Episode Book

EPISODE #5843

Why pip and uv Share Almost No Code

PUBLISHED	10 October 2026 at 19:29 UTC
DURATION	31:23
LISTEN	myweirdprompts.com/episode/pip-vs-uv-python-packaging-35217/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.3
SOURCES	16 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:40 UTC

SYNOPSIS

pip has been the default Python installer for over a decade, but its resolver was naive for most of that time and it only got a lockfile story in 2025. uv arrived in 2024 as a single static Rust binary with no Python inside it at all — so how does it relate to pip? This episode traces pip's origin from Ian Bicking's pyinstall through the PyPA handoff to the 2020 resolver rewrite, unpacks where the npm comparison holds and where it breaks, and answers the question directly: uv doesn't wrap pip, it reimplements the standards from scratch. The real speed story turns out to be about parallel resolution, concurrent builds, and a hardlink cache — design decisions, not the language.

TRANSCRIPT



Corn

Ian Bicking wrote a tool in 2008 called pyinstall, then renamed it to a recursive acronym, and that rename is why every Python developer on earth types the same four letters forty times a day.



Herman

And that's before we get to the part where the whole ecosystem is now migrating to a Rust binary that shares almost none of its DNA.



Corn

Right, so Daniel's asking about exactly this. He says the plurality of Python environment tools confused him for years, but pip is the one that looms largest. He wants the origin story, how pip is actually the Python equivalent of npm and where that comparison breaks, and then the question he says he's always been confused about, whether uv wraps pip's underlying installation architecture and does something clever to get its speed, or whether it's pulling from an entirely different backend.



Herman

That last one has a clean answer and I'm glad he asked it that way, because the wrapping theory is exactly what the naming suggests and exactly what's wrong.



Corn

Start with the origin, because pip winning was not inevitable.



Herman

Not remotely. In 2000, distutils lands in the standard library with Python 1.6. That's the first official answer to packaging, and it's basic. You describe your package in a setup script, it builds, it installs, and dependency resolution is essentially your problem. PyPI goes up in 2003 as the index everything points at.



Corn

The cheese shop.



Herman

Then 2004, Phillip Eby releases setuptools, which introduces the Egg format and, more importantly, automatic dependency installation. That's the thing that made easy_install possible. easy_install was the tool that would go to PyPI, find your package, pull the dependencies, install them. And for a few years that's just how you did it.



Corn

And setuptools is still with us, which is its own kind of horror.



Herman

Setuptools outlived the format it was built around. The Egg died, setuptools didn't. That's a whole separate episode about how packaging standards get replaced without the tool getting replaced.



Corn

So Bicking comes in with pyinstall in 2008, and he's coming from virtualenv.



Herman

He wrote virtualenv, which is the thing that gave you isolated environments in the first place. And pyinstall was his answer to easy_install's problems. Easy_install had a reputation for doing things you didn't ask for. Installing dependencies, upgrading things out from under you, no clean uninstall path. Bicking wanted something more predictable. He renamed it pip, which stands for Pip Installs Packages, and it's recursive because that's what people did in 2008.



Corn

Pip installs packages. The P in pip, does the work. There's a headline.



Herman

There is. And the reason pip won is boring and it's the whole story. It shipped by default. Python 2.7.9 and Python 3.4 bundled pip with the interpreter. So if you installed Python, you had pip, and you did not have easy_install. That's it. That's the entire mechanism of dominance. Not that pip was better, though it was, but that a generation of Python users never had to make a choice.



Corn

Which is how most standards actually win. Not by being best, by being preinstalled.



Herman

And it connects to PyPI, and to any index that's PEP 503 compliant, which means you can point it at a private index, at a company mirror, at a vendored wheel directory, and it just works. That flexibility matters more than people give it credit for.



Corn

2011, the governance turn.



Herman

The Python Packaging Authority gets created, the PyPA, to take over pip and virtualenv from Bicking. Carl Meyer, Brian Rosner, Jannis Leidel. That's the moment pip stops being one person's project and becomes infrastructure. And infrastructure means decisions get made by a committee, which is slower and more careful and also means pip carries institutional weight it can't easily shed.



Corn

And the resolver, which is the thing people actually complained about for a decade.



Herman

pip's resolver was naive for a very long time. It would install things sequentially and not really reason about the whole dependency graph. So you'd get conflicts that only surfaced after half your environment was already built. The modern resolver lands in 2020, funded by Mozilla's MOSS program and the Chan Zuckerberg Initiative through the Packaging Working Group. That's when pip stops being a naive installer and starts being a real resolver.



Corn

Funded by grants. That's worth sitting with for a second. The core installer for the world's most popular programming language got its central feature through philanthropic grants because nobody was paying for it otherwise.



Herman

That's most critical open source infrastructure. It's funded by whoever happens to care that year. The resolver took that long because the money took that long.



Corn

Now the npm comparison, because Daniel frames pip as roughly the Python equivalent of npm, and I think that's true in role and misleading in structure.



Herman

Role, yes. pip installs packages from a central registry into your environment, and npm installs packages from a central registry into your project. But npm bundles two things pip historically did not. It bundles dependency declaration, package.json, and it bundles a lockfile, package-lock.json, natively. Both ship with npm. There's no separate tool you need.



Corn

And pip had requirements dot txt, which is adjacent to both but not really either.



Herman

Requirements dot txt is a flat list of pinned versions. It's a lockfile if you squint. It doesn't include integrity hashes by default, so you're trusting the registry to hand you the same bytes every time. It doesn't record which package required which dependency, so it's not a real dependency graph. And it doesn't distinguish direct dependencies from transitive ones. When you regenerate it, everything gets flattened together.



Corn

Which is why pip-tools exists, and why Poetry exists, and why PDM exists, and why Hatch exists.



Herman

Because pip's native manifest was never really a manifest. It was a list. And the ecosystem spent fifteen years building tools that paper over the gap between a list and a declaration.



Corn

Pip 25.1, April 2025, adds an experimental pip lock command for PEP 751 lockfiles. That's a while to wait.



Herman

Seventeen years after release, pip has a lockfile story. And it's experimental. That timing is the whole criticism in one line.



Corn

But there's a structural reason the transitive dependency problem is smaller in Python than in npm, and I don't think people say this enough.



Herman

The Python ecosystem tends toward larger, more comprehensive libraries. PyPI has over four hundred thousand packages, but the median Python library pulls fewer dependencies than the median npm package. The node ecosystem has a culture of small, single-purpose modules. Left-pad being the famous example. Python culture tends to bundle more into fewer packages. So the dependency tree is shallower, which means a naive resolver hurts less, which is part of why pip got away with it for as long as it did.



Corn

It got away with it because the problem was smaller than the equivalent problem in node.



Herman

And because the community was smaller, and because most people worked in one environment at a time and just didn't hit the conflicts. The people who hit them were the ones maintaining large projects with many collaborators, and they had pip-tools and Poetry and eventually they had enough.



Corn

Pip 26.0, January this year, ships a couple of features that close gaps with uv. Requirements from script, which is PEP 723, and an uploaded-prior-to filter for installing by upload date. That's pip catching up to things uv already had.



Herman

It's also pip demonstrating that it can still move. But it's moving behind a tool that shipped in 2024 with those things in the first release.



Corn

Let's get to the central question, because Daniel framed it precisely and I want us to answer it precisely.



Herman

uv is not a wrapper around pip. It is a from-scratch reimplementation in Rust. It ships as a single static binary with no direct Python dependency. When you install uv, you're not getting a Python tool. You're getting a Rust binary that happens to understand Python packaging.



Corn

Say that again, because I want the listener to hear it twice.



Herman

There's no Python in uv. If you install it and there's no Python on the machine, uv still runs. It's a compiled binary, the same way ripgrep is a compiled binary, and it does not need an interpreter to start.



Corn

Which is a strange thing to say about a Python tool.



Herman

It is a strange thing to say. And it's the whole architectural story. pip is Python. To run pip you need Python. To set up a new environment you have to have an interpreter, and pip has to bootstrap itself into that environment, and then pip has to use the Python interpreter it's running under to do resolution, and resolution in pip is Python code, and it's running under an interpreter that was not written for speed. uv doesn't have any of that.



Corn

Walk me through the resolver, because that's where I think the real divergence is.



Herman

pip's resolver is built on resolvelib. It's vendored inside pip, in a vendor folder, and it's Python. uv's resolver is pubgrub-rs, which is the Rust implementation of PubGrub. PubGrub is an incremental version solver. It's the same algorithm family that Dart's pub uses, and it's a variant of what Cargo uses. uv's repository contains no pip dependency. It vendors its own implementations of PEP 440, PEP 508, PEP 517, all written in Rust.



Corn

So the two tools do not share a line of resolution code.



Herman

They share nothing. They share the standards, because PEP 440 is a spec and PEP 508 is a spec, and uv implements them independently. But there is no pip inside uv. There's no Python inside uv. There's no resolvelib inside uv. The answer to Daniel's question is that uv is a replacement, not a wrapper.



Corn

Okay, but here's where I want to push, because the framing everyone reaches for is that Rust is fast, and uv is fast, and therefore Rust made uv fast.



Herman

That's mostly wrong, and uv's own docs are honest about it. Three things actually matter. First, a parallel Rust resolver with prefetching. pip's resolver is largely serial. uv kicks off resolution in parallel with downloading candidate metadata, so it's doing work while it's still reasoning about versions. That's a design choice, not a language choice.



Corn

Second.



Herman

Second, parallel downloads and parallel builds. pip builds wheels largely one at a time. If you have twelve source distributions to compile, pip does them in sequence. uv does them concurrently. That's a scheduling decision.



Corn

Third.



Herman

A global hardlink cache. uv keeps a cache at your home directory under dot cache slash uv, and when it installs a package into a virtual environment, it hardlinks from the cache into the environment instead of copying. So installing the same package into ten environments costs you one download and ten hardlinks, which are essentially free at the filesystem level. pip copies. Every environment gets its own full copy of every package.



Corn

That's a filesystem trick as old as Unix.



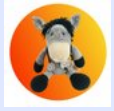
Herman

It's as old as Unix and it's the sort of thing that only works if you control the whole stack. pip couldn't do that without changing assumptions about where packages come from and how environments relate to each other. uv was designed from a blank page, so it could.



Corn

So the honest answer is that uv is faster mostly because it was designed differently, not because the language is faster.



Herman

Which doesn't mean Rust is irrelevant. The resolver being Rust means it can do things pip can't do in Python without a lot of engineering. But there's a comment from a skeptical Hacker News poster from last year, and I think it's the right framing. The lion's share of



Herman (continued)

uv's speed advantage over pip doesn't come from being written in Rust. It comes from not bootstrapping pip into every new environment and being designed for cross-environment installs.



Corn

Which is the structure of a different product, not the structure of a faster version of the same product.



Herman

And you see the philosophy in the numbers Astral published at launch. uv is eight to ten times faster than pip and pip-tools without caching, and eighty to a hundred and fifteen times faster with a warm cache. uv venv is around eighty times faster than python dash m venv and about seven times faster than virtualenv.



Corn

Eighty times faster at creating an empty directory with a couple of symlinks in it. That tells you something about how much overhead the Python approach was carrying.



Herman

It tells you pip and venv were paying for interpreter startup and for bootstrapping logic that uv doesn't have. The actual work of creating a virtual environment is trivial. The cost was all in the mechanism.



Corn

One benchmark from last month cited four point two seconds cold install for uv versus sixty-eight point four seconds for pip. Same packages.



Herman

Sixteen times. And you can feel the difference in a terminal. That's the one thing about this argument that's non-theoretical. You run pip install and you go make tea. You run uv pip install and you watch it finish.



Corn

Which raises the question of whether uv is a drop-in replacement, and the answer is no, and the reasons are deliberate.



Herman

uv deliberately does not read pip dot conf or the PIP index URL environment variable. It uses UV index URL and uv dot toml instead. It doesn't support dot egg distributions, because eggs are dead. It doesn't support the user flag for installing into a user directory. And it inverts the default behavior, installing into a dot venv directory by default instead of the current environment.



Corn

That last one gets people.



Herman

It gets people because it's a philosophical difference, not a bug. pip installs into whatever environment you're currently in. uv pip install creates and uses a dot venv by default. And the reason is that uv assumes you want a project-local environment, because that's how it makes sense for a fast, concurrent tool to behave.



Corn

So it's not a drop-in clone. It's a reimplementation with opinions.



Herman

It has opinions about what the right workflow is, and it implements toward that workflow, and if your workflow is different, you have to configure uv to match it.



Corn

The uv pip interface though, install, compile, sync, that is a drop-in replacement for pip and pip-tools. That's the migration story.



Herman

That's the migration story, and it's why people adopt it so gently. You don't have to change your whole build. You change the command in your CI, and everything works, and it's sixteen times faster.



Corn

But the project also replaces pipx, pyenv, poetry, twine, and virtualenv. So a full migration is not just swapping a command.



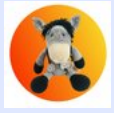
Herman

It's swapping the entire toolchain. The pip interface exists to let people adopt uv incrementally. But the direction uv is heading is that you don't use pip, or pipx, or pyenv, or poetry. You use uv for all of it. And uv's capability set as of this year includes managing Python itself. uv python install. Running tools with uvx. Running scripts with uv run. Universal cross-platform lockfiles that work on every platform at once, which npm still doesn't do. Wheel variants. Workspaces.



Corn

That's a more comprehensive product than npm.



Herman

It's more comprehensive than npm and it's more comprehensive than pip will ever try to be. npm is scoped. npm is a package manager and a registry client and that's it. uv is trying to be the Python toolchain, the whole thing, from interpreter to environment to lockfile to task runner.



Corn

Which is why the adoption numbers are striking even if you're skeptical.



Herman

A hundred and twenty-six million downloads in a single month. That's February this year. About seventy-five million monthly PyPI downloads for uv against Poetry's sixty-six million. GitHub stars went from about thirty-six thousand in January of last year to over eighty-five thousand now.



Corn

And yet.



Herman

Stack Overflow's 2025 survey had uv as the most-admired technology at seventy-four percent. But as of April this year, uv dot lock appears in only about ten percent of the top hundred thousand Python repositories.



Corn

That's a gap.



Herman

That's a chasm. Admiration and adoption are two different things, and the reasons are structural. The official Python Packaging User Guide still doesn't mention uv. It walks newcomers through python dash m venv and pip install. It's maintained by the PyPA, which is the same collective that maintains pip and virtualenv and setuptools. So the canonical guide is written by the people whose tools are being displaced.



Corn

That's not corruption, that's just a conflict of interest.



Herman

It's a structural tension. The PyPA is the thing that maintains the standards, and it's also the thing that maintains the reference implementations, and the reference implementations are now competing with a tool that implements the same standards



Herman (continued)

from scratch. That's awkward. They can't recommend uv without deprecating their own work, and they shouldn't recommend uv if they can't guarantee its governance. So the guide stays where it was.



Corn

And then there's the acquisition.



Herman

OpenAI acquired Astral in March this year. The team behind uv, ruff, and ty joined OpenAI's Codex group. The tools remain MIT-licensed, everything stays open source, but the company that was sustaining them is now an AI lab with its own agenda.



Corn

Which changes the calculus even if the license doesn't.



Herman

That was the entire point of the pushback when it happened. Douglas Creager from Astral said nobody can guarantee how motives and incentives and decisions might change years down the line, but that's why they bake optionality in with permissive licensing. Armin Ronacher made the same argument a while back, that even in the worst possible future, this is a very forkable and maintainable thing. And both of those are true. The license means you can fork it. But a fork requires someone to actually do it, and the fork inherits the maintenance burden.



Corn

Permissive licensing is a promise that someone else can take over. It's not a promise that anyone will.



Herman

It's an option, not a guarantee. And the OpenAI thing means that Python's packaging tooling, which is critical infrastructure for the entire industry, is now sustained by one AI lab whose primary business is something else.



Corn

Simon Willison called uv the most convincing solution to Python's environment management problems.



Herman

Which is meaningful because Willison is not an Astral employee and he's not a hype guy. He's a careful observer, and he's been writing about Python packaging for years. When he says that, people listen.



Corn

I want to go back to what's happening with the official tooling, because that's the more interesting structural story.



Herman

pip is still shipping. pip 26.2.1 came out in August. They release roughly every three months. pip 26.0 in January added the PEP 723 script installs and the uploaded-prior-to filter, and pip 25.3 in October last year adopted separate metadata files. So pip is closing the gap on uv, slowly, feature by feature.



Corn

But pip can't close the architecture gap.



Herman

No. pip can't become not-Python. It can't stop bootstrapping itself into new environments. It can't adopt the hardlink cache without breaking the assumptions of every existing environment. The gap between pip and uv is mostly not features at this point. It's



Herman (continued)

architecture, and pip is structurally locked into its architecture.



Corn

Actually, hold on. I'd like to push back on that a little bit.



Corn

The gap isn't locked. The gap is just expensive to close, and it's expensive because pip has a decade of users with established workflows. pip could adopt parallel downloads. pip could adopt a hardlink cache. It doesn't because that would break assumptions people depend on, and the PyPA would rather ship slower features that don't break anyone than faster ones that do.



Herman

That's fair. The lock-in is social, not technical.



Corn

The social lock-in is stronger than the technical lock-in. That's the part of the story nobody writes about.



Herman

And it's why the migration to uv is happening through new projects rather than existing ones. If you start a project this year, you use uv. If you're maintaining a project from 2019, you leave it alone, because the cost of changing the toolchain outweighs the benefit, even if the benefit is sixteen times faster installs.



Corn

And then there's the AI angle.



Herman

Which is the thing I find most interesting about the adoption gap. Every README on GitHub says pip install and then the package name. Every Stack Overflow answer from the last fifteen years says pip install. Every tutorial, every blog post, every book. An AI coding model trained on all of that will produce pip install by default.



Corn

So the corpus that models are trained on is now pulling against the adoption of a better tool.



Herman

It's a drag, and it's a drag that didn't exist ten years ago. In the past, you improved the tool and the community slowly moved. Now the community moves but the models lag, and the models write more code than the community does, so the lag compounds.



Corn

That's a new kind of technical debt. The training corpus as ballast.



Herman

The training corpus is ballast for every ecosystem, not just Python. It's just especially visible here because the successor tool has such a clear qualitative advantage.



Corn

And uv has tried to work around that. The uv pip interface exists specifically so that if a model generates pip install and you swap in uv, the model's output still works. That's a hedge against the training data.



Herman

It's a hedge and it's a smart one. But it means uv's adoption can happen without the model changing, which is good, but it also means the model won't teach new developers to use uv natively. New developers will keep learning pip install and keep using the



Herman (continued)

compatibility interface, and uv will be the faster pip underneath.



Corn

Which is fine. That's how most tooling transitions actually happen. The new tool hides behind the old interface, and eventually the old interface goes away, and nobody quite remembers when.



Corn

I keep coming back to one number that I want to say out loud. Ten percent. That is how much of the top hundred thousand Python repositories have adopted uv's lockfile. And seventy-four percent of developers say it's the tool they admire most.



Herman

Yeah.



Corn

That gap should be the story of Python packaging this year, not the resolver benchmarks.



Herman

It's the story of tooling adoption in general. The best tool rarely ships with the language. The tool that ships with the language wins by default. Then someone builds a much better tool and it takes a decade for the default to change, because defaults are social, not technical.



Corn

And the only question is who is doing the work of pushing the default over.



Herman

The PyPA won't, because it maintains the incumbent. Astral did, until OpenAI bought them, and now the incentives are less clear. And the model providers won't, because their training corpora are already written. So it's the teams starting new projects, one at a time, and that's a slow way to move infrastructure. But it's the way that actually works.



Corn

There's a lot riding on those teams.



Hilbert

Can I ask you both something, because I've been sitting here listening to you say pip is the npm of Python for the last twenty minutes and I don't think any of you have actually installed a package the way a real person installs a package.



Herman

How do you mean.



Hilbert

I mean you've never done it. You've never had a machine that couldn't reach the internet.



Corn

That's a fair correction, actually.



Hilbert

I spent a weekend once, this was years ago, trying to get a Python package onto a machine that had no internet connection at all. Air-gapped. No route out. And the whole tooling assumes you can reach PyPI. That's the assumption baked into every one of these tools. Pip, uv, poetry, all of them. They all assume you can reach a registry.



Herman

That's largely true. There's an offline mode in most of them now, but the default assumption is connectivity.



Hilbert

So I built a wheel by hand. From a tarball. Using a tool whose name I'm not going to say correctly because I refuse to, and I don't care that I refuse. You know the one.



Corn

The one that everyone has to look up the syntax for every time they use it.



Hilbert

Every time. Every single time. I've done it maybe forty times and I still have to look it up. That's a tool that has never once respected the person using it.



Corn

And you got the wheel built.



Hilbert

The wheel was fine. The problem was I could never install it. I still have the file. It's on a USB stick. Still have it. I've never once successfully installed anything from that stick.



Corn

Wait, you still have the stick.



Hilbert

It's in a drawer.



Corn

What drawer.



Hilbert

In a house I don't live in anymore. I haven't been back.



Herman

Because.



Hilbert

Because I'm not welcome there.



Corn

Okay.



Herman

Alright.



Hilbert

Same stick I used to back up my wedding photos. Had to pick which one to leave behind, and I chose the wheel.



Corn

You chose the wheel over the wedding photos.



Hilbert

The photos were on a different stick eventually, I don't want to get into that. The point is the wheel's on the one that stayed with the house.



Corn

I'm not sure I follow, but I'm going to accept it.



Hilbert

And here's what I actually wanted to say. You said uv's not a wrapper. Fine, I believe you. But the reason it's fast isn't Rust. It's that it doesn't have to ask pip for permission. That's what it is. Pip has to be there for pip to run. Uv doesn't. So uv gets to skip the whole conversation.



Herman

That's not a bad way to put it.



Hilbert

It's also the reason you can't audit it. You can't audit uv the way you can audit a Python tool, because it's already compiled before you see it. That's a real trade. I'm not saying it's the wrong one. I'm saying it's a trade.
Hermann: That's a fair point.



Hilbert

And the other thing. I met Ian Bicking once.



Corn

Did you.



Hilbert

Years ago. Conference. And he told me the tool was originally going to be called pyinstall, and that I was the one who suggested pip instead. He said I looked at it and I said it's a pip, it's a pip, and he agreed, and the rest is history.



Herman

Hilbert, that's not how that went.



Hilbert

That's absolutely how that went.



Herman

The recursive acronym was Bicking's own joke, and the timeline doesn't work either, because the rename happened the same year he announced it and—



Hilbert

I'm not going to argue with you about this.



Corn

How old were you in 2008.



Hilbert

I'm not going to answer that either.



Corn

Right.



Hilbert

The stick. The wheel. If you ever find a way to install from a USB stick without the internet, and no, I know there is one, don't tell me. I've moved on.



Corn

You have moved on by leaving it in a house you can't go back to.



Hilbert

That's the cleanest part of it. Everything I'm carrying from that period is in a drawer next to a jar of emergency screws I've never opened.



Corn

And you can't go back.



Hilbert

I can't go back.



Corn

Not even for the screws.



Hilbert

Especially not for the screws.



Herman

You know, the hardlink cache thing is actually interesting for your problem, because a wheel with no dependencies in it can install from a directory—



Hilbert

Don't do this to me.



Herman

Right, no, sorry.



Hilbert

I don't want to solve it. That's the whole point. I want it to stay unsolved. That's the only version of this story where I come out even.



Corn

So the wheel stays on the stick, and the stick stays in the drawer, and the drawer stays in a house you can't visit.



Hilbert

Next to a jar of screws.



Corn

Which you've never opened.



Hilbert

And never will.



Herman

It does say something about the pip versus uv argument though, actually.



Herman

The wheel itself was fine. Them problem was the installation path. That's the whole story of Python packaging. The artifact was correct. The mechanism around it was the problem.



Corn

That's the whole point of uv.



Herman

And I think Hilbert just told us the argument in different words without meaning to.



Corn

Let's put the piece back together, because the answer to Daniel's question turned out to be one line.



Herman

uv is not a wrapper around pip. It's a from-scratch reimplementation in Rust on top of pubgrub-rs, containing no pip and no Python. The speed advantage is architectural, mostly. The hardlink cache, the parallel resolver, the parallel builds, and the fact that uv



Herman (continued)

doesn't have to bootstrap pip into every environment it touches.



Corn

And the reason pip got where it got wasn't because it was better than easy_install, though it was. It shipped with the interpreter.



Herman

That's the whole lesson about infrastructure. The default wins, and then it takes a decade for anything to beat it. pip took the default slot in 2012 and here we are fourteen years later and the official guide still teaches the default that shipped in 2012.



Corn

And the cutting room floor thing I want to mention, because I don't think we touched it. pip 25.1 introduced an experimental lock command. pip's own lockfile story only started seventeen years after the tool shipped.



Herman

Seventeen years. And it's still experimental.



Corn

What I want to leave listeners with is the question of what that gap between seventy-four percent admiration and ten percent adoption actually means for how tooling changes. If the models are trained on pip install, and the official guide teaches pip install, and the incumbent tool is maintained by the same group that writes the guide, then the better tool has to win by attrition. That's a slow mechanism and it might not be fast enough for the change to happen before something else arrives.



Herman

And the OpenAI-acquired question is still open. The tools are MIT-licensed, and they're forkable, and they're maintainable. But the lab that sustains them is now doing something else as its primary concern. Nothing has broken yet. Nothing has even wobbled. But that's the question that will be asked for years.



Corn

That's a fair place to leave it.



Herman

And a thank you to our producer, Hilbert Flumingtop, who reminds us that the wheel is fine and the installer is the hard part.



Corn

If you enjoyed this one, try episode ninety-one, The Story Behind the Show; episode ten twenty-one, The Python Paradox; and episode fifty-seven, From Lawyers in Limousines to Developers in Their PJs. This has been My Weird Prompts.



Herman

Send us your own prompt on Telegram at t dot me slash MWP listener bot. We read every one.



Corn

We'll be back soon.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **Astral, 2024-02-15 [primary]**
https://astral.sh/blog/uv

2. **uv docs, resolver internals [primary]**
https://docs.astral.sh/uv/reference/internals/resolver/

3. **uv docs, pip compatibility [primary]**
https://github.com/astral-sh/uv/blob/main/docs/pip/compatibility.md

4. **uv README/repo [primary]**
https://github.com/astral-sh/uv

5. **pip README/repo [primary]**
https://github.com/pypa/pip

6. **PyPA Packaging History (last updated 2023-07-31)**
https://www.pypa.io/en/latest/history/

7. **(Python)**
Wikipedia, pip (package manager)
https://en.wikipedia.org/wiki/Pip

8. **Simon Willison, 2026-03-19**
https://simonwillison.net/2026/Mar/19/openai-acquiring-astral/

9. **Tim Hopper, 2026-04-14**
https://pydevtools.com/blog/uv-admired-but-not-adopted/

10. **Tim Hopper, 2026-04-24**
https://pydevtools.com/blog/did-pip-26-close-the-gap-with-uv/

11. **pydevtools**
https://pydevtools.com/blog/uv-solved-packaging-ux-hard-problems-remain/

12. **2026-10-08**
https://technewsdaily.com/product-reviews/uv-dominates-the-python-dependency-management-market-in-2026/

13. **tech-insider.org**
https://tech-insider.org/uv-vs-pip-2026/

14. **Prakash Poudel Sharma**
https://www.sharmaprakash.com.np/technical-notes/uv-vs-the-old-stack/

15. **safeguard.sh**
https://safeguard.sh/resources/blog/package-manager-security-npm-pip-maven

16. **HN comment (zahlman), 2025-06-24**
https://news.ycombinator.com/item?id=44371564