Release Process and Policy

Gabriele Comoretto
Configuration and Release Manager

DMLT Face-to-Face
June 3-6 2019

Presentation Summary

- Summarise the conclusions of LDM-672 and DMTN-106
  - Release Management
  - Definitions
  - Release policy LDM-672
  - Release Procedure
  - Conclusions
- Outline plans for short-term and longer-term changes to the DM release process
  - Short Term Plans
  - Medium Term Plans
  - Long Term Plans
  - Condarization
  - Discussion

Release Management

- LDM-294 section 3.6 (and partially in section 7.4, DMCCB)
  - High level Release Management approach for DM
- DMCCB
  - Schedule DM releases major / minor releases
  - Approve patch releases in coordination with the T/CAMs
  - Monitor RFCs in general
- RFC for requesting unscheduled releases
- Release Issue
  - It tracks the release activity, assigned to the Release Manager
  - Placeholder for all issues related to a release
    - Blocking issues to be related as blocker to the release issue

Definitions

- The definitions are given as a reference
  - See DMTN-106 section 2 [https://dmtn-106.lsst.io/]

- Environment
- Software Product
- Binary Package
- Dependencies
- Software Release
- Distribution

Definition: Environments

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

- 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
    - template.

Definition: SW Product

- A SW product is a defined and controlled software implementation with a specific scope or function, inside an overall system.
- A SW product is meant to be used as an indivisible artifact in a service, a process or, as a library, by other software products.
- A SW product is
  - Developed
  - Tested
  - Released
  - Packaged
  - Operated
- One SW repository per SW product
  - or many SW repository per SW product
  - **NEVER** many to many (it will not be possible to release)
- A SW product shall never include:
  - Build tools
  - Environment definition (except vanilla / example)

*

Definition: Binary Package

- A package containing executable and/or libraries (binaries, script, configurations or other files).
  - SHA1 or tag → Compiled Objects → Binary Package
  - Multiple Linux, macOS, windows.

- To be generated only once and used downstream
  - Sometimes it has to be regenerated: when dependencies changes but not the software product
    - Semantic versioning
    - Dependency version in a range
  - Need to keep track of the build number in the identification

*

Definition: Dependencies

- Build and Runtime dependencies
- Project dependencies → Other SW products
  - Listed in a file, that is part of the SW product.
    - requirements.txt
- External dependencies → Third Party libraries
  - Usually part of the environment
- To be resolved as binary packages
  - Official releases
  - Release candidates
  - Builds

*

Definition: Software Release

- TAG + Software Release Note
  - proved to work before the release tag is made
  - depends only on other officially released software products.

- It shall be sufficient for a developer to:
  - Get the SW product source code
  - Resolve the dependencies,
  - Build the binaries
  - Execute the software.

- It is a monolithic snapshot of the software product to be used as is, by the downstream processes or users.

Definition: Distribution

- A collection of objects to be deployed together.
  - SW products (possible the binary packages)
  - third-party packages
  - Data
  - source repositories
  - Tools
  - Etc.
- The distribution definition shall be versioned (in GitHub) and released (tag + release note).
- A distribution can be used for different purposes:
  - to make software releases available for operations
  - to test (integration, validation, operation rehearsals) software releases or services
  - to provide software releases to external collaborators.
- Distributions can be handled in different ways:
  - Git metapackages, such as lsst_distrib
  - Docker images
  - Others

*

Questions?

LDM-672: Release Policy

- Release Stakeholders
- Release Requirements
- Summary Review

LDM-672: Release Stakeholders

- Identify (DM) Release Stakeholders
  - Science Community
    - LSST Science Community
  - Data Processing
    - Data Processing operations activities at NCSA, CC-IN2P3 and Chile.
    - For example: Calibration processing, Alert Production, DR Production
  - Other LSST Subsystems
    - Telescope and Site uses DM software to build and run their software products
  - Infrastructure Software
    - For example LSP
  - Non operational activities
    - Commissioning
    - Verification activities in general

LDM-672: Release Requirements

- Stakeholders Requirements:
  - Stable
    - Stable API/ABI and schemas
  - Timely
    - New functionalities and fixes available as soon as possible
  - Patch and Fix backport
    - Make available fixes on top of existing stable builds or releases
  - Milestones Based
    - In order to fulfilling planned project milestones, some SW products need to be released accordingly
  - Deprecation
    - Breaking API/ABI changes need to follow the deprecation procedure
- General Requirements
  - Identify SW products
    - SW products need to be clearly and unequivocally identifiable in the software repository
  - Release Note
    - Releases need to be fully described in a release note (DMTN-106, section 3.2)
  - Release fully tested
    - A software release should be fully tested and test report available.

LDM-672: Summary View

Stakeholders vs Requirements

	Identify SW Prod.	Release Note	Release Full Test	Stable	Timely	Milestone Based	Patch and Fix Backport	Deprecation
Science Community	yes			yes	yes		yes	yes
Data Processing	yes	yes	yes		yes	yes	yes	
Other SubSystems	yes	yes	yes		?	?	?	
Infrastructure	yes	yes	yes			yes	yes	
Non Operational	yes	(yes)	(yes)		yes	yes	yes	

LDM-672: Assessment

- What these requirements mean in terms of
  - Release planning
  - Backporting
  - Other aspects

- Document to complete and RFC

DMTN-106: Release Procedure

- Assumptions and Status
- Problems
- Possible Solutions
- Conclusions

DMTN-106: Release Procedure

- Assuming no discontinuity:
  - Preserve the current workflow as much as possible
  - Same tooling as now (lsstsw, etc)
  - Same CI approach as now
  - DM SW packages managed with EUPS

- Generalization
  - SQR-016

- Status
  - EUPS used for packaging (in few cases Nexus, Conda, PiPy)
  - SW product identification:
    - Github organization + github team + metapackage (lsst_distrib)
  - Distribution: metapackage + script (newinstall.sh) and Docker
  - Environment: scipipe_conda_env
  - Tooling:
    - lsstsw - lsst_build
    - newinstall.sh
    - CI scripts
    - codekit

Problems: SW Products

- No clear definition
  - So far only the Science Pipelines (lsst_distrib) is defined and released
  - DM product tree in LDM-294 is a theoretical starting point
  - Apparently Many to Many Git Package per SW product
    - This can disappear when the SW products are properly defined
- **Requirement:** It shall be clearly identified which Git repository belongs to which SW product.
- **Requirement:** each Git repository shall be included in only one SW product.

Problems: Code Fragmentation

- It increases the time needed to build a release
- It increases failure probability
- So far it is a problem for *lsst_distrib* releases: cannot be done quickly.
  - If the SW products will be composed by a limited number of Git repositories, this may not be a problem
- Each time a git repository is added to a SW product, it will increase the overall complexity of the solution and therefore the costs.
  - Adding one repository is not a problem
  - Letting the number of repos grow without control, is a problem.
- **Requirement**: keep the number of Git repositories per SW product low (tbd) and controlled by the DMCCB.

Problems: Binaries Persistence

- EUPS persists binary packages, locally (native functionality)
- CI scripts make binary packages available at: https://eups.lsst.codes/
  - But they are not used for builds (newinstall.sh can resolve them)
- Each time a build is done, there is the possibility it fails
  - Therefore we should use binary packages when available.
- **Requirement**: Lsstsw or build_tools shall be able to resolve binaries from https://eups.lsst.codes/ if available.

Possible Improvements?

- Identify and characterize the SW products
  - List of Git repositories
  - Owner and short descriptions
  - How?
    - Metapackages ?
  - SW products to be released and included in a service
  - SW products to share code between other SW products
- Update the tools to interact with GitHub (codekit) to be able to handle the defined SW products.
- Update lsstsw/lsst_build to be able to resolve binaries from

https://eups.lsst.codes/

Conclusions

- LDM-672: Collected release requirements from stakeholders
  - Further assessment/actions is needed to complete the picture in terms of release planning
- Procedure
  - The current implementation presents some problems
  - Possible solutions are suggested
  - Other ideas?
- The slides next show possible plans…

Short Term Plans ?

- Short term impacts on the **lsst_distrib** release process:
  - **None**: until action is taken on the SW products definition and tools, nothing different can be done, only lsst_distrib can be released as it is done now.
  - It should be possible to reduce the number of packages, moving to conda a consistent number of 3rd party packages, however this may take time and resources

- We may be able to release some different products:
  - For example: suit
    - It just depends on firefly (external library)
    - It has no shared code with lsst_distrib (build and run time)
    - No impacts to anybody else
    - Minimal or no impact on LSP team (just be aware of the releases)

Medium Term Plans ?

- Identify other SW products.
  - Depending on how, update the tooling...
- Exercise the release procedure on the identified SW products:
  - Improve SW products definition
  - Improve the tools
  - Iterate… on improving procedure, products and tools.
- Development activities should remain the same
  - However new dependencies between SW products should to be handled differently
- Involve a small group of people

Long Term Plans ?

- All DM SW products released following the new (improved) procedure
  - Some SW product will used as dependency

- Science Pipelines distribution
  - In the long term it could be released as a collection of SW products instead of just all those Git packages

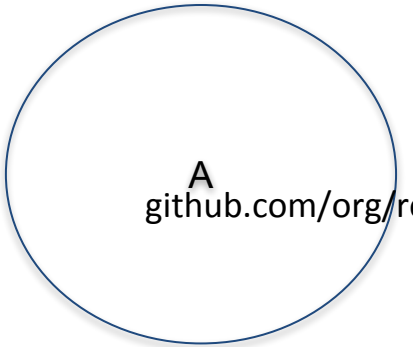
Condarization ?

- The fascinating idea to have each DM Git repository available as conda package
  - This is against the assumptions made at the beginning (continuity)
  - However we could to do it in parallel
    - Let all developments still use EUPS
    - Produce conda packages without disrupting development activities
    - Result: `conda install lsst_distrib`
    - Each Git repository released separately:
      - Drawback: hundred of releases! Reduce the number?
- To start: ensure that all 3rd party libraries are in conda

Presentation End

Example: Git Repos - SW Products

- 1 Git repository per SW product:
  - All source code of the SW product **R** is versioned in the Git repository *repoR*

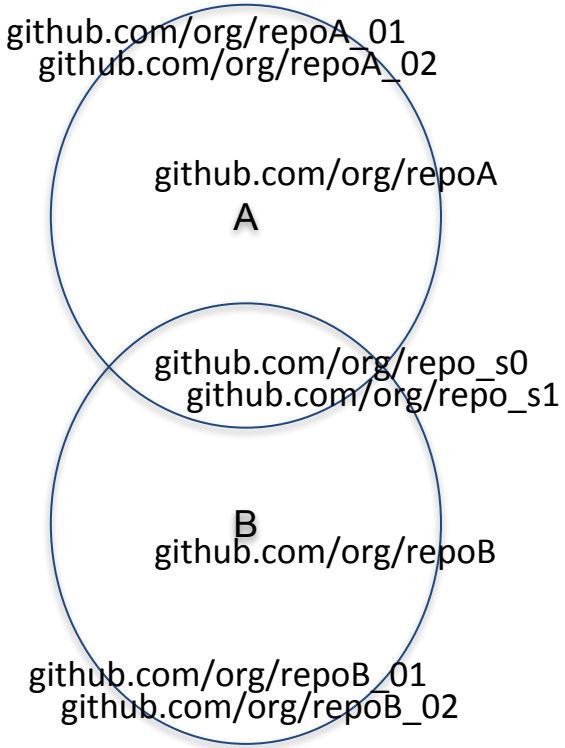


github.com/org/repo_01
github.com/org/repo_02
github.com/org/repo_03
github.com/org/repo_04
…..

- Many Git repositories per SW product
  - The source code for SW product **A** is distributed into many Git repositories
  - The SW product is represented by a Git repository that works as metapackage, *repoA*
  - The *repoA* includes a single file
    - *repositories.txt*

    listing all Git repositories included in SW product **A**
    - repo_01
    - repo_02
    - repo_03
    - repo_04
    - ….

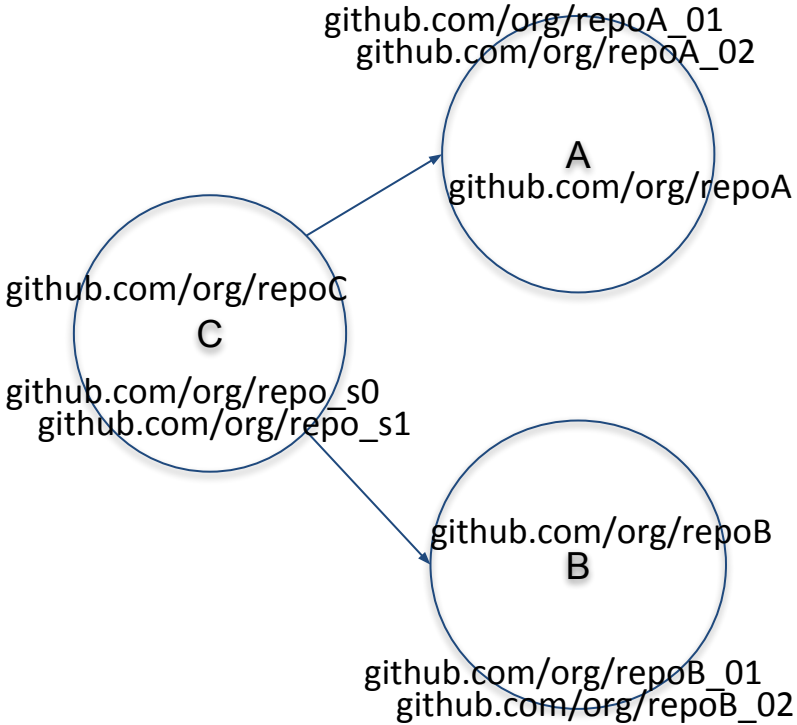
Example: many Git Repo many SW Product



- SW products **A** and **B** are composed by multiple Git repository
  - At least one Git repository is shared by both
- Product **A** release 1.0: all repos included in product **A** are tagged 1.0
  - the shared repositories are tagged also
- After some time product **B** release 1.0 is made
  - Changes has happened in the code in the meantime, shared repos are not the same
  - Those changes are required for the product **B** 1.0 release
- The new 1.0 tag on the shared repositories made for product **B**, would be different from the 1.0 tag made for product **A**
- Conclusion:
  - Is not possible to give a consistent 1.0 release tag to SW product B
    - some of its Git repositories had already that tag
    - the software has evolved in the meantime

*

Example: Dependencies

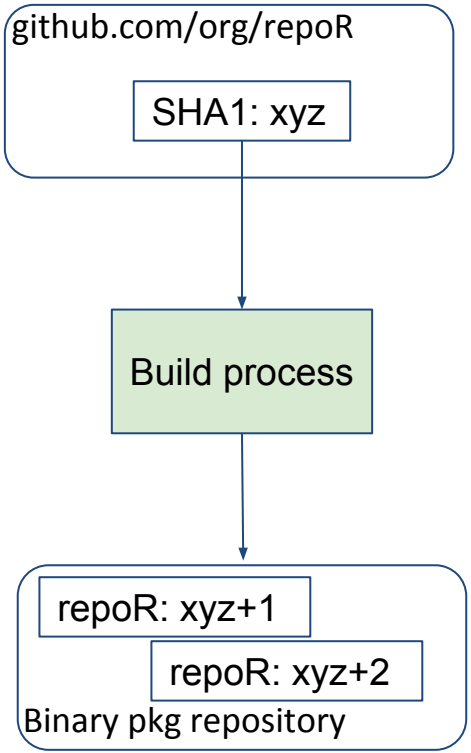


- All Git repositories shared between **A** and **B** can be collected in a third SW product **C**, on which **A** and **B** depend on
- Products **A** and **B** do not include any packages from **C**, but they will resolve them from the binary package repository.
- Products **A** and **B** have a second file, for example:
  - *dependencies.txt*

  Where it is specified from which other SW products they depend
- When builds are done (or releases):
  - First SW product **C** is build and the corresponding binary packages are uploaded in the binary package repository
  - Then build of SW products **A** and **B** are done, resolving the just created product **C** binaries packages.

*

Example: Binary Package



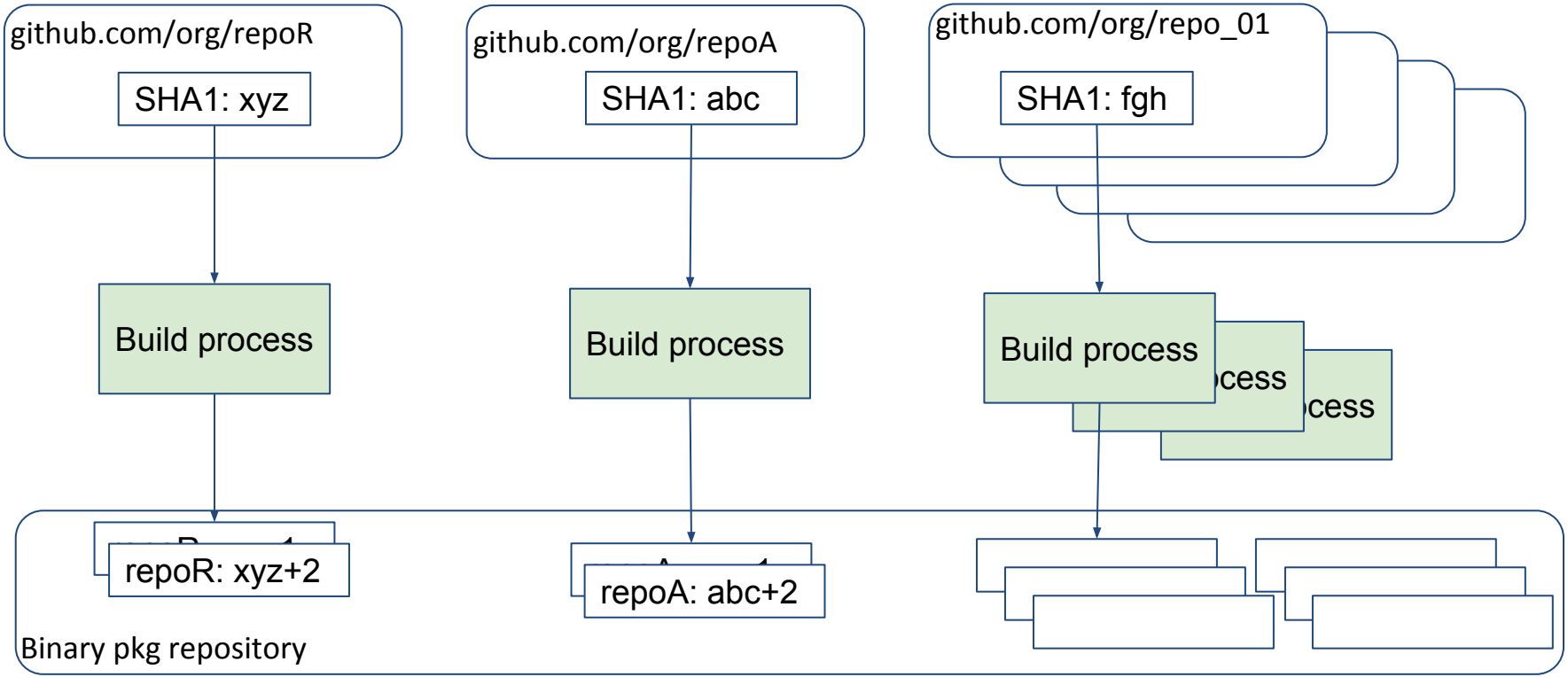
- Each identifiable revision xyz in SW repository repoR, that successfully goes through a build process
  - Build
  - Unit test
  - Packaging

  A binary package can be stored in the binary package repository

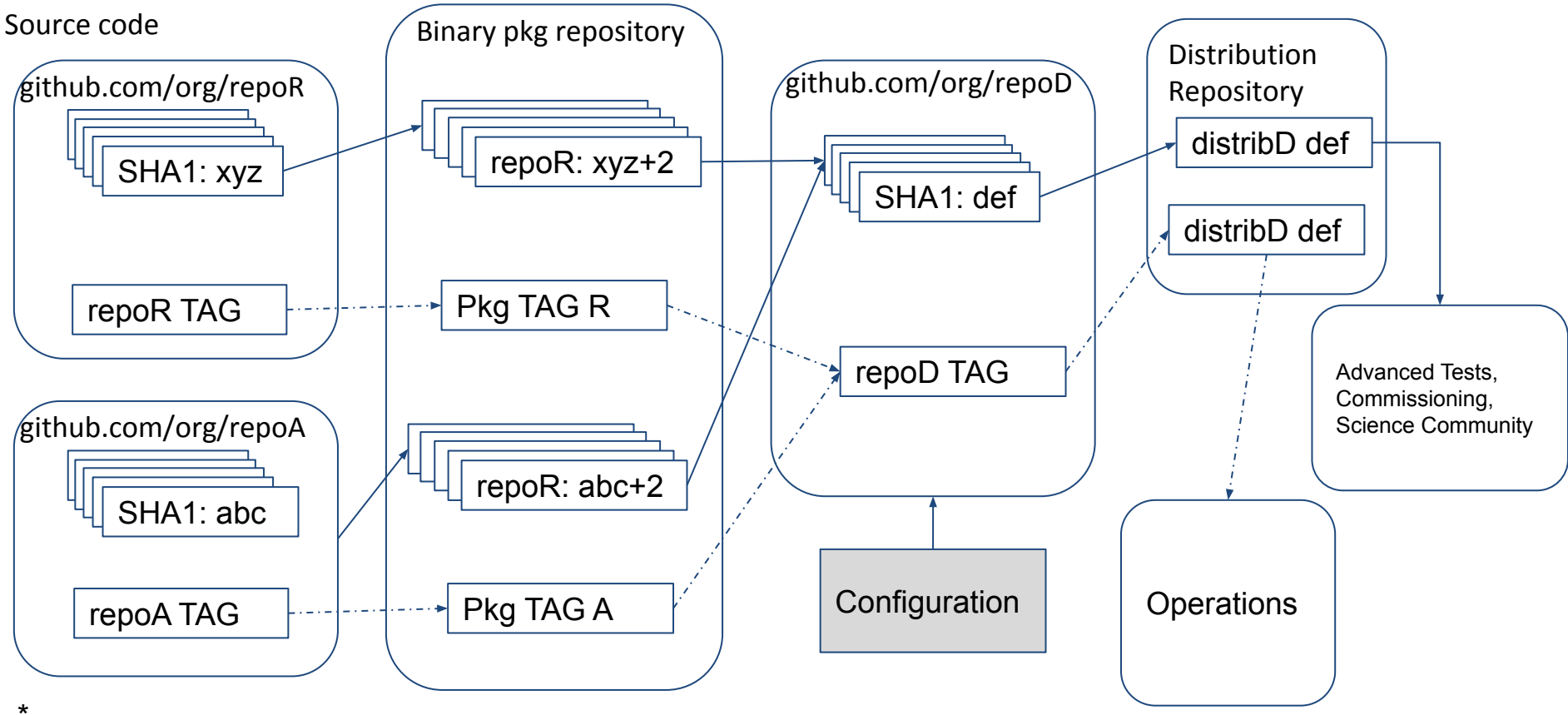
- The easiest way to identify the binary package is to use the repository revision number itself, that is already unique

- But, each revision, can be rebuild multiple time
  - Each build is different (checksum)
  - Each binary package build need to be identified, for example by a progressive build number starting from 1

Example: Binary Package [2]



Example: Overview



Extra Slides

Definition: Metapackage

- It consists in a Git repository that has only dependencies:
  - ups/metaPkgName.table