Dependency Inference: Caching, concurrency, and remoting without the boilerplate

As discussed [in our last post](https://blog.pantsbuild.org/introducing-pants-v2/), it's clear that Python has "grown up" by gaining facilities to help it to scale to larger projects. But as codebases grow and tool counts increase, it’s become clear that more users of Python need build tools like Pants v2. While you could write bespoke scripts to support each repo, using Pants brings benefits like caching, concurrency, introspection, a simple and uniform user experience, and more!

Unfortunately, scalable build tools have historically meant a significant boilerplate burden: scattering `BUILD` files throughout your repository and then needing to edit both your code and the redundant dependency information in build definitions.

But it doesn’t have to be that way! Pants v2 supports all of the caching, concurrency, and introspection you need to scale your repo, with significantly less boilerplate, thanks to… Dependency Inference!

Scaling up

Pants supports repos of all sizes with minimal boilerplate, but it is particularly helpful in repositories containing multiple deployable or publishable projects, each with potentially different requirements or interpreters: aka, monorepos.

Monorepos have lots of benefits (no dependency conflicts, atomic cross-project commits, easy top-to-bottom continuous integration, linear change history), but critical to making them scale are the abilities to:
1. test, check, and deploy precisely the portion of the repository that is relevant to you
2. cache builds and tests to avoid re-building when unrelated code has changed
3. manage and configure the variety of tooling that users of the repository will want to use

It's critical in a monorepo to be able to test, check, and deploy exactly the relevant portion of your code, ideally with zero impact from changes in unrelated parts of the codebase. For example: if you have three libraries `A`, `B`, and `C`, which depend on one another in a chain like `A -> B -> C`, a monorepo that builds from source using Pants allows you to completely ignore versioning (and `setup.py` files, per-project `requirements.txt`, etc) while you edit library `C`, even if you are running the tests for library `A`. If you have ten other projects (or one thousand!) in your repository and only a few of them depend on `C`, you'd like to avoid ever running tests or mypy for the unrelated libraries while editing `C`.

And in those cases when you _do_ want to take advantage of a monorepo's top-to-bottom integration testing by running "all of your dependent's tests" (or maybe just typecheck them with Mypy!), you'd like to do that as quickly as possible by taking advantage of caching, concurrency, or transparently executing them on a cluster of machines using remote execution.

To enable this scalability, monorepo build tools like Pants v1 and Bazel generally require that the dependencies between libraries and files are declared in `BUILD` files. These dependencies are then used to determine which portions of the repository need to be built, and which files need to be included in cache keys.

"But wait", you say! "Doesn't that mean we've traded editing `setup.py` and `requirements.txt` files for every library for editing `BUILD` files for every library?" In most tools, that would be the case: but not in Pants v2! Pants v2 is different.

Dependency inference

Pants almost entirely eliminates the boilerplate of declaring dependencies between libraries thanks to "Dependency inference". Dependency inference is roughly what it sounds like: Pants supports discovering the dependencies between Python libraries by parsing `import` statements (and you can [use our powerful plugin API](https://www.pantsbuild.org/docs/plugins-overview) to infer other dependencies, such as by parsing a Django settings file, YAML config, etc).

Rather than adding an import statement to your code resulting in your tests failing because you forgot to _also_ update a `BUILD` file, Pants will use your newly added import statement to infer that that file now has a dependency within the repository (or outside it via your `requirements.txt`). When designing inference, we strove to remove boilerplate without introducing magic, so `BUILD` files are still used to declare any metadata your library might have (the version of the Python interpreter to use, etc), and can be used to [override or extend](https://www.pantsbuild.org/docs/targets#dependencies-and-dependency-inference) the inferred dependencies.

Even better, Pants infers these dependencies _at the file level_. Rather than adding a dependency from "the library named `A`" to "the library named `B`" (or "target" in monorepo parlance), Pants tracks that "file `a.py` depends on file "`b.py`". Rather than staring at the content of your `BUILD` files to (attempt to?) understand your dependencies, you can use Pants' dependency introspection tools to easily explore them at the file level: `./pants dependencies $file`.

In practice, we've found that inferred file-level dependencies can reduce the total per-file dependency count by an average of **30%** (improving cache hit rates), and reduce the size of `BUILD` files by up to **90%** (reducing boilerplate)! See our docs for [a real world example](https://www.pantsbuild.org/docs/how-does-pants-work#dependency-inference).

And critically (as we'll discuss in further posts!), dependency inference is both 1) very safe, and 2) very fast. Because Pants invokes processes hermetically using SHA256 fingerprinting and strong sandboxes (your test frameworks, your linters, mypy, everything), failing to infer a dependency can never cause the wrong things to be cached. And because the core of Pants is

implemented in Rust, and uses a daemon, parallelism, and very-fine-grained memoization, inference won't slow you down!

Demonstration

To show what it's like to use dependency inference, we'll quickly add a feature with assistance from some new first and third party dependencies (from sources and PyPI, respectively).

We'll start with a broken test that expects TOML files to be supported by a library in a different directory:

```
1. demonstrate the layout and show a failing test
       fromfile support for toml.
           src/python/pants/option/parser.py
           src/python/pants/backend/python/lint/isort/rules_integration_test.py

./pants test
src/python/pants/backend/python/lint/isort/rules_integration_test.py -- -k
test_respects_passthrough_args
```

Because we're curious, we'll start by asking Pants whether the library already (directly) depends on TOML:

```
2. show the direct dependencies of the library

./pants dependencies src/python/pants/option/parser.py
```

Nope! Only YAML. But let's add the import statement and see what happens…

```
3. show the direct dependencies of the library and highlight TOML

git apply --verbose add-toml-import.diff
git
./pants dependencies src/python/pants/option/parser.py | grep --color -C100
'toml'
```

Voila! This file now declares a dependency on TOML via the import statement, without any modifications to `BUILD` files. But we're not finished: neither the flake8 linter nor the customer will be satisfied with an unused import statement! Let's finish adding the feature, and then re-run the test.

```
4. add code that uses TOML and re-run the failing test

git apply --verbose parse-toml.diff
```

```
./pants test
src/python/pants/backend/python/lint/isort/rules_integration_test.py -- -k
test_respects_passthrough_args
```

Ok, TOML is clearly being used: but this time our test needs tweaking. To fix it we can import test helper code from a second library (another new dependency!). We already know that we don't need to check the `BUILD` file to see whether this test declares a dependency on the library, so we can just focus on our code!:

```
5. then add a first party dependency to the test and re-run

git apply --verbose produce-toml-in-test.diff
./pants test
src/python/pants/backend/python/lint/isort/rules_integration_test.py -- -k
test_respects_passthrough_args
```

Great! Our test is now passing. To fix it, we edited two files and introduced two new dependencies -- but we didn't need to edit any `BUILD` files, despite these files living in different targets!

Finally, let's confirm the critical monorepo scalability property that edits to unrelated files don't invalidate the caching of our new test. Although this test is quick, there are a few hundred files in this repository, and plenty of other tests that could take long enough to result in coffee breaks!:

```
6. edit a file unrelated to that test, show that we still noop at the end, and
show that all of this occurred in the context of a large repository

git apply --verbose add-unrelated-comment.diff
./pants test
src/python/pants/backend/python/lint/isort/rules_integration_test.py -- -k
test_respects_passthrough_args
./pants list '**/*' | wc -l
```

Excellent, thanks to dependency inference's file level precision, our test is still quickly and correctly cached, even after editing neighboring files in the target! Productivity preserved; mission accomplished.

Conclusion

The precise, always-accurate dependencies in Pants have a lot of benefits:
1. it's really, really easy to get started with Pants!

2. `BUILD` files (or `requirements.txt`/`setup.py` files) can't go out of sync with the code, because you don't need to repeat all of your import statements there
3. the cache keys for processes (and thus the number of cache hits and amount of rebuilding) are more accurate and much more fine-grained than you would ever write by hand

If you're interested in speeding-up and scaling-up your builds without the boilerplate, the [Pants community would love to help](https://www.pantsbuild.org/docs/community)!