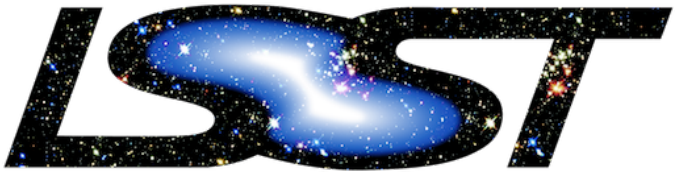

LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)
Data Management

DM Release Process

Gabriele Comoretto

DMTN-106

Latest Revision: 2019-03-25

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

Contents

DM Release Process

1 Introduction

The release procedure documented in this document should be general and valid for all software products in the LSST Data Management subsystem. It is an abstraction of SQR-016, but it is supposed that the Jenkins builds used for the science pipelines can be applied in the same way to other Software Products.

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

Section A summarizes the actual development process, and reports the derived problems that affect the release process.

1.1 Applicable Documents

LDM-148 DM Architecture
LDM-294 DM Project Management Plan

2 Definitions

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

2.0.1 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 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 operational product. It shall not contain any build or debug tool. It shall be optimize for the operational activities. [1]

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

In some cases, for simplicity, the same environment is used in development, tests and operations, but this does not justify the inclusion of the environment definition in the Software Product itself.

2.1 Software Product

A Software Product is a component of the subsystem (DM) product tree.

The following activities are done on a SW Product:

- development activities

[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.

- test activities, usually unit tests and system test
- release activities
- packaging activities
- operational activities, in stand alone 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 shall correspond to a single repository (git package). All its dependences are also a Software Product and therefore released separately. However, this is not the case for DM.

In DM, where a Software Product is comprised of multiple git packages, a GitHub *metapackage* is used to identify the Software Product. All git packages composing the Software Product are dependencies in a GitHub *metapackage* and released at the same time. In this case, a git package shall be related only to one Software Product, and it is not by himself a Software Product.

The following elements should not be part of a Software Product:

- build tools, otherwise we will have build tools deployed in operations.
- environment definition, since the environment depends on the final instantiation of the Software Product that are not visible in the development phase.

These elements shall follow a parallel development / release process.

Test data should also not be included in a Software Product. However including small datasets for enabling the unit tests is not a bad practice.

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

2.2 Dependencies

A package may only depend on other packages. Third party dependencies are either packages or part of the conda environment.

The dependency information is provided in each git package in specific dependency files according to the build system e.g. requirements.txt, EUPS table file, etc. These files , and hence the dependencies, are considered part of the source code and controlled and versioned in the same manner.

2.3 Software Release

A software release is a consciously identified version of a software product documented with a software release note. The identifier is usually of the form M.n (M.n.p in case of semantic versioning, see Section 2.6): and is used also as a tag in the SW repository.

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

2.4 Binary Packages

A binary package is a package containing executable binaries for the corresponding release. It is created by building the SW provided in the release tag. Binary packages can be created to support multiple platforms (such as Linux, OSX, windows) if required.

In some cases, these packages do not contain compiled binaries, but executable scripts or just text files.

Binaries should be generated only once, and made available for their use by dependent software or for deployment.

The majority of DM software is using EUPS for binary packaging. Other platforms can be considered, for example *conda* or *pypi*. Software products implemented in java are using jar/war binary packaging. [2]

2.5 Distribution

A distribution is a collection of binary packages to be deployed together.

A distribution can be used for different purposes:

[2] So far there are a few technologies used to handle binary packages. It is recommended to assess and optimize them, converging to use only one. Conda seems to have become a standard for python based projects.

- to make available software releases for operations or commissioning
- to test (integration, validation, operation rehearsals) software releases
- to provide software releases to external collaborators. In this case source code can be distributed instead of binaries.

Distributions may also include non binary packages or source code, as needed.

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

2.6 Versioning and Naming

DM will switch to Semantic Versioning [3], 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:
$16.1.0 \neq 16.10.0$

DM software is currently versioned using two digits separated by a "." . For example Science Pipelines latest release:
17.0 The first number is the major version and the the second number is the minor version.

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

2.6.1 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, where **0** means no patch.

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 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. 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 by 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 documents the content of a release.

The following information should be provided:

- Installation instructions. This is usually a manually written set of instructions.
- Narrative section describing the content of the release, this summary should be writ-

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

ten by a person not auto generated. The T/CAM is responsible for making sure this is provided.

- List of jira issues included in a release. This information can be extracted from Github. Completed epics will be highlighted while other issues will listed to be comprehensive.
- Technical information like the GitHub tag, dependencies, etc. can be extracted automatically from Github or other tools.

4 Software Release Procedure

This release procedure has been derived from 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 Github software package (SW product)
- if unit tests pass, generate binary packages
- build downstream dependencies: CI build on SW products that depend from the newly build SW product.

At the moment, continuous integration is done 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 few days before the release activity starts.

In this way, all contributors will be able to provided 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. [6]

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[7])
- 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:

[6] This is not a full test, but just ensures that there are no regressions nor problems at a first glance.
[7] 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 generate 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 "M.n.p.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 v.

4.5 Other Related Artifacts

Together with the creation of the release candidate, other artifacts may need to be branched, and possibly have a first release candidate also:

- 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 inline 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, demonstrating that the release candidate, and therefore the forthcoming release, is behaving as required and expected.

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 of 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, 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 2.6.1.

Backporting mechanism can be summarized as follows [8]:

- Create the backporting Jira issue DM-XYZ with subject *Backport issue DM-XXX on release branch bM.n.x*
  - DM-XXX Jira issue has to be already in status reviewed or done, and the corresponding ticket branch already merged to master.
  - The ticket branch must not be deleted until the backporting is concluded.
- Create the backporting ticket branch *tickets/DM-XYZ* based on DM-XXX ticket branch
- Backport the fix from master to the release branch using the following command:
  - $gitrebase--ontobM.n.x.$
- Open a PR from the backported ticket branch to merge into the release branch
- Merge the backported branch into the release branch when the PR is approved
- Remove the backported ticket branch and the original ticket branch

Note that porting mechanism is a development activity, under the responsibility of the development team, and therefore needs to be documented in the developer guide and removed from this document. This backporting procedure is just an example. Additional backporting mechanism can be proposed.

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 the first release candidate has been created, has to be included in the release.

The DMCCB has to overview the backporting of issues to the release branch, and take corrective actions if needed. Backporting may require a considerable use of development resources and/or delay in the final release availability.

[8] This needs to be documented in the DevGuide

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 ”bM.n.x M.n.p.rc1”. Where available the release branch will be used instead of the previous release candidate
- GIT_TAG: the release candidate to be generated, for example ”M.n.p.rc2”

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

- GIT_REFs: the starting git references, for example ”bM.n.x M.n.rc2”. Where available the release branch will be used instead of the previous release candidate
- GIT_TAG: the release candidate to be generated, for example ”M.n.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 created the release candidates, using following parameters:

- GIT_REFs: ”M.n.p.rcNf”
- GIT_TAG: ”M.n.p”

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 *M.n.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

Though the dev guide has a procedure [9] for making a third part package for inclusion in the pipelines distrib we have no official policy for how we accept and deal with such packages.

There are several reasons to include third part 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. AstroPy 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 the 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.

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. 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

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

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 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 **lsst_distrib** package is released and distributed.

A.1.1 Packaging System

The following technologies are already used in DM for packaging:

- **Eups**: for the majority of DM packages. This included 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 DM software developed packages
- **Conda**: for packages public available, that can be used without any customization
- **Pip/PiPy**: for python packages public available. In some cases for packages pip/PiPy is used for DM developed python packages, for example *docsteady*.

A.1.2 Distribution System

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

In `pipelines.lsst.io` instructions are provided on how to deploy a science pipelines docker image.

A second distribution approach is also 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.

A.1.3 SW Products Identification

The software products are developed using multiple GitHub packages. A GitHub metapackage, for example *lsst_distrib* for science pipelines, is identifying the software product. All git packages of a software product are dependencies in that metapackage, direct or indirect.

GitHub teams are used to differentiate between two types of git packages:

- **Data Management** team: it includes all dependencies of *lsst_distrib*
- **DM Externals** team: for packages used to build and distribute *lsst_distrib* but that they are not part of it.

There is a third team, **DM Auxiliary**, that identifies auxiliary packages to be tagged. Git packages in this team are not part of the software product, nor distributed with it.

Note that in reality, *lsst_distrib* does not correspond to a software product, but to a distribution product. It includes not only the different pipelines, but also test data and build tools.

A.1.4 Environment

The conda environment to be used for building the science pipelines is defined in the Git package *scipipe_conda_env*. No releases nor tags are made on the conda environment git package.

TO BE CLARIFIED: how are java build environments managed?

A.1.5 Build Tools

The build tools used to build science pipelines software are available in *lsstsw* and *lsst_build* git packages.

The Jenkins scripts also are an important part of the tooling, facilitating a large quantity 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 GitHub packages. Just as an example, it is used to create the same tag on all the GitHub packages composing a defined product automatically, instead of creating it manually.

TO BE CLARIFIED: which are java build tools?

A.2 Open Problems

In this section a list of problems derived from the current development approach are 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 GitHub package may be included in more than one software product.

This make impossible to apply in a consistent way the release procedure to a SW products that is not the science pipelines. The reason is explained in the following example.

Product A is composed by 100 Git packages.

Product B is composed by 80 Git packages.

70 Git packages are shared by both products.

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

One week lather, product B is ready for the release 1.0, and some of the 70 shared packages 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 packages, release 1.0 of product A, is different from release 1.0 of product B.

Requirement: there shall be a 1 to 1 correspondence between software products and GitHub packaged. If this is not possible, each GitHub package shall be included in only one software product. Note also that, each GitHub package 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 2.1.

A.2.2 Code Fragmentation

The high number of packages causes problems in that it :

- increases the build time: each Git package needs to be build each time.
- increases the release time: all Git packages need to be released each time.
- 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 reduce this problem. However this requires proper management of the conda environment.

Requirement: the number of Git packages shall be kept low. DM-CCB shall approve each time a new package is introduced, since this has an impact on the maintainability of the system. All non DM 3rd party libraries, shall also not be part of the stack, if not requiring source code changes.

A.2.3 Binaries Persistence

Each time we do a build, all git packages and rebuild. However, if the package is already installed in the local environment and is not affected by any changes, it will not be rebuild.

In reality, once a package has been built, and the corresponding binaries have been made available, for a specific platform (linux, osx, etc), in the package repository (`https://eups.lsst.codes/`), it shall be possible for all the downstream users, to use it without building it again.

A GitHub package should be rebuild, 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 the package binaries from `https://eups.lsst.codes/` if available, instead of building them from the source code.

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

B References

References

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

[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

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

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

C 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
OSX	Macintosh Operating System
PR	Pull Request
RFC	Request For Comment
SQR	SQuARE document handle
SW	Software (also denoted S/W)
T/CAM	Technical/Control (or Cost) Account Manager