Poison wheels: a Klaviyo continuous integration fire-fighting story

Author: Zac Bentley

Claps: 146

Date: May 2

It was a Wednesday morning, and while production wasnâ€™t precisely *down*, things werenâ€™t looking good. In one of our main repositories/applications, nobody could deploy code. Attempts to do so would cause any hosts targeted for deployment to fail to start.

This occurrence isnâ€™t terribly rare at Klaviyo. Occasionally, builds get broken and bad changes get shipped. When that happens, deploys to the affected service are halted, we roll back (and pin) production to the last known good deployable artifact version, go find the offending change, revert it, make sure things are good, and resume business as usual.

What was weird *this* time was that there was no offending change. After a certain point *in time*, all artifacts built by our build system (on any branch or commit) would, upon being deployed, fail to start.

More precisely, they would fail during initial `import` statements, on the first import of any compiled C extension moduleâ€“it didnâ€™t matter which module was first, it would fail to import with this error:

```
OSError: libpython3.7m.so.1.0: cannot open shared object file: No such fil
```

Again, this happened on all builds. We even resurrected a two-week-old, known good branch and ran it through the build system. Sure enough, production services failed to start on that branchâ€™s artifact with the same error.

Because there was nothing to roll back, we could not allow merges/deployments to resume. Production was pinned to the last known good artifact, developers were getting frustrated, and the queue of people and pull requests waiting to ship changes resembled (in length and urgency) the line outside the bathroom at the end of a show.

The problem, it seemed, was somewhere in our build system. Discovering what had gone wrong and how to fix it was somewhat complex, and yielded many useful and generally-applicable lessons in how to make high-quality CI/CD systems.

Because I hate clickbaity â€œdetective storyâ€ longreads that force you to follow the thought process of the people who debugged an issue when Iâ€™m just interested in the punchline, here are the spoilers:

> *A changed value in the build system we use for the Python interpreter itself resulted in C extensions being compiled during our artifact preparation process linking against an embedded Python library. A significant issue in Pip, combined with some less-than-rigorous caching of artifacts in our CI system, resulted in built artifacts being â€œpoisonedâ€ with other buildsâ€™ bad binary wheelsâ€“wheels which, when run against Python interpreters built and cached before the rollback, failed with a runtime linker error, breaking production.*

If youâ€™re here because you Googled something along the lines of `libpython3.7m.so.1.0: cannot open shared object file` and you just want a fix, the problem may lie with the Python interpreter youâ€™re using, and you may need to recompile either it, your C extension Pip distribution(s), or both so that they both agree on the `--enable-shared` CPython config flag (inspectable in Python via`sysconfig.get_config_vars(â€Py_ENABLE_SHAREDâ€™)`).

If, on the other hand, that piques your interest, or if youâ€™re curious about the many lessons on building and deploying software we learned while chasing it down, read on.

What changed?

If nothing changed in the application code to cause the issue, we reasoned that something must have changed in the CI system. First, we took a hard look at all changes to the CI harness code that had gone out around when the breakage occurred. Nothing looked particularly suspicious.

Then we looked at infrastructure changes: our CI system runs Jenkins; had any infrastructure updates been made to Jenkins itself, or any of the other software on the agent nodes, around when the breakage started? After some digging (and cautionary reverts of a few things that turned out to be unrelated), we came up empty here, too.

Floating pins considered harmful

Like many projects, Klaviyo pins our dependencies via a lockfile (in this case, one managed via pip-tools). Since all of our broken repoâ€™s dependencies (and their dependencies, and their dependenciesâ€¦) were pinned to fixed versions, our build *should* have been deterministic.

Before putting on our tinfoil hats and suspecting our supply chain, we first decided to test that assumption. Sure, the code in the application being deployed had a deterministic set of Python dependencies, but what about the interpreter itself?

Here, we chased a bit of a red herring. When we looked at the Dockerfiles that comprised our CI system, we saw that many of the files that set up the environment for our build/test suite and eventual deployment to production had â€œfloating pinsâ€; that is, they had statements like `FROM ubuntu:18.04`, or worse, `FROM debian:latest`.

Interestingly, that wasnâ€™t the root of the issue, though it was a bad practice that weâ€™re glad we discovered and addressed during this incident. This wasnâ€™t the root cause for a slightly odd reason: inside our build/deployment containers, we *compile and build the Python interpreter itself* at a specific version (e.g. 3.10.3), store it at a custom path inside each container, and run all of our package installations, tests, and application code against that interpreter.

Aside: our decision to do this at all is somewhat unusual. In containerized environments, itâ€™s customary to use the Python that comes with a containerâ€™s base image (Docker, for example, provides many containers for just this purpose, so you can write Dockerfiles that are `FROM python:3.10.13`). However, not all of our deployment targets are containerized environments; Linux VMs also run the application artifact that was having issues. Because of the presence of those whole-operating-system targets, and because the versions of Python *they* include with their operating systems are both variable and not easy to control, our build system needs to be able to create a â€œhermeticâ€ Python interpreter that can be copied out of the container along with all of its `pip install`ed packages, so that our application can run both inside and outside of containers against a deterministic (-ish) set of interpreter-level and package-level dependencies. This situation is temporary while we move to running this application entirely in containerized environments, at which point we will switch to the ordinary container-provided-Python approach.

Anyway, our containersâ€™ versions of their core system-level stuff was â€œfloatingâ€ (unpinned), but our Python interpreterâ€™s version was fixed. It seemed clear that the problem lay in the `FROM â€¦:latest` statements in our Dockerfiles, so we tried pinning the images involved in our build to hashes that pre-dated the incident. No luck!

Floating Pyenv

Increasingly frustrated (and aware of growing numbers of application developers who were *also* increasingly frustrated), I started reading through our build system code from top to bottom. After a little while, I found something suspicious:

```
RUN git clone https://github.com/pyenv/pyenv.git /path/to/pyenv/clone
```

Thatâ€™s the recommended installation method for Pyenv: a command line utility that helps build Python interpreter versions from source; you say `pyenv install $VERSION` and it takes care of installing a specific python, and integrates with your shell to activate/deactivate that Python in the same way venv helps activate/deactivate sets of packages and include paths to use with that Python. Itâ€™s primarily used in MacOS workstation environments, but also works on Linux.

Internally, it resembles autoconf, but written by someone who never heard of autoconf. Itâ€™s a large bash script which probes the environment for all sorts of things the Python interpreter compilation process might need. With the results of those probes, it runs a *highly opinionated* invocation of Pythonâ€™s configure/make/make-install process.

And our build system was `git clone`ing `pyenv` at the latest `master` versionâ€“a floating pin.

To understand how this related to the issue, three other pieces of context are needed:

1. How `pip install` (mis)handles C extensions.
2. What the linker error in the beginning actually means.
3. How a tech-debt-induced mistake regarding our Pyenv-built interpreters caused a mismatch in package/interpreter expectations in production.

Whatâ€™s in a (wheel) name?

A common misconception is that Python distributions (what most people mean when they say â€œpackages,â€ distributed, these days, as wheels) are built to work with a specific *version* of Python, e.g. â€œpython 3â€ or â€œpython 2.7â€. While thatâ€™s true for pure python packages, C extensions are a lot more complicated. For example, compare the metadata (the components of the names of .whl files) for requests with the metadata for mmh3, a popular C extension. Requests is pure python, and so its wheel name looks like `requests-2.28.2-py3-none-any.whl`, meaning â€œthis package works with any python identifying as a `py3`â€. `mmh3`, on the other hand, has many wheels, with names like `mmh3â€“3.1.0-cp37-cp37m-manylinux_2_17_aarch64.manylinux2014_aarch64.whl`. That means: â€œworks with CPython 3.7, aka `cp37`, aka `cp37m`, on the `manylinux_2_17` platform and on the `manylinux2014` platform, for the â€˜`aarch64`â€™ (ARM) hardware architecture.â€

Conceptually, that forms a 5-tuple: â€œdistribution-name, version, interpreter-version, platform-versions, architecture-versions.â€ If you request a package at a given name (and optionally version), and the package repository (PyPi in the example links above) contains a wheel matching your environmentâ€™s interpreter, platform, and architecture, you can install and `import` that package. This behavior is formally specified and well known in the Python community.

Itâ€™s also a lie.

There are a *lot* more than 5 things about a Python wheel that can cause it to be incompatible with a given target environment.

In pure python, a wheel could be incompatible because it (for example) can only be imported on hosts with a floppy disk drive:

```
# in a package's __init__.py:
import os
if not os.exists('/dev/fd0'):
  raise ImportError('this module should only be installed on systems with
```

Thereâ€™s no way to indicate, in the wheel name/metadata, that this package requires its install environment to have a floppy disk.

For C extensions, things are a lot more complicated. Regardless of what Python the wheel name indicates an extension module is compatible with, many C extensions additionally require:

- Certain externally-provided libraries to be installed on the host (e.g. via `apt-get` or `brew`)
- The system standard libraries to have been compiled in specific ways not encompassed in the â€œplatformâ€ part of the wheel name, and at specific compatibility versions.
- The Python interpreter loading the extension to have been compiled with a specific set of configuration options.

Thereâ€™s no way to indicate any of those things in the wheel name, and thus no way to tell `pip` â€œhey, donâ€™t install that wheel, the environment requesting the install isnâ€™t compatible with the assumptions encoded into the wheelâ€™s binary component.â€

One last fact about Pip is important to note before we move on: if a C extension package doesnâ€™t have a compatible wheel available, but does provide a source distribution, `pip install whatever` will:

1. Download the source distribution.
2. Invoke the host systemâ€™s compiler/linker/etc. toolchain (and whatever else the build system for the distribution in question needs) to compile it into a binary wheel named after the 5-tuple of platform/version specifiers.
3. Install that wheel by copying the compiled files into place in the current Python environment.
4. Cache that wheel in a (usually global) repository of downloaded/locally-compiled wheels to reduce future work.

What is libpython, anyway?

Libpython (the `libpython3.7m.so.1.0` from the original error) is the Python interpreter core packaged up as a shared library (in case you wanted to, say, embed a Python interpreter into a non-Python application without shelling out to `/path/to/python3`).

Whether or not libpython is present with your Python is a function of the *`--enable-shared`*/*`--disable-shared`* flags to the Python interpreterâ€™s build process.

By default, when most of the C extension module build systems out there compile their Python extensions, they will set up the compiled artifacts they generate to link against libpython or not, based on *whether the Python interpreter invoking the extensionâ€™s build system was built with*

`--enable-shared` or not. If you compile and store a binary wheelâ€“any binary wheelâ€“against a libpython-included interpreter, it will need to link against `libpython` when itâ€™s loaded.

Nothing about â€œlinked against libpython-or-notâ€ is indicated in compiled extensionsâ€™ wheel filesâ€™ name metadata. As a result `pip` will happily install a wheel file whose compiled binary files *do* link against libpython for use by a Python interpreter that was compiled to *not* contain libpython. Nothing about `pip install`â€™s successful result will indicate that this mismatch exists, but any attempt to `import` that installed library will fail with, you guessed it, an error like this:

```
OSError: libpython3.7m.so.1.0: cannot open shared object file: No such fil
```

What changed, Pyenv edition

Back to Pyenv. After finding the floating pin to Pyenvâ€™s master branch, we checked the release notes and recent changes to Pyenvâ€™s invocation of the Python interpreterâ€™s build process. Sure enough, a recent Pyenv change had switched to passing `--enable-shared` to Pythonâ€™s build.

When we checked out Pyenv at a version prior to the troublesome change and built a deployable artifact in a hand-massaged CI environment, that artifact worked in production.

Case closed! We checked in a change that pinned Pyenv to a version before the change to `--enable-shared` was made, and the problem was solved.

â€¦or not.

So itâ€™s fixed â€¦ right?

The problem persisted even after builds ran with our fix. While we now knew the change that initially induced the linker error, actually getting things healthy again required us to answer the other critical question: how did that Pyenv change actually break production?

Put another way: if we changed how we built Python itself such that C extensions `pip installed` in the presence of that Python behaved a different way, why was that a problem? After all, we shipped the same exact version of Python that we built as part of our production-deployable artifacts, so C extension binaries and Python interpreters should have â€œmatchedâ€ (with regards to their assumptions around `--enable-shared` and `libpython`) in all environments.

Two layers of caches interfered with that assumption being true (and the fix we made taking effect); the issue persisted until we addressed each of them.

There are only 2,917 hard problems in software engineering: off-by-one errors, cache invalidation, cache invalidation, cache invalidationâ€¦

The first caching issue was simple: it was a poor decision on our part that resulted in the Pyenv-built Python *actually installed and run in production* being different from the Pyenv-built Python built and run for purposes of C extension building and build/test running. The production-targeted one used a different `pyenv install` invocation (and indeed, a different `git clone pyenv`) than the one that actually compiled and stored a given C extension module.

That was how the issue made it into production in the first place (a Python without `--enable-shared` and thus without `libpython` was installed via the prod-only artifact, but `import`ed extension modules that we had separately built against a Python with the opposite assumption).

The reason for this duality was roughly â€œperformance needs + tech debt.â€ The parts of our CI system that built and tested code used one set of CI scriptsâ€“one tuned for parallelism and performance so folks wouldnâ€™t have to wait ages for deploymentsâ€“and once all of those completed successfully a final artifact was built against *different* CI scripts for shipment to production. Not a great situation to be in, but a fairly common antipattern.

Once we found this, we fixed it. Other than â€œwe fixed itâ€ (by using the same Python interpreter artifact for both CI and production deployments), this doesnâ€™t really bear talking about: it was a classic â€œdonâ€™t repeat yourselfâ€ software engineering mistake, and caused issues for the same reasons that mistake always does.

But that *still* didnâ€™t do the trick. Because caches? Caches are forever.

Specifically, whenever our artifact build code did a `pip install`, it pointed Pip at a common, shared cache directory on each Jenkins agentâ€™s underlying server. In our environment, that directory was shared *across builds;* many different containers (running similar build code, and using this cache) could run simultaneously or sequentially on the same agent. Pip is programmed to download-and-cache or compile-and-cache modules in *some* directory by default; we just changed â€œwhereâ€ to a shared location. A `pip install foobar==1.2.3` in one branch could (assuming â€œfoobarâ€ is a C extension) compile and build that package against a branch-specific Pyenv python that was itself compiled with branch-specific flags and behaviors and then store it in a cache that *other branches, including master/trunk/release branches* could use.

As we saw above, the 5-tuple that Pip uses to disambiguate â€œis this package something that I can install in my environment?â€ (the same 5-tuple that it uses to ask the local cache â€œis this package already compiled/cached into a wheel locally so I can skip the download/compile process?â€) is woefully insufficient to answer the question of â€œwill this wheel *actually work* with my Python?â€

That shared cache, it turns out, is vital to making our builds performant. If each branch had to re-build/re-download everything from scratch, it would add tens of minutes to a branchâ€™s initial build timeâ€“since we depend on hundreds of distributions, dozens of which were compiled C extensions, the extra time spent downloading and/or compiling things would be extreme if they werenâ€™t cached. Thatâ€™s why we used that shared pip cache in the first place.

Once we forcibly cleared that cache, and hunted down a couple of branches that were (for `libpython`-related reasons and a few other ways in which branch-specific code was setting environment variables that affected the validity of the extension binaries that got cached), everything began to rebuild â€¦ very slowly â€¦ but then it worked! Production updates were deployable without causing issues, and developers could resume shipping code.

Importantly, it meant we could also assemble the full â€œColonel Mustard, in the drawing room, with a candlestickâ€ causal chain that led to the issues in the first place.

Whodunit

The cascade that resulted in deployments breaking looked like this:

1. First, Pyenv updated its Python interpreter compile such that Pythons were built with `--enable-shared`.

2. Then, parts of our CI system found empty Docker layer caches over time (as Jenkins agents were replaced or system cleanup scripts ran) and rebuilt Python in `libpython`-all-the-things mode.
3. Our build tried to `pip install` extension modules for which we did not have cached pre-compiled wheels.
4. Pip then fell back to compiling those modules from scratch, which was done against the problematically-built python; this added a linker dependency on `libpython` to those modules binary components.
5. Pip then installed those modules in the build/test environment, where they worked and were usable because the build environmentâ€™s Python interpreter provided `libpython`, *howeverâ€¦*
6. The act of `pip install`ing those linker-poisoned binaries *also* populated a CI-agent-global Pip cache.
7. Extension binaries in that cache were `pip install`ed into our production-deployable artifacts, which were using a much older cached Pyenv-built interpreter that pre-dated the `--enable-shared` change. Importantly, this happened *separately* from our build/test code which made sure things worked in CI. This was the â€œdonâ€™t repeat yourselfâ€ mistake above.
8. As a result, the artifact shipped to production contained a Python built *without* `--enable-shared`, and a bunch of pre-compiled binary extension wheels built against a Python *with* `--enable shared`.
9. When any of those extension wheels were loaded, they tried to link against `libpython`, which wasnâ€™t provided by the interpreter they came with, and so the issue occurred.

```
ENV PYENV_ROOT $HOME/.pyenv
ENV PATH $PYENV_ROOT/shims:$PYENV_ROOT/bin:$PATH

# This pyenv is set here because of 2 incidents with python linker assumptions
# Do careful testing before upgrading
RUN git clone --branch v2.3.4 https://github.com/pyenv/pyenv.git $PYENV_ROOT &&
RUN pyenv install 3.11.1 && rm -rf /tmp/python*
```

A section of one of the problematic Dockerfiles implicated in this issue

What did we learn?

At the end of the day, we fixed some floating pins, paid down some tech debt, busted some caches, and resolved an incident. Not particularly hard problems, all in a dayâ€™s SRE work. But in the process of doing that, we learned a lot of valuable lessons that will inform our future work in this area and are (hopefully) useful for others as well:

1. Floating pins are bad, duh. Reproducible builds are considered a best practice for very good reason. Things like binary-deterministic build systems, while extreme, may have real benefits even for codebases that arenâ€™t the traditional (e.g. C/C++ apps) beneficiaries of strict reproducibility.
2. Floating pins are *everywhere*. Just because you use a lockfile doesnâ€™t mean your build is remotely deterministic. Many build tools encourage the use of floating pins.
3. A lockfile *with a hash of each installable artifact* would have saved us a lot of pain in this case. Locking package versions isnâ€™t sufficient for deterministic dependency installation. Using package-artifact hashes in your lockfiles doesnâ€™t just prevent supply chain attacks; it can also save you from bugs in your local package-compilation systems.
4. If you invoke a package manager in CI, consider running it in binary-artifacts-only mode and putting artifacts in a central store via a side channel. In this case, using `pip install --only-binary ':all:' --no-compile` in all of our CI would have forced Pip to only install from pre-built binaries, and prevented it from compiling

extensions on-demand. In a system with those settings, failures to install a given module would have forced developers to upload and test pre-built artifacts to PyPi (well, more likely our Artifactory mirror thereof), thus catching problems sooner.

5. Package managers can be defective and will happily install broken packages. While Pip, and the Python packaging ecosystem as a whole, are frequent punching bags of the software engineering community, many other languagesâ€™ package managers have similar defects wherein theyâ€™ll install pre-built binary packages in a configuration that cannot possibly work. Package managersâ€™ â€œcache keyingâ€ (the name/version/interpreter/platform/arch 5-tuple in Pipâ€™s case) should be considered best-effort in all cases, and should not be taken to provide assurance of package *usability*.
6. Cache invalidation, as always, is a very hard problem.
7. Write-through caches should be used sparingly in volatile environments, especially not in CI.
8. If your CI system can even *sometimes* compile binaries in untested branch-code environments, youâ€™ve already lost.
9. Developer-convenience-layer tools, like Pyenv, can be liabilities. They may be opinionated, and they may not share your opinions about how things should work. In fact, Pyenv itself is no longer as useful as it once was given the increasing robustness of the Python official build process on MacOS environments.
10. Docker doesnâ€™t save you from classic â€œCI is just a *similar* environment to productionâ€ antipattern. Even with robust tools like containers, it still takes effort and diligence to ensure that the stack your build runs against *is* the stack that you ship to production, not just something made to kinda-sorta resemble the target environment.
11. Improving the performance of a complex CI/CD build system comes at a cost. Thatâ€™s not inherently bad, but itâ€™s something you should know going in. That cost is not just paid in the *complexity* and *maintainability* of your build system; itâ€™s also paid in occasional production downtime and periods of halted deployments.
12. Lastly, my candidate for coldest take of the week: nonuniform cache selection behavior (that is, using a cache in one area of code but not using that cache for the same data in another) is a recipe for subtle bugs.