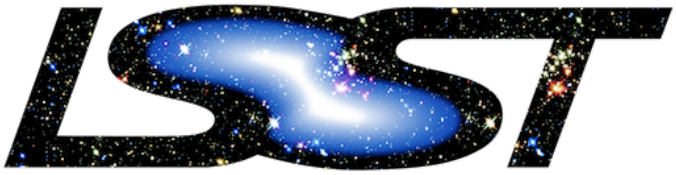

LARGE SYNOPTIC SURVEY TELESCOPE

## Large Synoptic Survey Telescope (LSST)
## Data Management

# DM Release Process

**Gabriele Comoretto**

**DMTN-106**

**Latest Revision: 2019-05-31**

**DRAFT**

## Abstract

Release procedure applicable to all Data Management SW products.

# Change Record

| Version | Date | Description | Owner name |
|---|---|---|---|
| | 2019-02-04 | DM Release Process | Gabriele Comoretto |

*Document source location:* `https://github.com/lsst-dm/dmtn-106`
*Version from source repository:* 5026f09

# Contents

# DM Release Process

## 1 Introduction

This release procedure is general and valid for all software products in the LSST Data Management subsystem. It is an abstraction of SQR-016, but it is assumed that the Jenkins builds used for the Science Pipelines can be applied in the same way to other software products.

The release manager should be in charge of the release activities documented here. All developers are not affected, except for backporting of fixes into release branches, when required.

Section A summarizes the actual process and reports the derived problems that affect it.

### 1.1 Applicable Documents

LDM-148 DM Architecture
LDM-294 DM Project Management Plan
LDM-672 LSST Software Release Management Policy

# 2 Definitions

This document uses certain terms which are often overloaded. In this section, we define these terms as intended in this document.

## 2.1 Environment

An environment is a set of libraries, executables, and configurations, that are predefined for a specific context or function.

An environment is usually needed in the following contexts:

- software development: the environment includes all libraries and tools required to build and debug a software product
- software test and verification: the environment includes, on top of the build and debug tools, additional tools to permit/facilitate the testing and validation activities.
- operations: the environment shall include only tools required for the execution of an operational product. It shall not contain any build or debug tools. It shall be optimized for the operational activities. [1]

These are the most common uses of an environment. Additional scenarios can be identified.

In some cases, the same environment is used in development, tests, and operations.

Operational environment definitions, including location-specific configuration, shall never be included in the software products. Vanilla environment definition can be provided for unit tests purposes, as an example or template.

## 2.2 Software Product

A SW product is a *defined* and *controlled* software implementation with a specific scope, inside an overall system.

- *defined*: it shall be clear which is the scope of the software product

[1] In case a problem is found in operation the first thing to do is to replicate it in a different environment, where also debug and further analysis is possible.

- *controlled*: the source code shall be maintained in a software repository, like for example GitHub.

The following activities are done on a SW Product:

- development activities
- test activities, usually unit tests and system test
- release activities
- packaging activities
- operational activities, in standalone mode, in an operational pipeline or as part of a service. Some SW Products are libraries, therefore are not operationally but used as dependencies by other software products.

A software product should correspond to a single repository.

In the case that a SW product is comprised of multiple Git repositories, each Git repository shall be related only to one software product. [2] In other words, there should be a correspondence *1:1* (preferably) or *1:many*, between software products and Git repositories. There shall never be a *many:many* correspondence.

The following elements should not be part of a software product, but follow a parallel development/release process:

- build tools, otherwise we will have them deployed in operations.
- environment definition (2.1) in general, with the exception of vanilla or example definitions.

Big test datasets should also not be included in a software product.

The content of this subsection is a *prerequisite* for any release procedure to be applied.

[2] In DM, a software product is comprised of multiple Git repositories.

## 2.3 Binary Packages

A binary package is a package containing executable and/or libraries (binaries or not). It is obtained by building the software provided in the repository (SHA1 or tag) and creating a package with the build outcome. It may contains also supporting scripts, documentation and other files provided but not compiled, or just required to run the software product. Binary packages can be created to support multiple platforms, such as Linux, macOS, Windows.

The *recipe* to build the binary package should be included in the software repository. If this is not done, it shall be versioned separately, and this may generate process overhead.

Binaries packages for releases should be generated only once and made available for their downstream use by dependent software or for deployment. In some cases, the binary package may need to be rebuilt if a dependency changes, without changing the software product tag or SHA1. This can happen for example when *Semantic Versioning* is applicable or the version of the dependencies spans over a range.

It should be possible also to identify the binary package with its checksum. That checksum can be used to identify which package is installed in an environment, or included in a distribution, and therefore the corresponding release.

## 2.4 Dependencies

As defined in the above section 2.2, each software product depends, during build and runtime, on other software products. The case where a software product has no dependencies is not common. [3]

The software product dependencies can be divided into two main categories:

- other software products developed inside the same project. These dependencies are developed and released on their own, following the same development and release process. Their source code is not part of the software product. The dependency information is part of the software product, in a specific configuration file, therefore versioned altogether.
- third party packages, developed by a team that is not part of the project. These packages follow a completely different development and release process. They are not part

[3] Operational dependencies or data flow dependencies are not taken into account in this contest.

of the software product. The dependency definition should, in general, be part of the environment definition, and therefore also not be part of the software product.

Both categories should be resolved from the binary package repository, ready to use, and not checkout and rebuilt from the source code each time. This is especially true for the third-party packages and shall be enforced when releases are done.

## 2.5 Software Release

A software release is a consciously identified tag of a software product repository, documented with a software release note.

A software release has been proved to work before the release tag is made in the repository. This ensures that a release meets the functionalities and fixes that are documented in the release note.

A software product release shall depend only from other officially released software products.

The tag in the Github repository and the software release note (3.2) are sufficient to identify the release, and therefore for a developer to resolve the dependencies, build the binaries and execute the software.

The software release is a monolithic snapshot of the software product to be used as is by the downstream processes or users.

## 2.6 Distribution

A distribution is a collection of objects to be deployed together. Those objects can be binary packages (2.3), third-party packages, data, source repository, tools, etc.

The distribution configuration files shall be versioned and released as it is done for the software products.

A distribution can be used for different purposes:

- to make available software releases for operations
- to test (integration, validation, operation rehearsals) software releases

- to provide software releases to external collaborators.

Distributions can be handled in different ways. Git metapackages, like *lsst_distrib*, or Docker images, are just a couple of examples.

The main DM software product currently being distributed is the Science Pipeline Stack - this may be distributed in a few ways, see `https://pipelines.lsst.io/install/index.html`.

# 3 Change control

The DMCCB is in charge of approving the changes which go in all releases (major, minor, or patch). The DM Release Plan LDM-564 provides the expected schedule for major features, based on the P6 milestones [4].

Changes to the release plan thus are at least partially controlled by the LSST change control process as milestones are moved or impacted by other changing milestones. Other changes not covered by the milestones such as features or content need to be proposed to the DMCCB using RFC Jira issues.

Patch release requests should use an RFC Jira issue which must be approved by the DMCCB, as per LDM-294 The issue should specify what needs to be fixed in the release and why it shall specify DM issues that are requested to be included in the patch.

This process is documented in LDM-294, sections 3.6 and 7.4.

## 3.1 Issue Management

The release is identified in Jira using a release issue.

All issues to be included in a release shall be added as blocking to the release issue.

Note that, introducing the field *Fix in Version(s)*, will permit having the same information in a much simpler way. Both release issue and *Fix in Version(s)* fields can be used in parallel. Once the release is done, it will also be possible to complete the field *Fix in Version(s)* with the exact release, ensuring in this way that each issue is fully documented and self consistent[5].

## 3.2 Release Note

The release note is the document that identifies the SW (tag) under release in a human-readable language, easy to access and share. It may contain technical details and/or narrative that fully characterize the release, and that may not always be available in the software repository.

The following information should be provided:

[4] The DM Release Plan is in the process to be updated, see DM-17001
[5] process to be automated

- The installation instructions. This is usually a manually written set of instructions.
- A narrative section describing the content of the release, this summary should be written by a person not auto-generated. The T/CAM of the product being released is responsible for making sure this is provided.
- The list of Jira issues included in a release. This information can be extracted from Github. Completed epics will be highlighted while other issues will be listed to be comprehensive.
- Technical information like the GitHub tag, dependencies, etc. can be extracted automatically from Github or other tools.

In addition, a list of implemented functionalities, requirements or completed milestones can be provided.

### 3.3 Versioning and Naming

DM will switch to Semantic Versioning, specifically Semver 2.0.0. [6]. This will help improve dependency management and make managing patch releases more comprehensible. The versioning schema will have 3 digits separated by a "." . For example :
16.0.0

Generalizing, the versioning scheme is the following:

$M.n.p$

where

- **M** is the major version identifier,
- **n** is the minor version identifier,
- **p** is the patch level identifier

The repository at this stage would be tagged 16.0.0 . One should note in this scheme there are no padding zeros thus:
$16.1.0 \neq 16.10.0$

[6] https://semver.org/spec/v2.0.0.html

DM software is currently versioned using two digits separated by a "." . For example Science Pipelines latest release:
17.0.1

### 3.3.1 Branch Naming

In order to clearly distinguish release branches from tags and other branches, the following naming scheme is suggested:

```
b.<M>.<n>.x
```

where **M** is the major version and **n** is the minor version starting from 0.

The letter **b** indicates that this is a branch. The letter **x** is a placeholder. It indicates that versions in a specific branch will have a fixed **M.n** but different patch level, starting from **1**.

### 3.3.2 Binary Packages Build Naming

As mentioned above in 2.3, binary packages may be generated multiple times for the same release. Therefore the build number is required, in order to uniquely identify each different build.

Assuming that the build tools are able to provide a unique build number each time a build is done, each binary package build will be identified as follows:

```
<M>.<n>.<p>-<bID>
```

where **M**, **n** and **p** have the same meaning as explained in the above subsection 3.3, and **bID** is the unique build number provided by the build tool.

# 4 Software Release Procedure

This release procedure is a generalization of the *Stack release playbook* (SQR-016).

## 4.1 Development

Development activities are not part of the release process but are the starting point for a stable and reliable master branch in all Github software packages. The Developer Guide lays out the guidelines and process. All changes are done on ticket branches and reviewed using the Pull Request mechanism before merging to master.

Each time a change is merged into master, the following activities should be performed:

- continuous integration build of the Git repository (software product)
- if unit tests pass, generate binary packages
- build downstream dependencies: CI build on SW products that depend on the newly build SW product.

Continuous integration is done also on a ticket branch prior to merging changes to master.

Binary packages (EUPS) and docker images for distribution are made available with the nightly and weekly builds (see next section 4.2).

## 4.2 Nightly and Weekly builds

Nightly and weekly builds are useful for a number of reasons such as :

- Maintaining a healthy code base.
- Forcing us to maintains build scripts.
- Finding breaking changes which got checked in and passed CI.
- A starting point for a development activity.
- A starting point for a release.

Nightly builds do not generate tags in the Github repositories. When a weekly build is done, a corresponding tag in the Github repository is created. Though weekly builds are considered releases in Github, from a release management point of view they are not releases - they are reference builds and, when they meet the criteria in 4.4, release candidate starting points.

## 4.3 Announcement

The preparation of the next release is announced using a community post. This has to be done a few days before the release activity starts.

In this way, all contributors will be able to provide feedback, for example, additional issues expected to be included in the release. Based on the feedback, DMCCB can take corrective actions such as delaying the release.

## 4.4 First Release Candidate

The first release candidate on a release is created when:

- all issues that are supposed to be included in it have been implemented and merged into master.
- a weekly build has been completed successfully.
- the product owner confirms that the weekly is good to go. [7]

In case a weekly is presenting problems that may affect the release, the DMCCB will decide if delay the release start or kick–off the process and fix the problems in forthcoming release candidates.

Following steps have to be used to create the first release candidate:

- create a release candidate in GitHub (Codekit[8])
- execute a build (using Jenkins job run-rebuild)
- create and publish the packages. For the Science Pipelines, EUPS packages are created and published in the package repository:

[7] This is not a full test, but just ensures that there are no regressions nor problems at a first glance.
[8] Code management toolkit `https://github.com/lsst-sqre/sqre-codekit`

  - source packages (using run-publish Jenkins job)
  - binaries packages (using tarball Jenkins job)

- create the distribution package. Still, for the Science Pipelines, a docker image is generated and made available in the docker repository:
  - distribution generated and published using build-stack Jenkins job

This procedure should be applied for all software products in the DM product tree.

The parameters required are:

- GIT_REFs: the existing Git references, for example, "w.2018.52". This is the stable weekly build identified.
- GIT_TAG: the release candidate to be generated, for example, "17.0.0.rc1"

For the Science Pipelines, the above steps are collected in one single Jenkins job:

```
releases/official-release
```

Note that for all software product using EUPS, like for example the Science Pipelines, release candidates need to start with a letter and not with a number. Currently, the letter $v$ is used.

## 4.5 Other Related Artifacts

Together with the creation of the release candidate, other artifacts need to be branched:

- Documentation: to consolidate all information relevant for the new release. This includes the release note. As an example, for Science Pipelines, the package *pipelines_lsst_io* need to be branched.
- Environment definition packages, like for example packages containing conda environment files (YAML).
- Distribution packages; still referring to the Science Pipelines, the package **lsst** where *newinstal.sh* is developed, need to be branched also.

- Test data packages.
- Other packages not tagged automatically, but required to be in line with the released software.

## 4.6 Release Candidates Validation

The release candidate needs to be validated.

In an ideal case, a test campaign following a specific test plan should be conducted. This test campaign should demonstrate that the release candidate, and therefore the forthcoming release, is behaving as required and expected. T/CAM and product owner shall be responsible for this activity.

Practically, the validation in many cases can be:

- installation/configuration of the binaries packages, or distribution image; usually done manually
- inspect of the installation, that all expected files and configuration are there
- execution of some demo package available case by case depending on the SW product
- try some use cases in order to prove that the release candidate is behaving properly
- ask downstream users to act as beta testers.

For Science Pipelines, a characterization report is produced by the product owner, in order to document the release outputs. This is a new document created for each major release.

## 4.7 Resolving Problems

In case problems are found during the validation, a DM issue needs to be created in Jira. This issue shall follow the usual development process as described in the developer guide. The fix will be first implemented in a ticket branch, reviewed and merged to master. Once the fix has been proved to work, it can be backported to the release branch.

The release branch will be created only for the packages to be fixed and will follow the naming convention described in 3.3.1.

The developer guide will provide examples of backporting procedures (DM-19950).

In a special case, that an issue cannot be fixed on master, the ticket branch can be opened based on the release branch.

The porting can be applied also in case that an issue, implemented on master after the first release candidate has been created, has to be included in the release.

The DMCCB has to overview the backporting process and take corrective actions if needed. T/CAMs shall be involved in the process. Backporting may require considerable use of development resources and/or delay in the final release availability.

## 4.8 Additional Release Candidates

Once one or more issues have been fixed on the release branch, a new release candidate has to be generated. The same steps used for generating the first release candidate, need to be used also in this case, with different parameters:

- GIT_REFs: the starting Git references, for example ”b17.0.x 17.0.0.rc1”. This implies, where available the release branch will be used instead of the previous release candidate
- GIT_TAG: the release candidate to be generated, for example ”17.0.0.rc2”

Further release candidates can be created just repeating the process with the appropriate parameters. The release candidate 3 will require the following parameters:

- GIT_REFs: the starting Git references, for example ”b17.0.x 17.0.0.rc2”.
- GIT_TAG: the release candidate to be generated, for example ”17.0.0.rc3”

Generalizing, a release candidate N will be based on the release branch and the previous release candidate N-1.

## 4.9 Final Release

Once a final release candidate **Nf** has been identified, the final release can be created.

To create the final release, just repeat the same steps used to create the release candidates, using the following parameters:

- GIT_REFs: ”17.0.0.rcNf”
- GIT_TAG: ”17.0.0”

Note that the final release is from a point of view of content identical to the last release candidate.

Final tags need to be done manually for all the additional documentation, environments and other packages.

# 5 Patch Releases

In the case that the DMCCB approves a proposed patch release, the process shall be:

- create the release branches from the available release tags on the Github packages impacted by the fixes (if they don't exist)
- backport the requested issues
- create a release candidate, for example *17.0.1.rc1* using the the same approach explained above (4.8)
- validate the release candidate
- fix eventual problems found in the release candidate and repeat the last 2 steps, create a new release candidate a validate it, until a valid release candidate is found
- create a final release as described above (4.9).

# 6 Incorporating third party code

The dev guide has a procedure [9] for making a third part package for inclusion in the pipelines distrib. In the future they may be other distributions as well. This procedure should be slightly modified such that the RFC is always flagged for CCB approval.

A distribution is defined in Section 2.6 an example distribution would be the docker container available as the kernel in the notebook aspect of the science platform. A collaboration may wish to get some non LSST code included in that distribution.

There are several reasons to include third-party code in our distributions:

1. There are packages which we use in our code e.g. starlink_ast. These are standard dependencies. Currently these are put in lsst github org and probably should not be. Apart from that this is a well understood reason for third party code.
2. There are packages which we want to use when working with the deployed software. Pandas might be such an example - we don't need it to build or run our code but its sure handy for interaction with the data.
3. A collaborator has a package which depends on our code and is intended for some form of post processing. Another example here would be an algorithm we want to use such as ngmix which we do not necessarily need to support. This is what most people on LSST are thinking about under the topic of third party software.

In all three cases the DMCCB should decide if a package is to be included in a given distribution. We currently only distribute pipelines and this is the code base most likely to be affected here. One could see a need to distribute AP separately from DRP in production - the principle would remain the same a package may be included in one or more distributions. The DMCCB needs to control the list of such packages.

We currently handle dependencies such as in type 1 above using using EUPS and soon via conda. We could use the same mechanism for the other types of packages also, we should require the third party contributed packages are conda installable.

[9] https://developer.lsst.io/stack/packaging-third-party-eups-dependencies.html

## 6.1 Support

We should take care to clarify that inclusion of a package in an LSST distribution does not imply support for that package. By support we mean that LSST personnel will not necessarily fix this package if has a problem. Packages included especially of the type 3 above, could be included as is not even necessarily built by our CI system nor do they need to conform to any LSST rules apart from licensing. These should not live in the LSST github org. We currently conflate inclusion in the distrib with inclusion in the org. We should create another github org such as lsst-community for these packages. Though if we use something like conda to install the requisite packages this is not strictly necessary. [10].

## 6.2 Testing

In the case of type 2 above we may want to have a certain level of testing to make sure it works when deployed.

In the case of type 3 above we should work with collaborators to include tests which exercises our code and interfaces in the way the code expects - this should be enough to alert us and our collaborators that the code may not work with some new release. The earlier we can catch this the better.

## 6.3 Migration

If some collaborator code should prove very useful and is demonstrated on say 1% of data to provide results the community would like to have for all data. Then if the the Project Science Team and DMCCB agree it should migrate to LSST proper and become supported. This would require work from the contributor and LSST staff to get the package to conform to an acceptable amount with LSST rules [11].

[10] Even if we do support packages like starlink_ast we should not put them under the LSST org I feel

[11] We should have a set of acceptable rules e.g. it is essential to have unit tests, it is desirable to have a clean commit history.

# A Status and Problems

## A.1 Status of the implementation

At the time this technote is in writing, March 2019, only the **lsst_distrib** repository is released and distributed.

### A.1.1 Packaging

The following technologies are already used in DM for packaging:

- **Eups**: for the majority of DM packages. It is used also for all third party libraries that require customization, or that are not available in the conda channels. The lsst DM binary Eups repository is hosted at `https://eups.lsst.codes/`.
- **Sonatype Nexus**: for java based software packages.
- **Conda**: for packages publicly available, that can be used without any customization
- **Pip/PiPy**: for python packages publicly available.

### A.1.2 Distribution

The main tool used for distribution to operations is **Docker**.

The Science Pipelines distribution to the science comunity is mainly provided using the script *newinstall.sh*. This scripts permits to retrieve a specific Science Pipelines build or release available in the Eups repository, and setup the environment for its build and execution.

In `pipelines.lsst.io` instructions are provided on how to deploy a Science Pipelines using *newinstall.sh* or **Docker**.

### A.1.3 SW Products Identification

No software products as described in the product tree in LDM-294 have official releases so far.

Only the Science Pipelines distribution, identified by the *lsst_distrib* Git mata-package, has regular official releases. All explicit and implicit dependencies in *lsst_distrib* which team is **Data**

**Management** or **DM Externals** are considered part of the distribution.

The following Git teams are relevant for the identification of the Science Pipelines distribution:

- the **Data Management** team identifies all DM developed software included in the distribution.
- the **DM Externals** team identifies third-party libraries required by *lsst_distrib*. These are software packages developed outside DM, that are not available in public conda channels, or that are updated often and therefore can't be included in the conda environment definition (A.1.4). See draft DMTN-110 for current problems and possible solutions on conda environemnts.

A third team, **DM Auxiliary**, identifies auxiliary packages to be tagged when a release or build of the distribution is done, but they are not part of it, therefore not distributed with it.

### A.1.4 Environment

The conda environment used for building the Science Pipelines is defined in the Git repository *scipipe_conda_env*.

### A.1.5 Other Tools

The build tools used to build Science Pipelines software are available in *lsstsw* and *lsst_build* Git repositories.

The Jenkins scripts also are an important part of the tooling, facilitating a large number of actions, like, test a ticket branch, automate releases, or provide periodic build (weekly/daily).

The tools picture is completed by *codekit*, which permits interaction with multiple Git repositories. Just as an example, it is used to create the same tag on all the Git repositories composing a defined product automatically, instead of creating it manually.

## A.2 Open Problems

In this section, a list of problems derived from the current development approach is enumerated.

While these problems are unresolved the only release process suitable for DM is to release the entire codebase each time.

Their resolution may lead to a different development approach, and therefore the proposed release procedure may need to be reviewed accordingly.

### A.2.1 SW Product Composition

A Git repository may be included in more than one software product.

This makes impossible to apply in a consistent way the release procedure to an SW product that is not the Science Pipelines. The reason is explained in the following example.

Product A is composed of 100 Git repositories.

Product B is composed of 80 Git repositories.

70 Git repositories are shared by both products.

When release 1.0 is done on product A, all 100 repositories will be tagged, and corresponding Eups packages are created. This will include the 70 shared repositories.

One week later, product B is ready for the release 1.0, and some of the 70 shared repositories have been updated. For them, the 1.0 tag required for product B release, will not be the same as the 1.0 tag required for the product A release. This implies that for some repositories, release 1.0 of product A, is different from release 1.0 of product B.

**Requirement**: there should be a 1 to 1 correspondence between software products and Git repository. If this is not possible, each Git repository shall be included in only one software product (one SW product to many Git repositories). Note also that, each Git repository used for building, unit testing and packaging the software products shall not be included in the SW product itself, but versioned separately and be part of the environment definition. See section 2.2.

### A.2.2 Code Fragmentation

The high number of repositories causes problems in that it :

- increases the build time: each Git repository needs to build each time. This imply that extra activities need to be done, such as clone, checkout, package creation and push to the `eups.lsst.code` repository. All these activities are not time consuming on itself, and may not represent a problem is just one repository is added. However, if the number of repositories included in a software product grows without control, the overall build time will become a problem.
- increases the release time: all Git repositories need to be built and released each time. As for the build time, adding one repository to the software product, will not represent a problem in terms of time. However, if the number of repositories grows without control, the overall time required to do a release will become a problem, especially for those software products that require patches to be available very quick.
- increases the failure probability: the tooling may be affected by network glitch or similar technical issues. This may lead to the failure of the build/release process in a non-deterministic way.

Moving 3rd party libraries to conda environment will mitigate these problem. However, this requires proper management of the conda environment. See draft DMTN-110.

**Requirement**: the number of Git repositories shall be kept low. DM-CCB shall approve each time a new repository is introduced since this has an impact on the maintainability of the system. All third party libraries, shall also not be part of the stack, if not requiring source code changes.

### A.2.3 Binaries Persistence

EUPS builds a Git repository and installs the binary packages locally. If a Git repository is not affected by any changes, once installed locally by EUPS, it will not be rebuilt.

Continuous integration tools instead are making available, in the remote repository at `https://eups.lsst.codes/`, the binary packages for macOS and Linux platforms. However, the build tools in lsstsw are not able to resolve the binary packages from that remote repository. Therefore, each time the Science Pipelines is built from scratch, all Git repositories are rebuilt.

A Git repository should be rebuilt, only in case its source code has changed, or one of its dependencies has changed (in case of semantic versioning, only for breaking changes that increase the major version).

**Requirement**: it shall be possible to resolve a package binaries from `https://eups.lsst.codes/` if available, instead of building it from the source code.

This problem is not blocking on the release process, but its resolution permits a better optimization of the builds.

# B Proposed Improvements

Given the definitions in section 2 and the current status and problems described in the appendix A, the following way forward is proposed.

- identify a software product that requires delivery soon. A possible candidate could be the *Calibration* pipeline which is now part of the Science Pipelines, to be used soon for AuxTel commissioning. The software product could be defined using a *meta-package* that includes only the Git packages relevant to it.
- identify all the Git packages that are shared with other software products and included them in a *library meta-package*. This library meta-package is a software product itself and needs to be managed and released separately.
- update the build system in order to be able to resolve the library from the binary repository (EUPS)
- update the tooling to deal with tagging (code-kit) and continuous integration (Jenkins scripts).

All the Git repositories included in a software product meta-package should be built and released all at the same time, as it is done now for the full Science Pipelines.

Initially, the above changes shall not affect the possibility to release the Science Pipelines as it is currently done.

# C References

## References

**[DMTN-110]**, Comoretto, G., 2019, *Conda Environment Proposal for Science Pipelines*, DMTN-110, URL https://ls.st/DMTN-110

**[SQR-016]**, Economou, F., 2018, *Stack release playbook*, SQR-016, URL https://sqr-016.lsst.io

**[LDM-672]**, Guy, L., Comoretto, G., O'Mullane, W., et al., 2019, *LSST Software Release Management Policy*, LDM-672, URL https://ls.st/LDM-672

**[LDM-148]**, Lim, K.T., Bosch, J., Dubois-Felsmann, G., et al., 2018, *Data Management System Design*, LDM-148, URL https://ls.st/LDM-148

**[LDM-294]**, O'Mullane, W., Swinbank, J., Jurić, M., DMLT, 2018, *Data Management Organization and Management*, LDM-294, URL https://ls.st/LDM-294

# D Acronyms used in this document

| Acronym | Description |
|---|---|
| AP | Alert Production |
| B | Byte (8 bit) |
| CAM | CAMera |
| CCB | Change Control Board |
| CI | Continuous Integration |
| DM | Data Management |
| DMCCB | DM Change Control Board |
| DMTN | DM Technical Note |
| DRP | Data Release Production |
| EUPS | Extended Unix Product System |
| LDM | LSST Data Management (document handle) |
| LSST | Large Synoptic Survey Telescope |
| RFC | Request For Comment |
| SQR | SQuARE document handle |

| | |
|---|---|
| SW | Software (also denoted S/W) |
| T/CAM | Technical/Control (or Cost) Account Manager |
| YAML | Yet Another Markup Language |