elaborate set of coding standards[19], substantial unit test suites, regular code review, and a set of automatic tests for compliance with code style rules. However, while essential, these are often inconsistently applied, and developers are left confused about what is actually required of them.

4.6.1 Unit tests

The scope of DM's unit test system is not well defined. Tests range from true unit tests—limited in scope to one "unit" of code—to what are effectively integration tests, relying on functionality from many packages working in concert and testing that the results meet some (often apparently arbitrary) numerical threshold. At this stage in the construction project, we do not believe that a wholesale attempt to refactor or reconsider the way that these tests are constructed is plausible, though: we suggest that the current situation is tolerable for the remainder of the project.

However, developers continue to take inconsistent approaches to testing, and disagree (occasionally publicly) about what it is necessary to test, in how much detail, and in what way. We further note that the Developer Guide provides advice which is obsolete and widely ignored[20]. We believe that refreshing the developer-facing documentation, together with closer attention to the form and structure of tests in code reviews (§4.6.2), will pay dividends in terms of reducing confusion and the potential for disagreement.

[19] https://developer.lsst.io/coding/intro.html
[20] https://developer.lsst.io/coding/unit-test-policy.html

Recommendation

Obsolete and unclear sections of the Developer Guide should be rewritten to provide clearer guidance on unit tests.
This should include at least:

- Guidance for unit vs. integration tests, and in which packages it is appropriate to write tests (e.g. is it adequate for certain functionality to be only tested through packages like ci_hsc?);
- Requirements for code coverage;
- Appropriate languages for writing tests (should C++ code be tested in C++, or is it acceptable—or encouraged—to test only the Python-wrapped version?);
- Are there certain types of code that it is appropriate not to test (e.g. boilerplate accessor methods)? How exhaustive should tests be?

4.6.2 Code review

Various aspects of the unit test system, e.g. coverage requirements, are best enforced through code review. However, we currently provide minimal written guidelines to developers about what code reviewers should be insisting on[21].

Recommendation

The Developer Guide should be expanded to provide checklist-style documentation for code reviewers making clear what is expected from them during the review.

4.7 Documentation and examples

Derived from §3.3.

Across the codebase there are scripts, example code, and other utilities. Often these are lurking in `examples` directories in stack packages. Occasionally they are exposed as Jupyter

[21] e.g. "is there adequate unit test coverage for the code?", but with no guidance on what constitutes "adequate".

notebooks, sometimes living in separate repositories. These examples are generally hard to discover; often, they are only made available to new team members (or external third parties) by chance conversation.

Recommendation

Provide a central location where examples, scripts and utilities which are not fundamental to pipeline execution are indexed and made discoverable.
See also DM-15807.

We observe that many of these examples are old, obsolete and often broken. We note that users attempting to run these examples will frequently report that they are found to be non-functional. We further observe that developers are unclear about their obligations for updating these examples when writing new code, an issue which is compounded when the code being changed is in a different package from the affected example. Finally, we regard broken examples as contributing to an actively hostile user experience: no example is better than a misleading or failing one.

Recommendation

The Project should adopt a documented (in the Developer Guide) policy on the maintenance of example code.

Ultimately, *all* code — including examples — should be tested in the CI system: see below. However, pending a mechanism for this, we suggest that:

- Developers are *not* required to search the codebase for examples which may be affected by changes they are making elsewhere;
- When a broken example is discovered, it may be fixed if the changes required are trivial. However, if substantial effort would be needed, the example should simply be removed and an issue filed on Jira to request its reinstatement in future.

Ultimately, we suggest that examples should be tightly integrated with the overall documentation effort.

Recommendation

The Project should prioritize the development of a documentation system which makes it convenient to include code examples and that tests those examples as part of a documentation build.

There are various technologies which could be adopted to address this goal[22]. The WG suggests that standardizing upon a single technology is essential, but takes no position as to which technology is most appropriate.

Finally, we note that the same concerns apply to executable code (in `bin` directories) which is not regularly tested in CI.

4.8 CI system updates

Derived from §3.3.

DM's "Jenkins" CI system provides a number of essential services to the Project and is a core part of developer workflow. However, the QAWG suggests that a relatively modest set of changes could dramatically improve its usefulness to developers and to the project overall.

4.8.1 Notifications and failures

Most importantly, the QAWG suggests that Jenkins should be more helpful in enabling rapid response to build failures.

[22] For example, Jupyter notebooks or Sphinx doctests.

Recommendation

When running regularly scheduled (timer) jobs on the `master` branch of any releasable product, any build failure should be announced prominently to key stakeholders.
Those stakeholders should include:

- Senior DM management (DM Project Manager, DM Software Architect, Science Pipelines T/CAMs and Science Leads);
- The developer who caused the failure, if it is possible to identify them

The term "prominent" should be taken to indicate a personalised message (e.g. e-mail, direct Slack message), not a general posting in a Slack channel which regularly sees traffic.

Recommendation

The Developer Guide should provide guidance about expected responses to Jenkins failures.
For example, the policy for handling integration test (e.g. ci_hsc) failures a merge must be documented: who is responsible for checking for failure? Should the merge be reverted? Who is responsible for doing so?

4.8.2 Execution environment

DM requires a range of external packages (NumPy, SciPy, matplotlib, etc). Versioning constraints on these are checked by "stub" packages at configuration time. Typically, the only check applied is that the available version exceeds some minimum: no maximum versioning constraint is enforced. This exposes us to possible API breakage in newer versions of upstream packages. We do not regard this as requiring urgent intervention, but the team may wish to consider enforcing version maximums at some point in the future.

For maximum coverage, all tests should be run with all possible combinations of external packages. In practice, of course, this is prohibitive. However, it is essential that the code be shown to work with the minimum versions of all packages documented as required.

Recommendation

The versions of external packages used in the Jenkins system must always correspond to the minimum versions specified in stub packages and/or in the document list of prerequisites.

We note that the details of this recommendation may be subject to alteration, depending on changes to the DM release process and the way it interacts with third party dependencies. However, the spirit of the recommendation — that our code must be checked with the library versions it claims to support — should stand in any eventuality.

Recommendation

The project should adopt a single source of dependency information and versions.
This might consist of "stub" EUPS packages, or of a Conda environment, or of a list of packages on a website, but there must be *one* unambiguously authoritative source of information.

4.9 Standard format dataset packages

Derived from §3.3.

The DM subsystem curates a number of "datasets": collections of data related by some underlying theme or use case. These themes might include originating from the same instrument or facility (e.g. testdata_cfht); being used to test a particular package (e.g. testdata_jointcal); or addressing some particular science case (e.g. ap_verify_hits2015).

Currently, DM's datasets are heterogeneous: there is no accepted standard for the type of data that a dataset should contain, and nor is there any standard for where the dataset is stored or how it is curated. Some of DM's datasets are stored on GPFS at the LSST Data Facility; some are made available through Git LFS[23]; some contain only raw data; some contain calibration data; some contain processed data; some are regularly updated; some have documentation describing in detail what the dataset contains.

This lack of standardization limits the reuse of datasets (it is hard to reuse a dataset curated

[23] `https://git-lfs.github.com`; `https://developer.lsst.io/git/git-lfs.html`

for one purpose for some other test unless one fully understands its contents) and means that developers often struggle to find the most appropriate data to test or debug some particular algorithm.

Recommendation

A standardized format for dataset repositories should be adopted across DM. Obviously, not all repositories will have exactly the same contents: in some cases, it may be necessary for a repository to contain (say) calibration products, while in others they may be inappropriate. However, it should be possible to establish at a glance what the contents of each dataset is; if calibration products *are* included, it should be immediately obvious what they are and how to apply them.

The key desiderata for standardizing the format of test datasets are:

- 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 must be well understood.
- Besides raw (unprocessed) data, intermediate and final data products from various stages of pipeline processing are desired. 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 (although older ones may also be useful to test backwards compatibility).

These considerations aside, this WG does not make a specific recommendation about the detailed layout of datasets, beyond noting the existence of the Common Dataset Organization and Policy [24] which might form a convenient basis for further development.

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

Regardless of the layout adopted, we expect that the dataset will need to evolve with time: as new versions of the LSST code are released, expectations about both the contents of data repositories and the format of persisted data products will change, occasionally in backwards-incompatible ways. For a dataset to remain useful, active curation is required.

Recommendation

Each dataset should have an explicitly named product owner.
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 could often be a member of the team with immediate use cases and knowledge of the relevant camera package.

The variety of different sources and use-cases for datasets means that they span a wide range of sizes. It is therefore impractical to store and distribute them all in the same way. Instead, the QAWG suggests they can usefully be divided into two categories: *small* and *large*.

Small datasets are those smaller than 100 GB in total size. They are intended for use, for example:

- as input test data in CI jobs;
- as example data in documentation, demos, and tutorials.

We recommend that they are:

- packaged as EUPS products;
- made publicly available on GitHub;
- stored as Git LFS repositories as needed;
- versioned with DM software releases.

Large datasets are bigger than 100 GB in total size. They are intended for use, for example:

- in large scale integration tests;
- in flushing out edge cases which may not be apparent from smaller data volumes;
- for archiving outputs from engineering facilities like the Camera test stands.

We recommend that they are:

- made available on project-provided shared network filesystems (i.e. GPFS);
- protected under a disaster recovery policy.

Recommendation

Datasets may be stored on either shared filesystems or Git LFS as appropriate, depending on the total size of the dataset.

4.10 Standard format test packages

Derived from §3.3.

Analogously to various inconsistent forms of packaging datasets described in §4.9, we also have a variety of different styles of packages which are fundamentally designed to perform some form of integration testing: that is, executing a test suite which exercises multiple constituent parts of the DM simultaneously. Broadly, we consider that these fall into one of two categories:

1. Scons-based execution, including ci_hsc and ci_ctio0m9; or
2. Shell-script based, including validate_drp and ap_verify[25].

Often packages of this type are designed for automatic execution within the Jenkins environment, typically on nightly timers, but we note that:

- Even for packages designed with Jenkins in mind, being able to run standalone is still generally essential;
- We expect future integration tests to involve executing large-scale jobs which will run on cluster-scale hardware at the Data Facility. Such jobs may be coordinated by Jenkins, but will (likely) not run entirely on Jenkins build agents.

We consider that broadly the same benefits to adopting a consistent structure in these packages apply here as in §4.9. In particular, adopting a standardized design would enable:

[25] But note that this shell-script entry point may cover a variety of lower-level implementation details, such as LSST-standard `CmdLineTasks`, Python scripts, or further shell scripting.

- Developers to make use of existing tests when developing new algorithms with minimal overhead;
- New tests to be added without significant design work.

Recommendation

A standardized test package design should be developed which addresses all existing use cases.
Existing test packages (lsst_dm_stack_demo, ci_hsc, validate_drp, ap_verify, etc) should be adapted to the new design, and, where possible, merged with each other.

Finally, we suggest that the existing variety of test packages reflects a lack of clarity and orthogonalization as to exactly what functionality can and should be tested in CI and on what cadences.

Recommendation

A coherent plan for integration testing at all scales should be developed and published.
Such a plan should then drive the development of the test package standard discussed above. Note DM-15348 in this context.

4.11 Metric collection & tracking

This section is derived from §§3.2 and 3.3.3.

We note that metrics describing pipeline execution may be considered in two separate — but related — contexts. In some cases, we care simply about the absolute value of some metric: is the performance "good enough"? Does it satisfy a requirement? In other circumstances, we might wish to keep track of a metric value as a time series: does performance change with time? Can we identify changesets which have introduced regressions (or improvements!)?

We note further that metrics values might be calculated in a number of different contexts. For example:

- Some metrics refer to characteristics of pipeline execution, like execution time, which *by definition* must be recorded during execution or they are lost[26].
- Others may only be calculated from intermediate data products, which would not normally be stored for posterity. These must be calculated before those intermediate products are removed.
- Finally, some are calculated from final science data products, and hence may be calculated at any time after pipeline execution.

4.11.1 Metric definition and calculation

The author of pipeline code can, of course, simply print to screen (or to log) a message containing some quantity that they have calculated on the fly during the execution of their code. This is low overhead and trivial to implement. It may be combined with the debugging system (§4.1) to provide a rich set of diagnostics when the code is executing. We suggest that there is no advantage to attempting to force some new framework onto developers operating in this mode.

However, at the level of long-term monitoring of pipeline performance, and especially at the level of requirements verification, we suggest that having a standardized, code-based definition of metrics is essential to enable clear and unambiguous comparison of results. The SQuaRE team has developed the `lsst.verify.metrics`[27] system to facilitate the centralised definition of metrics, which we regard as a key step in the right direction.

Recommendation

Formalise the `lsst.verify.metrics` system as the source of truth for metric definitions, by e.g. describing it in LDM-503 and LDM-639.
This should **not** be taken as a blanket endorsement of the current implementation of this system; §4.11.1 provides a number of recommendations as to the way that metrics are defined.

[26] Of course, it is not a requirement that they by recorded by a dedicated metric tracking system; one could imagine recording execution time by simply recording a log message, and then later parsing log outputs to retrieve it

[27] `https://github.com/lsst/verify_metrics`

Recommendation

Provide a high-level overview and data-model describing the metric definition system.

At present, the lack of material providing an overview of the system is a significant barrier to entry. Although excellent API documentation is available, it is accompanied by references to technical notes describiing relatively vaguely-specified design goals and referring to concepts like "provenance" and "specification" which require further elucidation. A revision of the documentation which expunges mention of old, obsolete packages[a] and provides a clear set of gettting-started guidelines should be untertaken. We suggest that that the SQuaRE group engage with one or more stakeholders in the Pipelines while working on this material.

[a] e.g. `lsst.validate.base`

Many metrics are defined as statistical aggregates of a per-source computed quantity — for example, FWHM or shape — 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 separate. For metrics computed in this way, the maximum storage granularity should be considered to be at the source level.

Recommendation

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

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 explicitly 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 and may be included in the relevant object catalogs or in post-processed tables; in others, the minimum granularity may be at the CCD, patch, or even dataset levels. This procedure provides the analyst the ability to filter metric values using arbitrary metadata (night, airmass, focal plane position, moon phase, etc) and re-aggregate to any level desired with any aggregation function (mean, median, percentiles, standard deviation, outliers, etc). 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.

We note that this procedure will result in a substantial increase in catalog volume. We suggest that metric value storage therefore be configurable through the regular pipeline configuration system. When working independently, developers may enable or disable at will depending on their use case; when running integration or other formalized tests, metrics will be stored for later analysis; metric storage when performing alert or data release processing is an operational decision.

"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.

Recommendation

Metric values should be stored as "tidy data" in columnar data stores (e.g., Apache Parquet) as part an output data repository.
In particular, this should make it possible to load the data quickly enough for interactive work on hundreds of tracts or an equivalent number of visits.

Additionally, we 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 tens to hundreds 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 I/O of small files when loading multiple tracts of data at once.

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

Recommendation

Metric values should have Butler dataIds.

As discussed in §4.11, the appropriate time to calculate a metric value may depend on the nature of the metric. Some are calculated and stored in real-time as pipeline execution continues; others may be handled by "afterburner" analysis of existing data repositories. In either case, we suggest that close integration with the Data Butler is essential, and we believe the scheme presented here is compatible with the design of the "Generation Three" Butler.

Recommendation

It should be possible to use the Data Butler to persist and retrieve metric values.

4.11.2 Collecting metrics

The `lsst.verify` package enables the conveneient packaging of `lsst.verify.metrics`-style metrics and their submission to the SQuaSH metric tracking dashboard (§4.11.3). We are concerned that the adoption of this system has been slow. In part, we expect this to be addressed by impreovements to the definition of metrics, as discussed in §4.11.1. However, we also believe that it is now appopriate for Pipelines leadership to begin to actively engage with this system.

Recommendation

Develop clear guidelines for integrating metric collection with pipeline code. DMTN-057 suggested a number of ways in which this might be done; DMTN-098 and related work has begun development of a specific approach, which may evolve into an accepted standard.

Recommendation

Pipelines leadership should start using the metric definition and collection system.
As the above recommendations are met, this system will be usable. However, driving adoption will require proactive measures from pipelines Product Owners and T/CAMs.

4.11.3 Metric tracking: dashboard and alerts

The SQuaSH system, SQR-009, has been developed by the SQuaRE team to provide "dashboard" functionality for metric tracking. At its core, SQuaSH provides a database to which metric values may be submitted using the `lsst.verify` (§4.11.2) system, and a web-based service for displaying metric values as a time series. This enables the user to track the evolution of metric values with time, and relate them directly to changes in code or configuration. SQuaSH has also been designed to provide some limited "drill-down" functionality to explore the way in which high-level metrics have been calculated.

To date, SQuaSH has been used to follow a set of metrics derived from high-level LSST requirements and codified in the validate_drp package. It has been designed, though, to enable use by individual developers to track metrics which are of interest only to them, or relevant to a particular subset of the codebase on which they are working.

The Working Group feels that the major value in the SQuaSH system is in tracking and responding to regressions in performance (be they scientific or run-time) as the code evolves. In this respect, it is in some ways analogous to the CI system (§4.8), and benefits from many of the same recommendations.

Recommendation

SQuaSH should issue alerts to developers and key stakeholders on regressions in important metric values.
Key stakeholders should include:

- Senior DM management (DM Project Manager, DM Subsystem Scientist, Pipelines Scientist, Science Pipelines T/CAMs and Science Leads);
- The developer who caused the regression, if it is possible to identify them (e.g. through commit logs).

This will require careful design, as it may be in tension with the desire to enable developers to define arbitrary metrics for their own use: clearly, key stakeholders will not wish to be informed of every developer-defined metric which suffers a regression. We suggest that, for example, a "subscription list" for each metric be defined, and the key stakeholders automatically be added to it for all metrics deriving directly from high-level requirements.

As with `lsst.verify`, we worry that there is confusion about how to distinguish metric values measured on different versions of the codebase, configurations, datasets, etc within the SQuaSH system. For example, it is possible to define and track a metric on an algorithm implemented by code in the `lsst.pipe.tasks` package. But that code may be run in multiple different contexts: as part of alert production, data release production, precovery, etc: how does SQuaSH distinguish between all of these environments? We believe that SQuaSH is capable of this, but existing "big picture" documentation is lacking and hard to follow.

Recommendation

Provide a single, reliable source of documentation describing the SQuaSH system and a vision for its use in DM-wide metric tracking.

We note that SQuaSH provides some drill-down capability to explore the source of metric values. We suggest that this should not be the core business of SQuaSH, and prefer to consider a separate drill-down environment (§4.12); further development of these capabilities within

SQuaSH should not be prioritised. However, if the metric submission system captures appropriate information, it would be desirable to enable SQuaSH to automatically spawn a copy of the drill-down system pointing at the repository corresponding to a particular metric value submission.

Recommendation

The SQuaSH system should be closely coupled to the drill-down environment; in particular, the former should use the latter to enable drill-down functionality into particular metric values..

We recognize that developers may wish to submit results to SQuaSH from a variety of systems. In particular, it must not be dependent upon a particular execution environment (e.g. Jenkins).

Recommendation

It must be possible to submit metrics to SQuaSH from arbitrary pipeline execution environments.

When handling large datasets, there is value to tracking metric values computed over subsets of the whole. For example, it may be more relevant to track how photometric repeatability varies over some patch, rather than over the whole sky.

Recommendation

SQuaSH should be able to store and display appropriate metric values per DataId.
For example, CCD, visit, patch, tract, filter.

4.12 Drill-down workflows

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 (§4.11.3).

Recommendation

DM should develop 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 summary of top-level global metrics (Fig. 1), and and a metric view showing more information on a selected metric (or set of metrics; Fig. 2). The more detailed metric dashboard should be able to explore both coadds and individual visits.

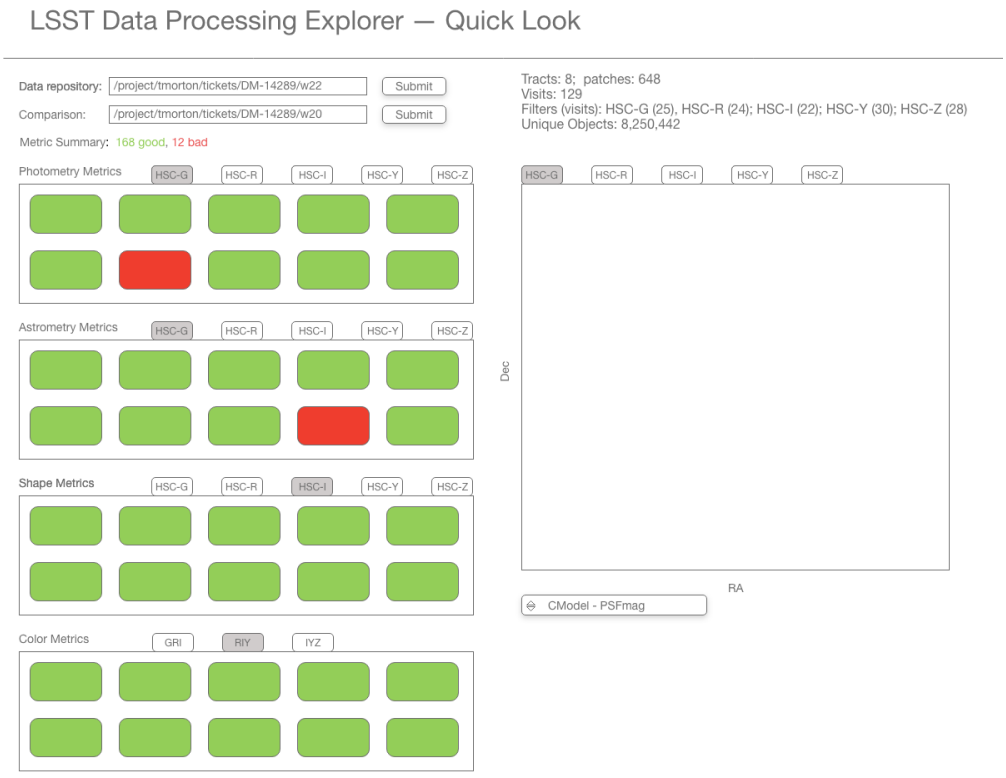

FIGURE 1: Mock-up of the "quick look" overview screen of the drill-down system. This provides a summary of all metrics calculated over the results of a particular pipeline execution, indicating any that fell below threshold.

A developer will load the dashboard by visiting a particular URL. They will then enter the path to a data repository[28] which contains metric values. They will be presented with the "quick look" screen as shown in Fig. 1 which summarizes the results of the data processing.

The dashboard will generate summary information on the fly by introspecting the repository. It follows that, in addition to the metric values themselves, the repository will also contain metadata that describes:

- 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 will have a choice to explore either the metrics in a single repository, or the differences in metrics between two data repositories by entering 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 an at-a-glance summary of the data and identification of any problems. The overall data summary could be in a header displaying the 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 minimalist 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).

Selecting any metric from the top-level page should load up a metric-detail page, as shown in Fig. 2, 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 the

[28] On some accessible filesystem; this assumes that the QA system is running e.g. on a host at the LSST Data Facility with access to `/scratch` or `/project`

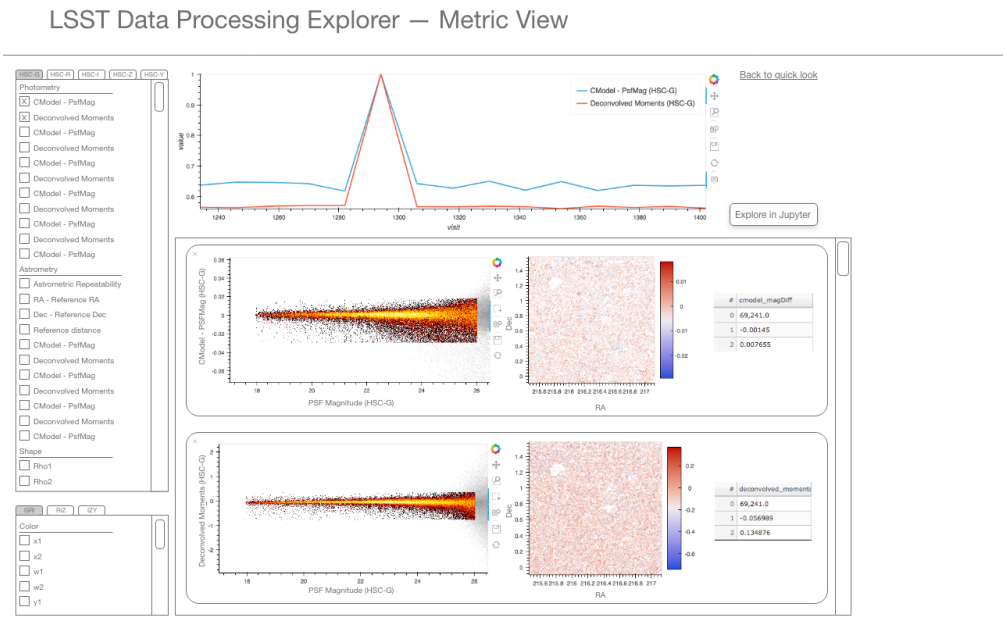

FIGURE 2: Mock-up of the "metric view" showing more information on how a particular metric value was calculated.

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 simultaneous 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.

5 Conclusion

This document has described the deliberations and conclusions of the QA Working Group. It has taken a wide-ranging view over various aspects of the DM Subsystem, and presented a wide range of recommendations, which are summarised in Appendix A. Many of these recommendations are evolutionary improvements to existing DM tools, practices or documentation. A few involve the development of new capabilities. Of particular note in this latter capability are the call for the development of a integrated drill-down system, described in §4.12, and for adoption of the Dask system §4.4[29]. These capabilities will require significant resources to deliver, and will therefore require action by DM Project Management. However, we also commend to management some of the lower-profile recommendations: in particular, we feel that modest improvements to dataset organization and to the CI system could have major impacts on DM's overall productivity.

A Recommendations

QAWG-REC-1: Adopt the definitions of QA-related terms in the DMTN-085 glossary subsystem-wide (§1)

QAWG-REC-2: Develop a new pipeline instrumentation and debugging system, replacing `lsstDebug` (§4.1)

QAWG-REC-3: Guidelines for the effective use of the pipeline debugging system should be supplied to developers (§4.1)

[29]We note that, at time of writing, some work involving Dask is now underway within DM, although we are not aware of design documentation describing exactly what capabilities will be provided.

QAWG-REC-4: Debugging mode should be binary: it is either enabled or disabled, with no further configuration (§4.1)

QAWG-REC-5: A log aggregation and monitoring service should be provided for large-scale processing jobs at the Data Facility (§4.2)

QAWG-REC-6: Tutorial and reference documentation for developers attempting to run jobs at scale should be refreshed (§4.3)

QAWG-REC-7: DM should formally adopt the PyViz ecosystem (§4.4.1)

QAWG-REC-8: DM should adopt Dask to enable users to work with larger than memory data (§4.4.1)

QAWG-REC-9: DM should provide clear, written guidance to developers about the availability, status and expected usage of image display tools (§4.4.2)

QAWG-REC-10: The design and implementation of the provenance system should have high priority in the project scheduling (§4.5)

QAWG-REC-11: Obsolete and unclear sections of the Developer Guide should be rewritten to provide clearer guidance on unit tests (§4.6.1)

QAWG-REC-12: The Developer Guide should be expanded to provide checklist-style documentation for code reviewers making clear what is expected from them during the review (§4.6.2)

QAWG-REC-13: Provide a central location where examples, scripts and utilities which are not fundamental to pipeline execution are indexed and made discoverable (§4.7)

QAWG-REC-14: The Project should adopt a documented (in the Developer Guide) policy on the maintenance of example code (§4.7)

QAWG-REC-15: The Project should prioritize the development of a documentation system which makes it convenient to include code examples and that tests those examples as part of a documentation build (§4.7)

QAWG-REC-16: When running regularly scheduled (timer) jobs on the `master` branch of any releasable product, any build failure should be announced prominently to key stakeholders (§4.8.1)

QAWG-REC-17: The Developer Guide should provide guidance about expected responses to Jenkins failures (§4.8.1)

QAWG-REC-18: The versions of external packages used in the Jenkins system must always correspond to the minimum versions specified in stub packages and/or in the document list of prerequisites (§4.8.2)

QAWG-REC-19: The project should adopt a single source of dependency information and versions (§4.8.2)

QAWG-REC-20: A standardized format for dataset repositories should be adopted across DM (§4.9)

QAWG-REC-21: Each dataset should have an explicitly named product owner (§4.9)

QAWG-REC-22: Datasets may be stored on either shared filesystems or Git LFS as appropriate, depending on the total size of the dataset (§4.9)

QAWG-REC-23: A standardized test package design should be developed which addresses all existing use cases (§4.10)

QAWG-REC-24: A coherent plan for integration testing at all scales should be developed and published (§4.10)

QAWG-REC-25: Formalise the `lsst.verify.metrics` system as the source of truth for metric definitions, by e.g. describing it in LDM-503 and LDM-639 (§4.11.1)

QAWG-REC-26: Provide a high-level overview and data-model describing the metric definition system (§4.11.1)

QAWG-REC-27: The computation, selection, and aggregation steps that define a metric should be well compartmentalized (§4.11.1)

QAWG-REC-28: Metric values should be stored with full granularity (source, CCD, patch, dataset) (§4.11.1)

QAWG-REC-29: Metric values should be stored as "tidy data" in columnar data stores (e.g., Apache Parquet) as part an output data repository (§4.11.1)

QAWG-REC-30: Metric values should have Butler dataIds (§4.11.1)

QAWG-REC-31: It should be possible to use the Data Butler to persist and retrieve metric values (§4.11.1)

QAWG-REC-32: Develop clear guidelines for integrating metric collection with pipeline code (§4.11.2)

QAWG-REC-33: Pipelines leadership should start using the metric definition and collection system (§4.11.2)

QAWG-REC-34: SQuaSH should issue alerts to developers and key stakeholders on regressions in important metric values (§4.11.3)

QAWG-REC-35: Provide a single, reliable source of documentation describing the SQuaSH system and a vision for its use in DM-wide metric tracking (§4.11.3)

QAWG-REC-36: The SQuaSH system should be closely coupled to the drill-down environment; in particular, the former should use the latter to enable drill-down functionality into particular metric values. (§4.11.3)

QAWG-REC-37: It must be possible to submit metrics to SQuaSH from arbitrary pipeline execution environments (§4.11.3)

QAWG-REC-38: SQuaSH should be able to store and display appropriate metric values per DataId (§4.11.3)

QAWG-REC-39: DM should develop 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 (§4.12)

QAWG-REC-40: The dashboard should enable the analyst to start a Jupyter notebook session with the relevant datasets already loaded (§4.12)

B 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.

Apache Parquet A columnar storage data persistence format maintained by the Apache project; `http://parquet.apache.org`.

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.

HSC Hyper Suprime-Cam.

KPM Key Performance Metric.

LDF LSST Data Facility.

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.

provenance A description of the inputs and processes which have been used to generate a particular result or data product.

QA Quality Assurance. For the purposes of this document, we take QA to describe all activities, deliverables, services, documents, procedures or artifacts which are designed to ensure the quality of DM deliverables. This may include QC systems, in so far as they are covered in the charge described in §1 and LDM-622. Note that contrasts with the LDM-522 definition of "QA" as "Quality Analysis", a manual process which occurs only during commissioning and operations.

QAWG QA Strategy Working Group.

QC Quality Control. Following LDM-522, QC describes services and processes which are aimed at measuring and monitoring a system to verify and characterize its performance. QC systems run autonomously, only notifying people when an anomaly has been detected. Contrast QA.

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

[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

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

[LDM-522], Economou, F., Wood-Vasey, M., 2017, *DM Science Quality Data Assurance System Conceptual Design*, LDM-522, URL https://ls.st/LDM-522

[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.

[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

[DMTN-098], Findeisen, K., 2018, *Metrics Measurement Framework Design*, DMTN-098, URL https://dmtn-098.lsst.io,
LSST Data Management Technical Note

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

[LDM-639], Guy, L., 2018, *DM Acceptance Test Specification*, LDM-639, URL https://ls.st/LDM-639

LSST Data Management, LSST DM Developer Guide, URL https://developer.lsst.io/

[DMTN-096], O'Mullane, W., Swinbank, J., Guy, L., 2018, *Implementation and impacts of DM scope options*, DMTN-096, URL https://dmtn-096.lsst.io,
LSST Data Management Technical Note

[LDM-503], O'Mullane, W., Swinbank, J., Jurić, M., Economou, F., 2018, *Data Management Test Plan*, LDM-503, URL https://ls.st/LDM-503

[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