Collaborative & Automated Packaging

32nd Tango Meeting, Prague 2018

Carlos Falcón
Carlos Pascual-Izarra
Frédéric Picca (SOLEIL)
Zbigniew Reszela
Marc Rosanes
Disclaimer

- This talk will be focused on **Debian** packaging
- ... but most of it is also applicable to other *standard* packaging systems
- ... including Conda
- ... and the workflow can be used to package for various systems *simultaneously*
Why?

- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?

- How to package?
- How to package collaboratively?
- How to package automatically?
Why?

- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?

- How to package?
- How to package collaboratively?
- How to package automatically?
Why packaging?

- Prevents messy installations (scales better)
- Handles dependencies
- Allows reusing packages from others
- Enables using standard tools:
  - pkg repos (e.g. aptly)
  - pkg managers (e.g. apt-get)
  - Docker
  - CI/CD, configuration managers
- Simplifies recruiting experts
- Brings benefits from the distro ecosystem:
  - integration efforts
  - infrastructure (salsa.debian.org)
  - official docs and help
  - multi-platform support
Why packaging?

- Prevents messy installations (scales better)
- Handles dependencies
Why packaging?

- Prevents messy installations (scales better)
- Handles dependencies
- Allows reusing packages from others
Why packaging?

- Prevents messy installations (scales better)
- Handles dependencies
- Allows reusing packages from others
- Enables using standard tools:
  - pkg repos (e.g. aptly)
  - pkg managers (e.g. apt-get)
  - Docker
  - CI/CD, configuration managers
- Simplifies recruiting experts
- Brings benefits from the distro ecosystem:
  - integration efforts
  - infrastructure (salsa.debian.org)
  - official docs and help
  - multi-platform support
Why packaging?

- Prevents messy installations (scales better)
- Handles dependencies
- Allows reusing packages from others
- Enables using standard tools:
  - pkg repos (e.g. aptly)
  - pkg managers (e.g. apt-get)
  - Docker
  - CI/CD, configuration managers
- Simplifies recruiting experts
Why packaging?

- Prevents messy installations (scales better)
- Handles dependencies
- Allows reusing packages from others
- Enables using standard tools:
  - pkg repos (e.g. aptly)
  - pkg managers (e.g. apt-get)
  - Docker
  - CI/CD, configuration managers
- Simplifies recruiting experts
- Brings benefits from the distro ecosystem
  - integration efforts
  - infrastructure (salsa.debian.org)
  - official docs and help
  - multi-platform support
Why?

- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?

- How to package?
- How to package collaboratively?
- How to package automatically?
Why **collaborative** packaging?

- Packaging alone can be hard
Why collaborative packaging?

- Packaging alone can be hard
- Experts can help you

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred
- Prevents duplication
- Promotes good practices
- Again: it scales better
- Infrastructure is available (salsa, OBS, Conda-forge)
- It is free! (even if no one joins you)
Why **collaborative** packaging?

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred

Collaborative & Automated Packaging
Why **collaborative** packaging?

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred
- Prevents duplication
- Infrastructure is available (salsa, OBS, Conda-forge)
- It is free! (even if no one joins you)
Why **collaborative** packaging?

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred
- Prevents duplication
- Promotes good practices
- Again: it scales better
- Infrastructure is available (salsa, OBS, Conda-forge)
- It is free! (even if no one joins you)
Why **collaborative** packaging?

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred
- Prevents duplication
- Promotes good practices
- Again: it scales better

---

Collaborative & Automated Packaging

32nd Tango Meeting, Prague 2018
Why collaborative packaging?

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred
- Prevents duplication
- Promotes good practices
- Again: it scales better
- Infrastructure is available (salsa, OBS, Conda-forge)
Why collaborative packaging?

- Packaging alone can be hard
- Experts can help you
- Efforts can be shared & transferred
- Prevents duplication
- Promotes good practices
- Again: it scales better
- Infrastructure is available (salsa, OBS, Conda-forge)
- It is free! (even if no one joins you)
Why?
- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?
- How to package?
- How to package collaboratively?
- How to package automatically?
Why automated packaging?

- because “Robots are Cool”
- It is a huge time-saver
- Saves boring/repetitive tasks: you focus on the relevant
- It fits well with upstream CI
- Enables Continuous Delivery
- Improves quality
- Improves reproducibility
- Makes the process transparent
- Facilitates collaboration
Why?

- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?

- How to package?
- How to package collaboratively?
- How to package automatically?
How to package (assumptions)

- Both upstream & packagers may be geographically-disperse teams
- Upstream has a public git repo
  - Otherwise, the packaging team may create an “upstream” branch in their own git repo and import upstream releases there
- Packagers have access to a gitlab-ci instance
  - ... Ideally, http://salsa.debian.org
  - ... http://gitlab.com SaaS can also be used
  - ... and private instances / forks are also ok if required
- Packagers and Upstream may (or not) be the same people
  - ...and the upstream & packaging repos may (or not) be the same
How to package

- For Debian
  - add a debian dir to the upstream source. It should contain:
    - debian/control (dependencies, descriptions and other metadata)
    - debian/rules (makefile-style recipe for how to create the package)
    - debian/changelog (package versioning)
    - debian/copyright (licensing info)
    - debian/...
  - run dpkg-buildpackage
  - run Q/A checks (lintian, piuparts, reprotest,...)
  - sign and upload

- For Conda
  - create a conda recipe dir (with meta.yaml, etc.)
  - run conda-build
  - ...
How to package (notes on versions)

- If the upstream developers do not change the version, then the packager must do it.

1. If the upstream developers do not change the version, then the packager must do it.

```plaintext
master ffffff
1.2.3

foo_?????-1

foo_1.2.3-1
```

- The `upstream` version is the version from the upstream repository.
- The `debian` version is the version used in the Debian distribution.

- If the upstream version is `1.2.3`, then the packager must change the version to `foo_1.2.3-1`.
- If the upstream version is `?????`, then the packager must change the version to `foo_?????-1`.

- The packager should ensure that the version in the Debian package matches the upstream version.
How to package (notes on versions)

- If the upstream developers do not change the version, then the packager must do it
How to package (notes on versions)

- local/private (non-official) packages can be treated as backports:

  ![Diagram]

  - This scheme is compatible with official versions (allows correct upgrades if official versions are eventually created)
Why?

- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?

- How to package?
- How to package collaboratively?
- How to package automatically?
How to package **collaboratively**

- **Upstream**
  - `github.com/user/foo`
  - `master`
  - Release foo 1.0
  - Create foo
How to package collaboratively

1. Release foo 1.0
2. Create foo

Upstream:
- github.com/user/foo

Debian Packagers:
- salsa.debian.org/ debit/ foo
- debian/sid

Add debian dir
- d/1.0-1
- u/1.0

Test package:
- foo_1.0-1.deb
How to package collaboratively

Release foo 2.0
(work...)
Release foo 1.0
Create foo

Fix package
Add debian dir

Upstream
github.com/user/foo
salsa.debian.org/debuser/foo

Debian Packagers

debian/sid
foo_1.0-1.deb
foo_1.0-2.deb

master
debian/sid
u/1.0
2.0
1.0
d/1.0-2
d/1.0-1

Add debian dir
Fix package

Create foo
Release foo 1.0
Release foo 2.0
How to package collaboratively

Release foo 2.0
(work...)
Release foo 1.0
Create foo

Upstream
github.com/user/foo
salsa.debian.org/debuser/foo

Debian Packagers

Adapt debian files
Merge new upstream version
Fix package
Add debian dir

Upstream
Debian Packagers
Why?

- Why packaging?
- Why collaborative packaging?
- Why automated packaging?

How?

- How to package?
- How to package collaboratively?
- How to package automatically?
How to package automatically (gitlab-ci pipelines)

- A gitlab-ci pipeline is a sequence of stages. Each stage executes one or more jobs in parallel.
- Each job is executed by a gitlab runner, which typically spawns a disposable docker container.
- Our pipeline is described in `.gitlab-ci.yml` (or in `debian/.gitlab-ci.yml` if we only want packaging).

```
  ● foo_VER.tgz
  ● foo-VER.whl
  ● docs/html

  ● foo_VER-1.deb
  ● foo_VER-1 deb-src
  ● foo_VER-1~bpo9+1.deb
  ● foo_VER-1~bpo9+1 deb-src
```
How to package **automatically** (upstream CI)

- A push to a **non-debian/*** branch triggers the following stages:

**UPSTREAM-BUILD**

**BUILD_UPSTREAM**

- `foo_custom:stretch`
  - `setup.py sdist`
  - `setup.py bdist_wheel`
  - `setup.py build_sphinx`

**PREPARE_DEBREPO**

- `debian:stretch`
  - `create tmp_repo`
  - `VER=x.y+git<date>.<build_id>.<hash>`
  - `tag u/VER`
  - `merge u/VER into debian/*`
  - `update debian/changelogs (gbp dch)`

**UPSTREAM-TEST**

**TEST_DEVEL**

- `foo_custom:stretch`
  - `setup.py develop`
  - `setup.py test`

**TEST_SDIST (1)**

- `foo_custom:1`
  - `untar foo-VER.tgz`
  - `setup.py test`

**TEST_SDIST (2)**

- `foo_custom:2`
  - `...

Artifacts

- `foo-VER.tgz`
- `foo-VER.whl`
- `docs/html`
- `tmp_repo`
How to package automatically (debian packaging)

- If UPSTREAM-TEST succeeds (or if a push is done to a debian/* branch) the following stages take place:

**BUILD-PKG**

**BUILD SID**

- `debian:sid`
- `>apt-get build-dep`
- `>gbp-buildpackage`
- `>tag d/x.y-1`

**BUILD STRETCH**

- `debian:stretch`
- `>apt-get build-dep`
- `>gbp-buildpackage`
- `>tag d/x.y-1~bpo9+1`

**Artifacts**

- `foo_x.y-1.deb`
- `foo_x.y-1 src`
- `foo_x.y-1~bpo9+1.deb`
- `foo_x.y-1~bpo9+1 src`
- `tmp_repo`

**TEST-PKG**

**LINTIAN SID**

- `debian:sid`
- `>lintian`

**LINTIAN STRETCH**

- `debian:stretch`
- `>lintian`

**PIUPARTS**

- `debian:sid`
- `>piuparts -d sid`
- `>piuparts -d stretch`
- `...`

**PUSH-REPO**

**PUSH TO PKG REPO**

- `alpine:latest`
- `>cd tmp_repo`
- `>push salsa d/sid`
- `>push salsa d/stretch`
- `>push salsa --tags`

**PUSH TO REPO (1)**

- `manual confirmation`
- `manual confirmation`
How to package **automatically** (deployment)

- After the new package becomes “blessed” (via manual trigger of PUSH_REPO) the deployment part of the pipeline is enabled:

```
DEPLOY-STAGING

UPLOAD_TO_STAGING  debian:stretch
  > aptly upload staging

DEPLOY-PRODUCTION

PROMOTE_TO_PRODUCTION
  manual confirmation
  debian:stretch
  > aptly move staging → production

UPLOAD_TO_PYPI
  manual confirmation
  python:latest
  > twine upload foo*.tgz foo*.whl
  ...
```
Should we all join forces?

• Many people from Tango institutes are already maintaining deb packages:
  – Frédéric Picca (Soleil), Sandor Bodo-Merle (ELI-ALPS), Eugen Wintersberger (Desy), Jerome Kieffer & Stuart James (ESRF), Teemu Ikonen(?), ~10 packagers at ALBA, ...

  (I am surely missing many... sorry... one more reason for coordinating!)

• Some examples of packages that we could maintain collaboratively:
  – tango, tango-java, pytango, tango-matlab-binding, taurus, sardana, fandango, pytangoarchiving, silx, pymca, pyfai, ufo-filters, pynx, lima, panic, hdb++ DS, scpi, tangovimba, imggrabber&IBA DSs, moxa-npreal2, pylon, modbus, ros, pystatecomposer, serial DS, vacca, yat, yat4tango, pyplc, omniORB, ISL, ...

• ALBA will be locally maintaining packages for most of its own code:
  – ~250 GUIs, ~200 DS, ~100 sardana macros & controllers, ~80 libs, ~30 kernel drivers,...

let's contribute with our small drop in an ocean of ~60K official Debian packages

GOAL: at least 1 Debian Developer per institute, and joining debian-Science and/or debian-Astro teams
Conclusions & final remarks

- We propose a **simple-but-general pipeline** that can be run on publicly available GitLab-CI services.

- Institutes can **collaborate on a common package** and at the same time keep a fork with their own customizations (Git merge and cherry-pick are your friends).

- This workflow **can be used for all packages** (even for private ones).

- **ALBA will use this pipeline for all its internal packages** (600+). We will push any of them to public repos (preferably Salsa) if there is someone interested.
**.gitlab-ci.yml example**

```yaml
prepare_debrepo:
  image: cmft/debpack-ci
  stage: ups_build
  artifacts:
    (...)  
  script:
    # create & configure the temporary repo
    - git clone . ${ARTIFACTS_DIR}/tmprepo
    - git push ${ARTIFACTS_DIR}/tmprepo remotes/origin/debian/stretch:refs/heads/debian/stretch
    - git push ${ARTIFACTS_DIR}/tmprepo remotes/origin/debian/sid:refs/heads/debian/sid
    # exit here if the push was done to a debian branch
    - if 
      [[ $CI_COMMIT_REF_NAME =~ ^debian/.*$ ]]
      then echo "We are already in a debian branch. Nothing more to do." ; exit 0; fi
    - cd ${ARTIFACTS_DIR}/tmprepo
    - git checkout -b upstream
    - git config user.name $DEBFULLNAME
    - git config user.email $DEBEMAIL
    # tag upstream
    - UPVERSION=`python setup.py --version`+git`date +%Y%m%d`.$CI_PIPELINE_ID.`git rev-parse --short=6 upstream`
    - git tag upstream/$UPVERSION
    # merge and update debian/sid branch
    - git checkout debian/sid
    - git merge -X theirs --no-ff --no-edit -m "Merge upstream/$UPVERSION into debian/sid" -m "[ci skip]" upstream/$UPVERSION
    - gbp dch --commit --commit-msg="Update changelog for %(version)s release [ci skip]"
      --spawn-editor=never --debian-branch=debian/sid --upstream-branch=upstream
      --release --new-version=$UPVERSION-1 --git-log='-first-parent'
    - cat debian/changelog
    # merge and update debian/stretch branch
    (...)  
  except:
    - tags
```

**Collaborative & Automated Packaging**

32nd Tango Meeting, Prague 2018
.gitlab-ci.yml example

```
build_package_sid:
  image: debian:unstable
  stage: pkg_build
  dependencies:
    - prepare_debrepo
  artifacts:
    expire_in: 1 day
    name: "$CI_JOB_NAME"
    paths:
      - ${ARTIFACTS_DIR}/sid
      - ${ARTIFACTS_DIR}/tmprepo
  before_script:
    - apt-get update
    - apt-get install eatmydata -y
    - eatmydata apt-get install git-buildpackage -y
  script:
    - cd ${ARTIFACTS_DIR}/tmprepo
    - git checkout debian/sid
    - eatmydata apt-get build-dep -y
    - DEB_BUILD_OPTIONS=nocheck gbp buildpackage --git-tag --git-debian-branch=debian/sid
      --git-export-dir=${ARTIFACTS_DIR}/sid -us -uc
  except:
    - tags
```
.gitlab-ci.yml example

```
lintian_sid:
  image: debian:unstable
  stage: pkg_test
  before_script:
    - apt-get update && apt-get install lintian -y --no-install-recommends
  script:
    - lintian -i ${ARTIFACTS_DIR}/sid/*.changes
  except:
    - tags

push_to_pkg_repo:
  image: alpine:latest
  stage: pkg_push
  dependencies:
    - build_package_sid
    - build_package_stretch
  before_script:
    # prior to uploading, push the repo changes
    - apk update && apk add git
  script:
    - cd ${ARTIFACTS_DIR}/tmprepo
    - git push https://$BOT_NAME:$BOT_PASS@$DEPLOY_GIT_REPO debian/sid:debian/sid
    - git push https://$BOT_NAME:$BOT_PASS@$DEPLOY_GIT_REPO debian/stretch:debian/stretch
    - git push https://$BOT_NAME:$BOT_PASS@$DEPLOY_GIT_REPO --tags
  when: manual
  except:
    - tags
```