Dependency management: Lessons learned

Simon Heybrock
simon.heybrock@ess.eu

European Spallation Source / DMSC / 2022-05-11

Context

- Developing Python and/or C++ libraries and applications
- Using GitHub (or similar) with Github Actions (or similar) to:
  - Build and run tests for pull requests (PRs).
  - Build documentation.
  - Build conda packages.
  - Build PyPI wheels.

Problem

Example: scipp, scippnexus, scippneutron, ess

- Experienced issues in about 10 cases in 2021, such as:
  - Pull request builds suddenly fail for no apparent reason.
  - Documentation builds stop working.
  - Package builds don't work any more.
- . . . despite previously passing CI, issue is *not* a bad merge.

Previous approach

- Waste time discussing the "weird" issue at the daily standup.
- Dig through build logs: which package upgraded?
- Does older version work? Check upstream for issue, or open one.
- Pin version, restart all builds.
- Open issue as a reminder to remove the pin once upstream made a fix.
- Add pin across all projects.
- Remove pins once possible, half the time it is unclear how to reproduce.

Concrete examples from 2022

Five such issues is less than four months!

February 18	Conan suddenly produces an error, caused by update of `markupsafe` `https://github.com/pallets/markupsafe/issues/282`. Lost ability to make a (patch) release until we found this and pinned the version.
March 24	Documentation builds stop working due to `https://github.com/jupyter/nbconvert/issues/1736` causes by `https://github.com/pallets/jinja/issues/1626`. Lost ability to pass CI (blocking all merges) as well as blocking release builds (which build and update docs).
April 1	We notice that navigation buttons in documentation pages have disappeared. The `sphinx-book-theme` 0.3.0 release causes this.
April 7	`scikit-build` releases 0.14.0, breaking our PyPI builds. Lost ability to make a (patch) release until we found this and pinned the version.
April 19	We notice that plots no longer show up in documentation pages, traced to an `ipympl` update `https://github.com/matplotlib/ipympl/issues/462`.

By continuously upgrading, we experience *almost every upstream issue*, even if it is fixed without hours or days.

Recommended reading

Approach

- Step 1: Read `https://hynek.me/articles/semver-will-not-save-you/`. Recommends to freeze everything, including transitive dependencies.
- Step 2: Conclude that we cannot do that because if we do that for a *library* almost no one will be able to use it.

Recommended reading

Approach

- Step 1: Read `https://hynek.me/articles/semver-will-not-save-you/`. Recommends to freeze everything, including transitive dependencies.
- Step 2: ~~Conclude that we cannot do that because if we do that for a *library* almost no one will be able to use it.~~
- Step 3: Realize that many or most of our dependency issues are not issues that ever reach a user. Actually we can freeze for quite a few cases:
  - Building documentation.
  - Requirements need for building packages.
  - Pull-request builds.
- Step 4: Must read for more details and better understanding: `https://iscinumpy.dev/post/bound-version-constraints/`

Implementation

Requirement: All builds should be *reproducible*

Reproducible means:

- Builds for a commit that passed previously should continue passing if we re-run it later.
- Builds should fail *only* when there is a problem in the *current* source repository, *not* if a dependency or transitive dependency makes a release.

This requirement applies to "everything":

- Going back to much older version, e.g., to make a patch release.
- Documentation builds (Sphinx).
- Release builds (making packages for PyPI and conda).
- PR builds.
- Builds on `main` after merging a PR.
- ...

Workflow

Setup (example for Python-based project with `pip` workflow)

1. Write `requirements.in` file(s). Do this for *everything* this is needed for your builds.
2. Use `pip-compile` from `pip-tools`. See also `pip-compile-multi`. Resulting `requirements.txt` is *checked into version control*.
3. Treat *warnings as errors* in CI, in particular `DeprecationWarning`
   - Avoids issues from *intentional* breaking changes, since we can take action early.

From time to time

- Re-run `pip-compile`, run all the builds (including release builds!) and check that things work, merge.
  - `dependabot` may be an alternative, but it updates individual requirements, which seems too noisy for us, with too little control.
- Make patch-releases when a dependency update causes issues for users
  - Much rarer than issues we experience in development.
  - Treating `DeprecationWarning` as errors should catch many issues months/years ahead.
  - Users can freeze dependency version *when setting up their environment* as a immediate workaround.

Status

- We (scipp and subprojects) do *not* have long term experience with this yet:
  - So far implemented in `https://github.com/scipp/scippnexus` and `https://github.com/scipp/scipp` (excluding C++ build dependencies).
  - Have not gone through update cycles (re-run of `pip-compile`) and do not know how many headaches this will cause.
- Our `conan` dependencies have been pinned since the beginning.
- `conda` dependencies are not pinned yet. What is best use of tools for keeing this up to date, including pins of transitive dependencies?
  - Does the `conda-forge` global pinning help? See `https://conda-forge.org/docs/maintainer/pinning_deps.html`.

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.
- The builds do not pass, since a completely unrelated test is failing (because there was a unrelated dependency update). Alice is confused.

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.
- The builds do not pass, since a completely unrelated test is failing (because there was a unrelated dependency update). Alice is confused.
- Bob, the project maintainer, figures out that what the issue is, explains it in the PR, fixes the issue, merges into `main`, merges `main` into the PR, explains that all is good now, ...

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.
- The builds do not pass, since a completely unrelated test is failing (because there was a unrelated dependency update). Alice is confused.
- Bob, the project maintainer, figures out that what the issue is, explains it in the PR, fixes the issue, merges into `main`, merges `main` into the PR, explains that all is good now, ...
- Alice is frustrated by the delay and noise and moves on.

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.
- The builds do not pass, since a completely unrelated test is failing (because there was a unrelated dependency update). Alice is confused.
- Bob, the project maintainer, figures out that what the issue is, explains it in the PR, fixes the issue, merges into `main`, merges `main` into the PR, explains that all is good now, ...
- Alice is frustrated by the delay and noise and moves on.
- Bob is frustrated because he has to do things like this all the time.

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.
- The builds do not pass, since a completely unrelated test is failing (because there was a unrelated dependency update). Alice is confused.
- Bob, the project maintainer, figures out that what the issue is, explains it in the PR, fixes the issue, merges into `main`, merges `main` into the PR, explains that all is good now, ...
- Alice is frustrated by the delay and noise and moves on.
- Bob is frustrated because he has to do things like this all the time.

Case study: Should we freeze dependencies in PR builds?

Temptation: *Use latest version of library dependencies to be alerted early of upstream changes*

Would this be a good idea?

- A new contributor, Alice, opens a PR with a small bugfix.
- The builds do not pass, since a completely unrelated test is failing (because there was a unrelated dependency update). Alice is confused.
- Bob, the project maintainer, figures out that what the issue is, explains it in the PR, fixes the issue, merges into `main`, merges `main` into the PR, explains that all is good now, ...
- Alice is frustrated by the delay and noise and moves on.
- Bob is frustrated because he has to do things like this all the time.

Better solution: Freeze all dependencies for PR builds to *decouple* the tasks

- A new contributor, Alice, opens a PR with a small bugfix. Builds pass and it gets merged. Alice is happy and may continue contributing.
- *Independently*, Bob notices that a new upstream release requires a fix. He re-runs `pip-compile` and fixes the issue in a PR. After a merge, all future PR builds will now use the new upstream version.