LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)

LSST DM Software Release Considerations

John Swinbank

DMTN-044

Latest Revision: 2017-04-24

Abstract

This attempts to summarise the debate around, and suggest a path forward, for LSST software releases. Although some recommendations are made in §6, they are intended to serve as the basis of discussion, rather than as a complete solution.

This material is based on discussions with several team members over a considerable period. Errors are to be expected; apologies are extended; corrections are welcome.

Contents

LSST DM Software Release Considerations

1 Consumers of Releases

Our usual discussion over releases conflates (at least) three different classes of consumers, which I summarize below.

1. Regular LSST developers (discussed in §2);
2. Close collaborators, who are not intimately familiar with the codebase, but require access to new features and bugfixes with minimal latency (§3);
3. External users, where the term "external" is used loosely to include science collaborations or the Commissioning Team, who require codebase which they can use for carrying out medium- to long-term projects without handling regular API changes (§4).

Particularly as regards the first class of users — LSST Developers — the question of releases is intimately linked with concerns over development workflow.

2 Developer Workflow

2.1 Requirements & Issues

Our regular developer workflow is well described in the Developer Guide[1]. Fundamentally, developers perform their work on ticket branches (`tickets/DM-NNNN`) which are reviewed and rebased onto `master` before merging.

Note that in this model, the tip of the ticket branch immediately before merging should be identical to the state of `master` immediately after the merge. This means that the developer can demonstrate (e.g. using the CI system) that merging their changes will not cause `master` to break or regress[2].

[1] `https://developer.lsst.io/processes/workflow.html`

[2] Merging without rebasing does not have this property: the post-merge state of `master` will neither be the same as its pre-merge state *nor* the state of the ticket branch.

It is worth emphasizing that this development model means that code which hits `master` is generally well behaved: we do not frequently have problems with it failing to build or with bugs which render the codebase fundamentally useless[3].

Key issues include:

- Turnaround time on reviews, in particular of complex work, can be long. This can cause substantial time to pass before the work on the ticket branch being completed and the merge to `master`. This means:
  - Other work dependent on the new feature backs up onto a series of ticket branches, all based upon each other. Keeping track of which branches require which other branches becomes awkward.
  - Since releases are made from `master`, the new functionality is not available in any released version of the codebase.
- `master` continually changes as other developers are merging their work. This means:
  - Rebasing can take substantial effort.
  - Rebasing a long-running branch against a substantially changed `master` at the end of a long development project can be a major task. To minimise the effort, developers generally frequently rebase against the latest `master`. After rebasing, it's frequently necessary to rebuild all or much of the stack which (as below) can be a length process.
- Nominally "high-level" work, which superficially seems to affect only Python code in packages on the leaves of the tree, often also requires changes to lower-level packages (usually, but not always, afw). This means:
  - Rebuilding after rebasing is often a length process: it takes of order one hour to rebuild from a low-level package.
- Our build process is fragile: although (as above) it is rare (but not unknown) for the codebase to be unbuildable, it makes numerous assumptions about the system on which it is building, and relies on complex (and often poorly-understood by most developers) bespoke tooling (lsstsw, EUPS, ...). This means:

[3] Of course, sometimes regressions do slip in.

- Even after successfully rebasing, developers often waste time debugging local build problems before they can resume work.

2.2 Possible Approaches

- Enforce API/ABI stability by preventing modifications to low-level packages. This means:
  - Packages obviously cannot be frozen indefinitely, but rather we would accept changes only during certain windows.
  - A release manager (§5.1) may be required to coordinate this process.
  - Work which requires changes to low-level packages could only be merged during dedicated windows, and would otherwise back-up.
  - Any other work which happened to coincide with a window would encounter substantial disruption as, likely, a large number of interfaces would change at once.
- Require developers to develop against stable (e.g. weekly) releases, rather than latest `master`. This means:
  - Developers have a stable ABI and API to target.
  - The "shared stack" available on LSST-provided developer hardware (`lsst-dev01`, the Verification Cluster) should always provide the latest weekly, so developers can develop against that with no stack maintenance required on their part.
  - Work must still be merged to `master` (or some other common development branch) eventually. This means one of two things:
    1. Work is merged without rebasing. The merge is complex. The post-merge state of the stack is not testable until after the merge has taken place.
    2. Work is rebased from the weekly to `master` before merging. The rebase is potentially complex (since it involves doing in one step a rebase which might otherwise have been undertaken incrementally). The advantages vis-à-vis the current workflow are unclear.
- Provide regularly updated *binary* releases of current `master` (§5.4). This means:
  - Following a rebase, developers should not need to recompile any packages other than those on which they are directly working.

  - Shared stacks on common developer hardware can be automatically updated with the latest release.
  - Generating and installing binary releases takes time, so there is still a potential speed-bump to developer workflow.
  - It is likely impossible to provide binaries targeting all possible platforms which developers may reasonably wish to use.

- Restructure stack code to better orthogonalize between packages (§5.2). This means:
  - Changes to high-level packages are less likely to require changes to low-level packages.
  - Huge low-level packages (afw) could be split into smaller components, so that a change would be less disruptive.
  - This might help, but is not in itself a complete solution: some high level changes will *always* require changes in low level packages.

- Provide better development tools, replacing or augmenting tools like lsstsw to make our build process more reliable, less prone to user error, and (where possible) faster (§5.3). This means:
  - Developers who need to recompile after rebasing should find the task easier, faster and less disruptive.
  - It's unclear how much benefit could really be gained here: lsstsw is clunky and could certainly be streamlined, but it's not a disaster.

3 Close Collaborators

3.1 Requirements & Issues

It is important to be able to rapidly provide collaborators, e.g. members of the Camera Team, with versions of the stack which may resolve particular issues or provide new features with minimal latency. However, these individuals are unfamiliar with the stack and our particular toolchain than regular developers. They are frustrated by awkward software and failed builds.

Collaborators may be already using a particular weekly (or other) release, and not be in a position to deal with API or other instability. Therefore, simply upgrading to a newer weekly

may not always be convenient. Further, for the reasons discussed in §2, it may not be practical for the work to "simply" be merged to `master` and hence appear in an upcoming weekly.

However, it is accepted that access to the latest features will, at some level, require tracking new development. It is not practical to commit to provide arbitrary releases with an arbitrary set of bug fixes and new features upon request.

3.2 Possible Approaches

- Provide better development tools (§5.3). It seems unlikely that this would be able to provide a sufficiently seamless process, though.
- Provide easy-to-install binary distributions corresponding to particular branches (§5.4). This means:
  - End users could use our standard binary workflow to install a binary release which contains the specific features pulled onto a branch by their LSST contact.
  - The project would make no ongoing effort to support these binary releases.
  - When LSST developers may develop work on feature branches and provide it to collaborators using this method, the motivation to go through the effort of merging the work to `master` may be reduced. There is, therefore, a danger of producing a series of "stale" ticket branches which contain important functionality but which have never been merged. Avoiding this would require real discipline.

4 External Users

4.1 Requirements & Issues

External users, for a generous definition of "external", require a release which provides some level of functionality and is "supported" in the long term — that is, errors discovered are fixed, and there is no expectation that they need to upgrade to a newer version to use functionality advertised as being part of the release.

It is not clear to what extent external users may require that new features (as opposed to bug fixes) be added to these long-term supported releases.

The definition of "long-term" is vague, but we might reasonably assume the value of six months based on current practice. Of course, the longer support is required, the further the current development version will diverge from the supported codebase, and the harder it will be to make fixes which apply to both of them.

However, our current practice centres on date-based releases. In future, it is likely that feature-driven releases will predominate as we are required to deliver and support a particular set of functionality as required by e.g. the Commissioning Team.

It may be necessary to maintain two (or more?) release "trains" in parallel, in addition to ongoing development on `master`: for example, to support different audiences or to provide a semi-stable branch[4]. No concrete requirements have been expressed here.

4.2 Possible Approaches

- Release procedure:
  - Releases can be made by directly selecting (and subjecting to a battery of tests) a particular version of the `master` branch, without requiring any special action from most developers. This is how releases have been handled to date. As per §2, we have been relatively successful in maintaining developer discipline and strict practices, so that `master` is generally in a "releasable" state. This approach enables developers to continue with their work normally while the release happens in parallel.
  - Releases could be based around a freeze of development on `master` to ensure that a well-tested and working version of the codebase is released. The details of implementing such a freeze, how developers might continue working while `master` is frozen, the branch policy, etc, are regarded as out of scope for this document[5].
- Back-porting fixes:
  - While minor technical fixes will often be straightforward to back-port to stable releases, deciding which changes to science logic constitute "bugs" that require porting will require careful thought.

[4] e.g. FreeBSD's "STABLE" vs "RELEASE".

[5] Although it is not current practice, historically a `next` branch has been used to enable ongoing development while `master` is frozen.

- This process might reasonably be coordinated by a dedicated Release manager (§5.1), working in conjunction with the DMCCB.
- Porting substantial changes may become extremely complex as not just the logic but the underlying infrastructure may be quite different. The level of effort required is likely large.
- Fixes, even those which are known to be required by an older release, should *always* be developed targeting `master` and then back-ported (unless the code being fixed has already been removed from `master`). No new work may be performed on release branches.

5 Technical and Resource Implications

5.1 Release Manager

Some of the approaches outlined above require the services of a Release Manager. Such an individual might be required to:

- Carry out work relating to the mechanics of making releases (e.g. applying appropriate tags, ensuring that releases contain the requisite features, etc);
- Collate and compile release notes and other supporting material for stable releases;
- Work with the community and the DMCCB to understand which issues need to be back-ported to stable releases;
- Make fixes as required, including back-porting where necessary.

Some of this work is currently being carried out by SQuaRE. Other parts of it, in particular back-porting of bug fixes, may require development expertise that is currently only found within other teams (e.g. Pipelines likely best understand how to backport Pipelines code, ditto for DAX, etc). Two possible approaches have been suggested here:

- Centralising all of this expertise in one named individual would streamline the process and enable easier resource loading for the development teams.

- Having the release manager role be simply a managerial one (which may rotate between project members) who can call on support from the development teams may reduce the concentration of skill required in one individual, at the expense of a more complex management and resourcing structure.

5.2 Code Restructuring

A significant overhaul of the structure of the codebase has been suggested on several previous occasions. See, for example, DM-2003 or various discussions on Confluence[6][7]. These have generally foundered due to the large amount of work required, the desire to address more immediately pressing concerns, and uncertainty over long term priorities.

Broadly speaking, those concerns still apply. There is no effort currently available within the Science Pipelines groups to spearhead a restructuring effort, although some developer time could be devoted to supporting work organized by another team (e.g. Architecture, SQuaRE).

5.3 Developer Tool Upgrades

Action here might range from a relatively modest overhaul of the lsstsw script to make it more user friendly to a root-and-branch reconsideration of the way our software is packaged, the build tools and utilities, and the distribution system.

The effort required would obviously vary substantially depending on how ambitious this project was. It seems unlikely that a significant overhaul would be worth the effort insofar as it narrowly affects pipeline developers, but there might also be knock-on benefits for other consumers of the software (who would find it easier to install and better integrated with their typical environment, e.g. pip, conda, etc.).

There is no effort currently available within the Science Pipelines groups to spearhead a restructuring effort, although some developer time could be devoted to supporting work organized by another team (e.g. Architecture, SQuaRE).

[6] https://confluence.lsstcorp.org/display/DM/Winter2015+Package+Reorganization+Planning
[7] https://confluence.lsstcorp.org/display/DM/Summer2015+Package+Reorganization+Planning

5.4 Binary Release Procedures

The SQuaRE team is developing a system which makes it possible to produce binary releases from Jenkins runs. Specifically:

- They directly anticipate being able to generate a binary release from ticket branches upon request.
- They are open to the possibility of generating binaries from all builds of `master` (or, presumably, based on some more sophisticated selection algorithm).

Note that a binary release currently consumes around 5 GB of storage. Long-term archiving of binaries corresponding to *every* build of `master` is likely impractical given the storage considerations.

It would take some effort for SQuaRE to develop a system capable of generating binaries for all `master` builds, including time devoted to developing sensible resource management systems, etc. However, this is likely possible upon request.

Note that the established variety of platforms upon which team members and their collaborators are working, together with issues such as C++ ABI compatibility [8] mean that it is likely impossible to guarantee to provide binaries compatible with every system.

6 Recommendations

Given the above considerations, the following course of action sounds plausible:

6.1 Developers

- Provide regularly updated binary releases of current `master`, automatically installing them onto shared developer architecture.
- Encourage developers to develop against weekly releases wherever possible, making use of binaries to help with rebasing.

[8] e.g. `https://wiki.gentoo.org/wiki/Upgrading_GCC#The_special_case_C.2B.2B11_.28and_C.2B.2B14.29`

- Schedule minor package reorganization and development tool improvement work when possible and when they align with other goals, but do not anticipate a major overhaul in this area.

6.2 Close Collaborators

- Make use of the ability to generate binary releases from ticket branches to provide collaborators with access to new features and bug fixes with low latency and minimal effort on their part.
- Rely on disciplined management to ensure that those features & fixes are also merged to `master` in a timely fashion.

6.3 Long Term Support

- Appoint a Release Manager to coordinate long-term supported releases in conjunction with the DMCCB. This individual would (presumably) be part of the SQuaRE team.
- Make releases by tagging `master`, then having the Release Manager maintain the branch by back-porting essential fixes.
- Where necessary, multiple stable branches may be maintained. Developers will continue working on `master` with the RM and DMCCB back-porting work.
- Consider switching from a time-based to a feature-based release schedule.