Episode Book

EPISODE #6598

Why Your Python Command Is Probably a Rust Binary

PUBLISHED	14 November 2026 at 15:15 UTC
DURATION	27:21
LISTEN	myweirdprompts.com/episode/cli-languages-and-packaging-14477/
HOSTS	Corn, Herman, Clive, Hilbert
SCRIPT MODEL	z-ai/glm-5.3
VOICES	omnivoice
PIPELINE	V5.26
SOURCES	25 cited by the research (listed at the end)
BOOK GENERATED	15 November 2026 at 11:16 UTC

SYNOPSIS

Installing a command-line tool like ruff or uv through pip looks like installing Python, yet it actually drops a compiled Rust binary onto your disk. Every CLI hides three layers: its implementation language, its artifact format, and its install path. This breakdown explores why Go dominates infrastructure tools like Docker and kubectl, why Rust is replacing C++ in tools like the fish shell, how Python entry-point wrappers work, and why distribution headaches—not raw speed—drive most language choices.

TRANSCRIPT



Corn

One word. That's all a command is. You type ruff, you hit enter, and behind that four-letter word sits a compiler toolchain, a package resolver, sometimes an entire interpreter that got smuggled onto your disk.



Herman

And nobody thinks about it, because the whole point of a command line is that it answers back instantly and doesn't explain itself.



Corn

Daniel's prompt this week picks at exactly that. He points out that we install CLIs from package managers, we type them, and unless you're the person building one, it's a black box. He wants to know what languages these things are actually written in now, how they get compiled and distributed, and how much complexity a simple-looking command can hide. And he's specific about where he wants us to look — the heavy hitters in Python tooling and Linux sysadmin.



Herman

Which is the right place to look, because those two worlds have diverged in opposite directions over the last five years.



Corn

Before we get into it, one grounding example, because I think everyone has had this moment. You run which on a command you've used for a year, and the path is somewhere you didn't expect — some dot-folder in your home directory, or a virtualenv you forgot existed, or a pyenv shim that's been silently intercepting every Python command you've typed for months. That little shock of "oh, that's where you live" — that's the whole topic in miniature.



Herman

The package manager is the wrapper. The CLI is the payload. Let's open the box.



Herman

Here's the inversion that anchors the whole thing. The thing you install is almost never the thing you run. Pip install ruff — that looks like installing a Python program, right? Pip is a Python package manager. It installs a Rust binary. Pip install uv installs a compiled Rust executable of about fifty-three megabytes, plus a stub Python module whose only actual job is to tell you where the executable lives. That's it. That's the Python in the package.



Corn

Fifty-three megabytes, and the Python contribution is a function that points at it.



Herman

A very well-organized pointer. And this pattern repeats everywhere, because every CLI sits on three layers, and all three are invisible from the command line. There's the language it's written in, there's the artifact format it ships as — wheel, static binary, bundled interpreter, distro package — and there's the install path that delivers it. The command name tells you none of that.



Corn

So the same command can arrive as completely different objects depending on how you asked for it.



Herman

Ruff comes by pip, by pipx, by uv tool install, by Homebrew, by Conda, by a curl-pipe-sh script, by a PowerShell one-liner. That's five or six different artifact types behind one four-letter command. Same version string, completely different files on disk.



Corn

Four letters, six disguises.



Herman

And the version string is the part that fools people, because it's the one piece of information every install path agrees to print. You run ruff dash dash version on two machines, you get the same answer, and you conclude you have the same thing. You might have a Homebrew pour on one and a pip wheel on the other, built by different people, against different libraries, with different update schedules. The version string is a checksum of nothing.



Corn

That's a fun-house mirror of reproducibility — everything that's checkable is identical, and everything that matters is invisible.



Herman

So the two richest threads are Python tooling, where the command is usually a generated wrapper around something compiled, and Linux sysadmin tooling, where Go and Rust have basically displaced Python and C. Start with languages, because that's where the pattern is cleanest.



Corn

Go owns infrastructure. That's the part I want you to defend, because the usual explanation is wrong.



Herman

The usual explanation is performance, and it's mostly not performance. Docker, kubectl, Terraform, Hugo, the GitHub CLI, k9s, lazygit — that whole tier is Go. One writer put it bluntly: the Python distribution story was so painful it became the primary reason developers chose Go. Not performance, not type safety, not goroutines. Distribution. A Go CLI compiles to a single static binary. No runtime, no virtualenv, no "which Python are you running," no pip install dash dash user versus pipx debate. Cross-compiling is two environment variables — you set the target operating and architecture, you build, and your entire release process fits in a ten-line Makefile target.



Corn

But how does that work in practice? You say two environment variables — walk me through what a release actually looks like.



Herman

You set GOOS to the target operating system and GOARCH to the target architecture, you run go build, and out comes a binary for that pair. Want to release for Linux, macOS, and Windows, on two architectures each? That's six invocations of the same command. No container images, no per-platform build machines, no matrix of CI runners each with their own quirks. The release process for a small Go CLI can be a for-loop in a shell script, and for a lot of maintainers, it is.



Corn

The junior engineer story is the one that sticks. Somebody documented a fresh laptop, one internal Python CLI, and forty-five minutes of virtualenv conflicts.



Herman

Forty-five minutes, and the Go equivalent is download a file, chmod it, done. There's also the startup tax people forget — one deploy script called a Python CLI twelve times across four environments, and interpreter startup alone burned about fourteen seconds. A Go tool that did the same job cut the pipeline by almost twenty seconds doing nothing clever.



Corn

Twenty seconds of nothing happening, proudly.



Herman

And the twelve-calls detail is worth sitting on, because it's the shape of how CLIs actually get used. Nobody writes a deploy script that calls a tool once. You call it per environment, per service, per stage — and a startup cost that's invisible once becomes a tax you pay a dozen times in a row. A quarter-second of interpreter boot is nothing. Twelve quarters of interpreter boot plus imports is a coffee break.



Corn

Now, Rust owns the new generation of developer tooling. The Astral stack is the obvious example — uv and ruff, both written in Rust, both monstrous on GitHub. But the wave is wider than Astral. The fish shell moved from C++ to Rust. asciinema went through its whole journey. CRuby's YJIT compiler got ported from C to Rust. Meta's pyrefly type checker replaced the OCaml pyre. ripgrep is the canonical Rust CLI and has been since twenty-sixteen.



Corn

The fish maintainers' reasoning is the honest one, isn't it.



Herman

It's refreshingly unglamorous. Nobody really likes C++ or CMake, and there's no clear path for getting off old toolchains. Every year the pain will get worse. That's not "Rust is beautiful." That's "the roof is leaking and this is the ladder we have."



Corn

And I think that distinction matters more than people admit, because the two stories get blended. There's the aesthetic story — memory safety, fearless concurrency, the whole brochure — and then there's the maintenance story, which is just "the old toolchain is dying and something has to replace it." Fish took the second road and said so out loud. Most projects take it and don't.



Herman

Meanwhile Python still holds one trench.



Herman

Data work. Anything that transforms CSVs, queries databases, generates reports, runs ML inference — Python owns it, because you are not reimplementing pandas and polars and scikit-learn and sqlalchemy in Go. Somebody tried. A hundred-and-twenty-line Python data-analysis CLI, port to Go, gave up on the library gap alone. The batteries-included promise broke on distribution, but the batteries themselves are still unmatched.



Corn

And TypeScript?



Herman

The other big camp, and it's retreating. OpenAI's Codex CLI originally required Node twenty-two or newer, and when they went native they said it outright — zero-dependency install, currently Node v22+ is required, which is frustrating or a blocker for some users. They shipped the Rust version and retired the TypeScript one within a week.



Corn

The blocker was the installation story, again. It keeps coming back to that.



Herman

Which is why the asciinema story is the one that breaks the clean narrative. It went Python, then Go, then back to Python, then finally Rust. The Go rewrite got reverted. And the author's reasons were pure distribution politics — no need for concurrency or high speed here, Go's lack of versioned packages and a central repository made packaging cumbersome for distro maintainers, and, quote, if err not nil gets old even faster. The tool had zero external dependencies in Python, so the language that's supposed to be the easy-ship language was the hard-ship language for his case.



Corn

He ran the experiment everybody else skips and published the retraction.



Herman

Language choice for a CLI is usually about who has to receive it, not who has to write it. Distro maintainers, Windows users, Alpine users, people with no compiler — the language is chosen for them.



Corn

I'd have saved us ten minutes if I'd said that.



Herman

You would have, and then we'd have missed the Go retraction. So we know what languages they're written in. The next question is how a command actually gets made — and this is the part I find delightful, because in Python the command is usually a generated file.



Corn

Not the program.



Herman

Here's the mechanism. A package declares its commands in the metadata — name equals package colon function, pointing at some no-argument entry function. When the wheel is built, that lands in an entry-points file inside the dist-info directory. When pip installs it, pip reads that file and writes an executable script into the environment's bin directory. A small Python file, generated from a template, that imports your module and calls the function, with exit code wired through. That file is the command. It's a generated wrapper, maybe ten lines, and the actual program is somewhere behind it.



Corn

So when I cat the command in my bin directory, I'm not looking at the tool. I'm looking at pip's handwriting.



Herman

Pip's handwriting, on a template maintained inside the packaging libraries, which means the file sitting in your bin directory can literally change because you upgraded pip — same tool, same version, different wrapper. Nobody's tool changelog mentions it, because it's not in the tool.



Corn

I typed a command and got a bookmark.



Herman

On Windows it's better — pip doesn't write a script, it embeds a launcher executable. The distlib code that ships inside pip literally carries a table of prebuilt launcher binaries — w.exe, w_arm.exe, w64-arm.exe — and it picks the right one for your machine and glues your entry point into it. There's shebang handling in there too, for the Unix side, with a fallback to a contrived dual-purpose shebang through bin slash sh if the interpreter path is too long or has spaces in it.



Corn

Somebody has thought very hard about spaces in paths.



Herman

Decades of scar tissue, all of it hidden. Then the next layer: wheels versus source distributions. Pip installs a prebuilt wheel when a compatible one exists, and otherwise downloads an sdist and compiles it on your machine. What decides "compatible" is the wheel tag, and on Linux that's largely about glibc. Binaries built against an older glibc run on newer systems. The reverse doesn't work. So wheels are built inside deliberately ancient Docker images — the manylinux images — so the binary asks as little as possible of your libc.

C

Clive

One-Star Getaways. In Kraków I stayed at the Pod Wawelem Vista Guesthouse, where the room advertised a river view and delivered a framed photograph of the river, hung at a slight angle, over a radiator that clanked like a man counting to ten. One star.

C

Clive

For anyone looking for a disappointing trip this winter, I recommend it. Tell them Clive sent you. They will not know who that is, which is part of the experience.



Corn

The binary is aged like wine, on purpose.



Herman

PEP 513 even warns you off the obvious fix: statically linking glibc is usually a bad idea, because things like dlopen can't be called from code that statically links it. So a runtime dependency on the system glibc is unavoidable, and the whole tagging machinery exists to manage it. And Alpine users, with musl instead of glibc? manylinux makes no attempt at musl compatibility, so Alpine routinely compiles from source. Same command, same version, one machine gets a downloaded binary and the machine next to it gets a compiler run.



Corn

Which nobody is notified about. It just takes longer.



Herman

Sometimes much longer. Now, the other way to ship a Python CLI: bundle the interpreter. PyInstaller packs your bytecode, the dependencies, and a whole copy of the Python interpreter into one executable. Nuitka compiles your Python to C, builds it with a C compiler, and links it against the Python runtime. Both give you a binary that runs on machines with no Python installed. Both are enormous.



Corn

This is where the benchmark made me wince.



Herman

It should make everyone wince. Someone measured a pandas script on Apple silicon. PyInstaller's onefile mode: about thirty-six megabytes, and thirty-five seconds to start. Every launch. Not the first launch — every one, because a onefile binary unpacks itself into a fresh temp directory each time you run it. The same program shipped as a folder started in about two-tenths of a second.



Corn

Thirty-five seconds. A sloth finds that hasty.



Herman

Nuitka's onefile was smaller, around twenty-three megabytes, but took seventy-seven seconds to build. cx_Freeze shipped a hundred and forty-nine megabytes across two thousand two hundred and seventy-three files. For one pandas script.



Corn

Two thousand files. That's not an install, that's a suburb.



Herman

The subtle one is the matplotlib trap. PyInstaller's matplotlib hook rebuilds the font cache on every launch — by design. As a folder that's under eight seconds, as onefile it's twenty-six, versus about a quarter of a second for Nuitka and cx_Freeze. People pick onefile because it looks tidy, one file, and they ship a CLI that feels broken, and nothing in the install step ever tells them why.



Corn

The convenience flag is the slow one. Nobody puts that on the tin.



Herman

Rust distribution, briefly, because it's the counterpoint. The recommended path for a Rust CLI is musl targets with static linking flags — that gets you one binary that runs everywhere, Alpine included. Though one person tested an Alpine-built musl binary and it still showed as dynamically linked, so you verify, you don't assume. Then there's cargo install, the closest thing to npm install dash dash g, with one big difference: it compiles from source on the user's machine and drops the result into the cargo bin directory. Your user needs a full Rust toolchain to install your tool. cargo-binstall exists to fetch prebuilt binaries from GitHub releases instead. Three different things behind install instructions that all look the same in a README.



Corn

And the deepest layer is the one I actually respect. The deb.



Herman

The .deb is my favorite artifact in all of this. It's not a tarball. It's an ar archive with exactly three members — a format-version file, a control tarball holding metadata and install scripts and checksums, and a data tarball with the actual files. Built by a maintainer you've never met, with patches you never see. And the only real answer to that trust problem is reproducible builds — .buildinfo files that record the build environment and get published, so an independent party can rebuild the thing and verify nothing was slipped in during the build.



Corn

How many people actually verify?



Herman

Almost nobody. The verification layer exists, it's public, and it sits there unused. That's the shape of the whole install step — every layer of honesty is optional, and almost everyone opts out.



Corn

Which is the transition, really. We've covered how they're built. The scarier question is what the tool actually is once it's installed.



Herman

And that's where I get to my favorite argument on the internet. The claim everyone repeats is that uv is fast because it's written in Rust. And there's a strong counter-argument that uv is fast because of standards. PEP 658 — static metadata served directly from PyPI — went live in May twenty twenty-three. uv launched February twenty twenty-four. One commenter's line: a tool like uv couldn't have shipped in twenty twenty. The standards weren't there yet. The speed comes from being able to fetch metadata without downloading whole packages, from a wheel-first strategy, from being designed to install across environments instead of bootstrapping a slow Python tool into each one. Rust is maybe the delivery vehicle, not the engine.



Corn

Ten-to-a-hundred-x faster than pip, and possibly none of it is the language.



Herman

It's contested in the same threads — people push back — but the strongest supporting fact is that pip's own slowness was diagnosed as pre-compiling bytecode, importing hundreds of modules to do nothing, building complex wrappers before connecting to the internet. Written in Python versus Rust, as one commenter put it, is simply not relevant in that context.



Corn

I want that printed on the side of every rewrite announcement.



Herman

Now scope. The uv README calls it a single tool to replace pip, pip-tools, pipx, poetry, pyenv, twine, virtualenv, and more. Seven-plus tools. And it installs and manages Python interpreters themselves — you can ask it to install three different CPython versions and it goes and gets them. The release this week shows managed CPython builds shipping with pip, zlib, and Expat baked in, plus a GraalPy build. It's a package manager that also redistributes language runtimes and their C dependencies.



Corn

The command that installs Python is itself a Python distributor. That loop closes quietly.



Herman

ruff replaced six linters — Flake8 plus dozens of plugins, Black, isort, pydocstyle, pyupgrade, autoflake — with over nine hundred built-in rules. And the speed reports are absurd. Nick Schrock, on a quarter-million-line codebase: pylint takes about two and a half minutes, ruff does the entire business in point-four seconds. Nearly a thousand times faster. Literally, he says, not a typo. The isort creator switched and said the only downside was that it was so fast he couldn't believe it was working until he intentionally introduced errors.



Corn

That's the most relatable sentence in benchmarking.



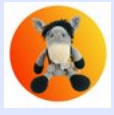
Herman

And now, our regular segment, One-Star Getaways.



Corn

A photograph of the view. That's the onefile binary of hospitality.



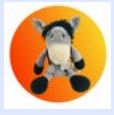
Herman

You're not booking it.



Corn

I'm not saying that.



Herman

You're saying that. Ripgrep, since we're cataloguing hidden scope — people file it under "fast grep," and it's a custom regex engine with an optional PCR-two fallback for look-around, full Unicode on by default, a gitignore implementation, a directory walker, file-type filtering, and as of the major release last year, awareness of Jujutsu repositories. The README's own benchmark: over a thirteen-gigabyte corpus, a pattern that takes ripgrep about fifteen and a half seconds takes GNU grep with Unicode eight and a half minutes.



Corn

Eight and a half minutes against fifteen seconds. The other fifty-three seconds were the excuse to go get coffee.



Herman

That's what a command name hides now. You type three letters, and you're invoking a regex engine and a walker and an ignore-parser, and the install step told you none of it. The name is a lie of omission — not a false statement, just a silence where a description should be.



Hilbert

One line in the log. That's how I knew.



Hilbert

I had two machines, same command, same version string, behaving differently. Took me a weekend. One had pulled a prebuilt wheel, the other had compiled from source and picked up a different system library, and the only difference in the logs was the compiled one had an extra line about a C compiler being invoked. Herman would have found it in an hour, but Herman reads logs the way other people read menus.



Hilbert

I kept both install logs side by side in a text file. Kept it for years. I can still tell you which machine was which.



Hilbert

People call that hidden complexity a scandal. It isn't. It's the most reasonable arrangement in the world. If the install step told you what it was actually doing, nobody would install anything. My brother-in-law says the same about boiler parts. He's not to be trusted on anything else, but on that he's right.



Hilbert

I have a rule — I don't install a CLI on a machine I can't reach over SSH. I want to be able to read the install log afterward. Broke that rule once, on a laptop at a client site in Beersheba. Still regret it.



Corn

You kept a text file of two install logs for years.



Hilbert

It was a good text file.



Herman

The extra line about the C compiler — that's the piece that would have taken me the longest, honestly. Not noticing the behavior difference. Caring that the logs differed.



Herman

I ran ruff's install four ways on the flat machines last month, if we're confessing things. Same version, four different on-disk artifacts. Never occurred to me to keep the logs.



Corn

The two-machine story is also the honest answer to the misconception, though. People think a CLI is one program. uv is a toolchain wearing a command name. Ruff is six linters. Ripgrep is a regex engine doing a grep impression.



Herman

And the companion misconception — pip install ruff installs a Python program. It installs a Rust binary with a stub module around it. The Python in the package is mostly packaging.



Corn

Which raises the question we haven't asked. If the command name is a lie of omission, what would an honest install step even look like? The reproducible-builds people built their answer — buildinfo files, independent verification — and it sits there unused. So is that a tooling problem or a trust problem?



Herman

I think it's a habit problem. The tools exist. What's missing is anyone feeling the absence. You only read the label after the bottle hurts you.



Corn

And the labels are shrinking. Every year a command replaces six more tools, and the name gets shorter and the payload gets fatter. The black box isn't getting smaller.



Herman

Fifty-three megabytes said it was four letters.



Corn

That's the show. Thanks to Hilbert Flumingtop at the desk, who has a good text file he's willing to share. If today's episode changed how you look at anything you've installed, rate and review us — and tell us about a command that turned out to be something else entirely under the hood. If you enjoyed this one, try episode ninety-one, The Story Behind the Show; episode twenty-four ninety-seven, Tracing One Python Print Through six Abstraction Layers; and episode eight, Building Your Own Whisper. This has been My Weird Prompts. We'll see you tomorrow.

SOURCES

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

1. **GitHub, astral-sh/uv , retrieved 2026-11-14, latest release 0.13.8 (2026-11-12). [primary]**
   https://github.com/astral-sh/uv

2. **GitHub, astral-sh/ruff , retrieved 2026-11-14, latest release 0.17.5 (2026-11-12). [primary]**
   https://github.com/astral-sh/ruff

3. **GitHub, BurntSushi/ripgrep , retrieved 2026-11-14, latest release 15.2.0 (2026-07-15). [primary]**
   https://github.com/BurntSushi/ripgrep

4. **setuptools docs, Entry Points, v84.2.1, retrieved 2026-11-14 . [primary]**
   https://setuptools.pypa.io/en/latest/userguide/entry_point.html

5. **pypa/installer source, `_SCRIPT_TEMPLATE` and Windows launcher table, retrieved 2026-11-14 . [primary]**
   https://github.com/pypa/installer/blob/main/src/installer/scripts.py

6. **pip vendored distlib `scripts.py`, shebang handling, retrieved 2026-11-14 . [primary]**
https://github.com/pypa/pip/blob/master/src/pip/_vendor/distlib/scripts.py

7. **PEP 513, A Platform Tag for Portable Linux Built Distributions, retrieved 2026-11-14 . [primary]**
https://peps.python.org/pep-0513/

8. **manylinux/auditwheel README, retrieved 2026-11-14 . [primary]**
https://github.com/manylinux/auditwheel

9. **Debian, Debian Packaging Tutorial, `.deb` ar archive structure, retrieved 2026-11-14 . [primary]**
https://www.debian.org/doc/manuals/packaging-tutorial/packaging-tutorial

10. **Debian, Advanced packaging, §10.7 Reproducible build, retrieved 2026-11-14 . [primary]**
https://www.debian.org/doc/manuals/debmake-doc/ch10.en.html

11. **cargo-binstall README, retrieved 2026-11-14 . [primary]**
https://github.com/cargo-bins/cargo-binstall

12. **unixy.io, Why Go Replaced Python for CLI Tools**
And Where Python Fights Back, published 2024-10-17, retrieved 2026-11-14.
https://unixy.io/blog/go-replaced-python-cli/

13. **mindFLOW, The State of 'Let's Rewrite It in Rust', published 2025-06-10, retrieved 2026-11-14.**
https://halyph.github.io/blog/2025/2025-06-10-rewrite-in-rust/

14. **Runcell, PyInstaller vs Nuitka vs cx_Freeze: A Measured Packaging Benchmark, retrieved 2026-11-14 (measured on macOS 26.5.2, Python 3.13.1).**
https://www.runcell.dev/blog/pyinstaller-vs-nuitka-benchmark

15. **Orbifolds and Other Games, Portable Python Binary Wheels, retrieved 2026-11-14.**
https://orbifold.xyz/binary-wheels.html

16. **Unix StackExchange, answer by Stephen Kitt on `.buildinfo` files, retrieved 2026-11-14.**
https://unix.stackexchange.com/questions/673090/how-to-verify-if-a-given-package-is-built-in-a-reproducible-way-as-an-end-user

17. **racum.blog, Multi-platform binary releases for a Rust CLI, musl vs glibc targets, retrieved 2026-11-14.**
https://racum.blog/articles/rust-cli-releases/

18. **Rust SME Cookbook, Distributing CLI Tools, `cargo install` vs prebuilt binaries, retrieved 2026-11-14.**
https://rs4ts.dev/18-cli-tools/10-distribution/

19. **inerit blog, static linking verification with `ldd`, retrieved 2026-11-14.**
https://blog.inerit.dev/en/post/346

20. **Python CLI Toolcraft, Nuitka vs PyInstaller for Python CLI Binaries, retrieved 2026-11-14.**
https://python-cli-toolcraft.com/project-setup-dependency-management/distributing-clis-as-standalone-binaries/nuitka-vs-pyinstaller-for-python-clis/

21. **Thought Parameters, Nuitka vs PyInstaller vs cx_Freeze: Python Packaging Compared (2026), retrieved 2026-11-14.**
https://blog.thoughtparameters.com/post/nuitka_vs_pyinstaller_python_packaging/

22. **RendereelStudio, Nuitka vs PyInstaller: Complete Comparison 2026, production build sizes, retrieved 2026-11-14.**
https://beatsyncpro.ai/blog/nuitka-vs-pyinstaller.html

23. **JetBrains Blog, published 2025-11-10, retrieved 2026-11-14.**
https://blog.jetbrains.com/rust/2025/11/10/rust-vs-python-finding-the-right-balance-between-speed-and-simplicity/

24. **Oleg Sotnikov, Rust vs Go vs Python for CLI tools in production, retrieved 2026-11-14.**
https://oleg.is/blog/rust-go-python-cli-tools

25. **Hacker News discussion of Uv is the best thing to happen to the Python ecosystem in a decade, comments dated 2025-10-29, retrieved 2026-11-14.**
https://news.ycombinator.com/item?id=45753725