Python Monorepos: What, Why and How

Benjy Weinberger
Maintainer, Pants Build

PyCon Israel 2021

About me

- 25 years' experience as a Software Engineer.
- Worked at Check Point, Google, Twitter, Foursquare.
- Maintainer of the Pants OSS project.
- Co-founder of Toolchain.

Overview

1. What is a monorepo
2. Why would I want one?
3. Tooling for a Python monorepo

1. What is a monorepo?

A common codebase characteristic

They

g r o W

over time.

A common consequence of growth

Builds get harder: slower, less manageable

Two ways to scale your codebase

Multi-repo vs. Monorepo

Multi-repo

Split the codebase into growing numbers of small repos, along team or project boundaries.

Monorepo

A *monorepo* is a unified codebase containing code for multiple projects that share underlying dependencies, data models, functionality, tooling and processes.

monorepo != monolithic server

Monorepos are often *great* for microservices.

2. Why should I want a monorepo?

Multi-repo kinda sounds better at first

More decentralized. More bottom-up.

I can do my own thing in my own repo.

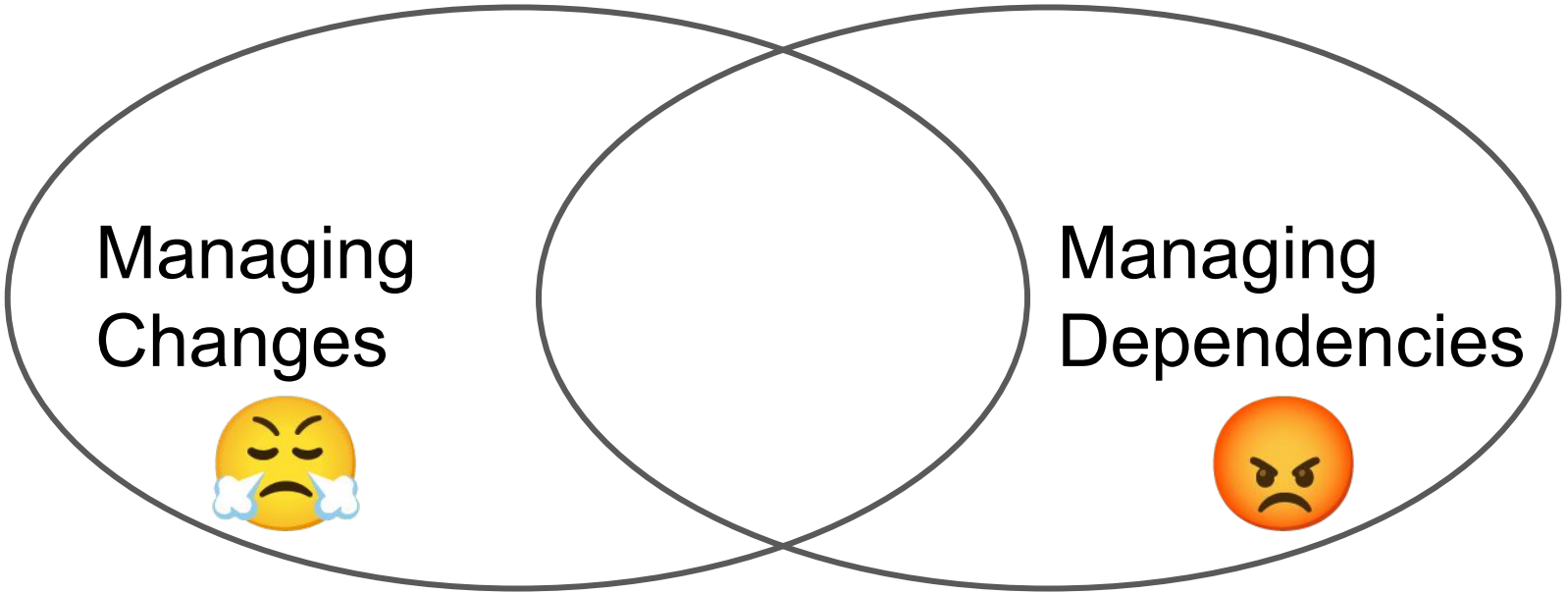
But, for some core problems...

Multi-repo doesn't solve them.

It hides them.

And it creates new ones.

The hardest codebase problems are...



Multi-repo relies on publishing

For code from repo A to be consumed by other repos, it must publish an artifact, such as an sdist or wheel.

`A-1.2.0`

Multi-repo relies on versioning

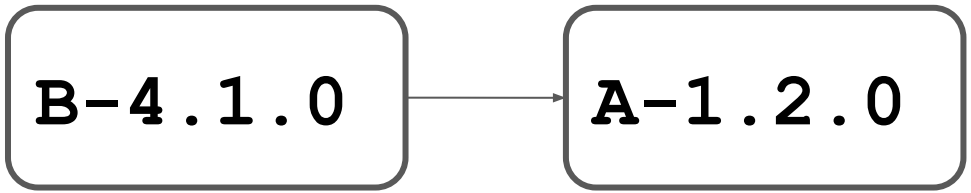
When repo A makes a change, it has to re-publish under a new version.

`A-1.2.0`

`A-1.3.0`

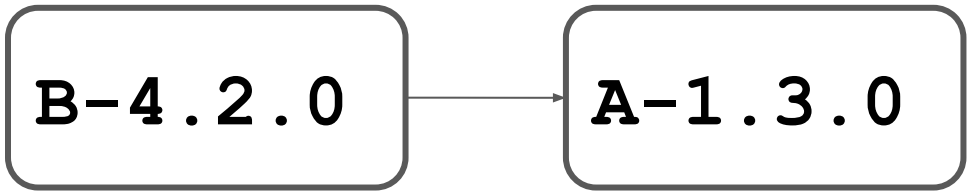
Say repo B depends on repo A

It does so at a specific version:



When repo B needs a change in repo A

Modify A, publish it at a new version, and consume that new version in a new version of B.



Now, you have two choices...

Change management: virtuous choice

1. Find all the consumers of repo A
2. Ensure that they still work at `A-1.3.0`
3. Make changes as needed until tests pass
4. Repeat - recursively! - for all repos you changed

Change management: lazy choice

Don't worry about the other consumers of repo A.

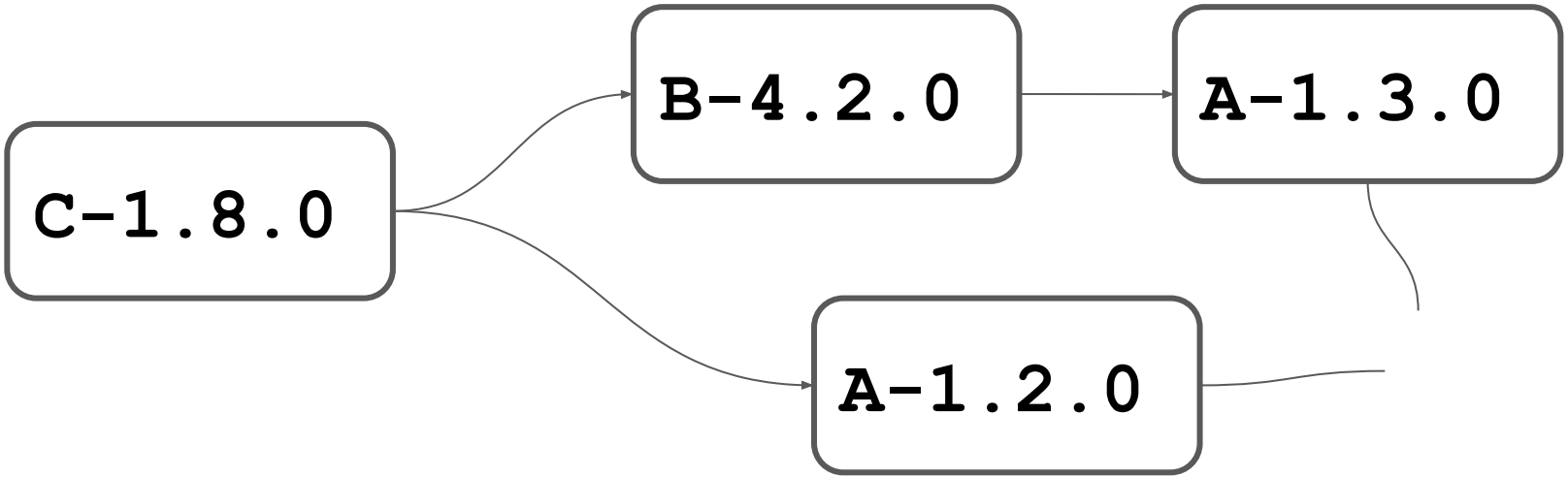
After all, they're safely pinned to `A-1.2.0`.

Let them deal with the problems when they upgrade.

But...

Dependency hell

This causes a huge dependency resolution problem.



But in a monorepo

There is no versioning or publishing.

All the consumers are right there in the same repo.

Breakages are immediately visible.

Monorepos are more flexible

Easier to refactor

Easier to debug

Easier to discover and reuse code

Unified change history

Your codebase ➜ your organization

Balkanized codebase ➜ balkanized org

Unified codebase ➜ unified org

3. Tooling for a Python monorepo

Build Performance At Scale

Standard Python tools not designed for monorepos.

Small changes trigger full rebuilds.

As your codebase grows, so do your build times.

How to speed things up

Do less work

- Fine-grained invalidation
- Caching

Do more work at once

- Concurrency
- Remote execution

What kind of tooling has these features?

To work effectively, you need a build system designed for monorepos.

It sits on top of existing standard tooling, and orchestrates them for you.

Examples of such tools include

Pants

Bazel

Buck

How do these tools work?

- Goal-based command interface
- Build graph metadata
- Extensible workflow with no side-effects

Goals

A monorepo build system typically supports requesting *goals* on specific inputs.

```
$ pants test src/python/foo/bar/test.py

$ pants package src/python/foo/**

$ pants lint fmt --changed-since=HEAD
```

Code dependencies

A monorepo build system requires extra metadata to describe the *build graph*: the units of code and the dependencies between them.

Task dependencies

A monorepo build system maintains the *rule graph*: The units of work and the dependencies between them.

Custom rules can be plugged in, for extensibility.

Build workflow

Code dependencies + task dependencies = workflow.

Workflow recursively maps initial inputs to final outputs - the goals the user requested.

Workflow is side effect-free.

The explicitly-modelled workflow enables

- fine-grained invalidation
- caching
- concurrency
- remote execution

Which is what makes builds scale with your codebase!

Summary

- Monorepos are an effective codebase architecture
- They require appropriate tooling for performance and reliability at scale
- This tooling exists!

Thanks for attending!

I'll be happy to take any questions.

You can also find us on https://www.pantsbuild.org/