Episode Book

EPISODE #5965

Compiling From Source: What You Actually Get

PUBLISHED	14 October 2026 at 14:59 UTC
DURATION	27:51
LISTEN	myweirdprompts.com/episode/compiling-from-source-tradeoffs-56838/
HOSTS	Corn, Herman
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.4
SOURCES	10 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:42 UTC

SYNOPSIS

Daniel spent twenty years on Linux desktops without understanding why anyone would compile software from source. This episode takes that question seriously, starting with the assumption everyone carries: that a self-built binary must be better optimized. It mostly isn't. A 1999 kernel recompile gained 1.8 percent; a 2026 Modern C++ benchmark found GCC with profile-guided optimization ran 9.4 percent slower than a plain O2 build, because the optimizer's bets didn't pay off. So why does the source tradition survive? Not for speed — for control over what gets built. USE flags let you compile out codecs, protocols, and integrations you never use, and code that isn't there can't be exploited. We walk through the two poles, Ubuntu's signed reproducible binaries versus Gentoo's Portage, what actually happens during a build, and why watching the compiler work is itself part of what people are buying.

TRANSCRIPT



Corn

Daniel has been on Linux desktops for about twenty years. Twenty years, and he says he has never fully understood why anyone would compile software from source instead of installing a precompiled package.



Herman

That's a good confession, honestly. Most people just nod along and never ask.



Corn

This picks up from the kernel episode, the one about tailoring an OS to your hardware. That led him somewhere more basic. On Ubuntu, you run apt install and you get a binary that somebody else already built. But Linux has this whole other tradition where the compilation happens on your machine, and Gentoo built an entire distribution around it.



Herman

And Daniel wants the real rationale. Not the folklore.



Corn

He wants to know what actually happens when software gets compiled. What decisions get made in that process, and which of them touch performance, functionality, compatibility, security. Then the assumption he wants challenged: that compiling it yourself must produce a better-optimized executable. Is that true, or are the distribution binaries basically as good?



Herman

That's the right question to start with, because that assumption is doing a lot of quiet work.



Corn

Then the practical side. If you run Gentoo, are you really sitting there compiling every package by hand, every update? How do dependencies, upgrades, security patches and rebuilds actually get handled? Can the middle ground work, where you're on Ubuntu but you compile one or two things from source? Why does scientific computing and HPC still build from source so much? And finally, what do you actually lose? What's the maintenance burden, what are the security risks, and what do you give up by relying entirely on your distribution?



Herman

The whole thing comes down to one choice. When you type apt install, you're picking a binary somebody else compiled, tested and signed. When you type emerge, you're the one running the compiler.



Corn

So what are you actually choosing between?



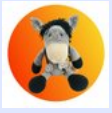
Herman

Let's set up the two poles properly, because they're further apart than people think, and closer than people think, both at once. Ubuntu and APT are the binary model pushed to its logical end. A maintainer compiles the package once, on their infrastructure, and ships that exact binary to millions of machines. Gentoo and Portage are the other pole. The source tarball comes down to your machine, and your machine builds it.



Corn

And the binary model is the default for good reason.



Herman

Very good reasons. It's fast. It's tested before it reaches you. It's signed, so you can verify it came from the maintainer and not from someone in the middle. Modern Ubuntu builds are reproducible, meaning you can rebuild the same source and get a byte-identical result, which is how you audit a binary you didn't compile yourself. And it's the same artifact on every machine, so when something breaks, a million people have the same thing and it gets fixed once.



Corn

So if the binary path is that good, why does the source tradition survive at all?



Herman

Not for speed. That's the thing to get straight up front. The reason source builds persist is control over what actually gets built. Which features go in, which libraries get linked, which code paths exist in the binary at all.



Corn

Control, not speed.



Herman

Control. And the speed argument is the one everybody reaches for first, which is why it deserves the most scrutiny.



Corn

Start with what compilation actually is. Because I think people picture something mystical happening, and it isn't.



Herman

It's completely mundane, and that's the first thing to deflate. When Portage installs a package, it downloads the source into a cache directory. Unpacks it. Runs the compiler with a specific set of flags. Installs whatever comes out. That's it. A binary package is the exact same output, produced once by a maintainer and shipped to you. The difference isn't in the nature of the result. It's who ran the compiler, and with what settings.



Corn

Same product, different cook.



Herman

And that reframes the whole question, because now it's not "is compiled code better than compiled code". It's "whose compilation settings, and do they matter".



Corn

It also raises a question I don't think people ask enough. If the output is the same kind of thing either way, why does the source path feel so different to use? Is that just psychology, or is there something structural?



Herman

There's something structural, and it's the feedback loop. When you install a binary, the process is opaque by design. You type a command, you get a result. When you build from source, you watch all of it happen. You see the configure step checking for libraries. You see the compiler churning through files. You see warnings scroll past. That visibility is part of what people are actually buying, even if they'd never phrase it that way. It's not that the result is better. It's that you watched it get made.



Corn

So the experience of the build is itself a feature.



Herman

For some people, absolutely. It's the same instinct that makes someone want to see the kitchen in a restaurant. You don't necessarily get a better meal. You get to know what went into it.



Corn

So which settings matter?



Herman

Three categories. Compiler flags, the CFLAGS and CXXFLAGS. USE flags, which decide which optional features and dependencies get compiled in. And which optional libraries get linked against. Those three cover almost everything that differs between a self-built package and a distribution package.



Corn

Take the compiler flags first.



Herman

Things like march and mtune, which tell the compiler what CPU to target. And the optimization level, dash O2 versus dash O3. march is the interesting one, because it lets the compiler emit instructions for your specific processor, instructions a generic build can't use.



Corn

And that's where the folklore lives.



Herman

That's exactly where it lives. The theory is that if you build for your exact CPU, you get faster code. And there's a kernel of truth, which is why the folklore survives. Gentoo's own wiki, on the binary host page, says some march settings can reportedly give up to a five percent boost on x86-64 processors, in some cases. Five percent. That's Gentoo's own number, on their own page about their own binaries.



Corn

"Reportedly." "Some cases." That's not a ringing endorsement. That's a shrug with a number attached.



Herman

And it gets weaker. There's a 1999 analysis in Linux Gazette, someone recompiling a kernel on a Pentium 75 with CPU-tuned GCC, and the tuned build came out 1.8 percent faster. One point eight. The conclusion in that piece was blunt, that kernel compiling is not an effective way to optimize a Linux box. That was twenty-seven years ago and the hardware has changed enormously, but the structural reason hasn't.



Corn

What's the structural reason?



Herman

The compiler is already very good. Modern GCC and Clang do a lot of work that used to require hand-tuning. If you write reasonable code and compile it at O2, you're already close to what the hardware can do for most workloads. The gap between a generic build and a tuned build is small because the generic build isn't leaving much on the table.



Corn

And there's a counterexample that's much more recent.



Herman

This is the one that should end the argument. A 2026 benchmark on a Modern C++ virtual-dispatch workload, which is exactly the kind of code where aggressive optimization should help. Profile-guided optimization. You run the program, collect a profile of which branches actually execute, and feed that back into the compiler so it optimizes for the real path.



Corn

That sounds like it should be a clear win.



Herman

It sounds like it. GCC with PGO came out 9.4 percent slower than a plain O2 build. Clang with PGO, 8.7 percent slower. And the reason is interesting. The optimizer used the profile to speculatively devirtualize the calls, replacing indirect calls, which the CPU predicts well, with type guards that mispredict. It made the code dumber while thinking it was making it smarter. AutoFDO, the sample-based variant, only gained 2.4 percent.



Corn

So the most aggressive optimization available made it slower, on the workload it was designed for.



Herman

By nearly ten percent. The author's line was that they've now seen PGO hurt by 9.4 percent on a workload that should have been its best case. That's the entire business in one sentence. Optimization is not monotonic. More aggressive settings do not mean faster code, because the compiler is making predictions and sometimes it predicts wrong.



Corn

Let me push on that, because it feels counterintuitive enough that someone listening might think the benchmark is just an outlier.



Herman

It's not an outlier in kind, only in degree. The general principle is well understood. Every optimization the compiler makes is a bet about what the code will actually do at runtime. Loop unrolling bets that the loop overhead matters more than the instruction cache pressure. Inlining bets that the call overhead matters more than the code size. Devirtualization bets that the type check is cheaper than the indirect call. When the bet pays off, you win. When it doesn't, you lose, and sometimes you lose more than you would have gained.



Corn

So the more aggressive you get, the more bets you're stacking.



Herman

The more bets, and the more correlated the losses can become. That's why O3 is not reliably faster than O2, and why the safe advice is to measure rather than assume. The compiler doesn't know your workload. It's guessing from heuristics and, with PGO, from a profile that might not match reality.



Corn

What about USE flags? Because that's the part I suspect actually matters, and nobody talks about it.



Herman

That's the real answer to Daniel's question. Compiler flags are the flashy part and they mostly don't move the needle. USE flags are the boring part and they're the entire reason source builds persist. A USE flag decides whether an optional feature gets compiled into the binary at all. So you can build a package without the optional integration you don't use, the codec you never play, the protocol you never speak.



Corn

And that's not a performance argument.



Herman

It's a security argument, and a much better one. Every feature compiled in is code that exists on your system, that can have bugs, that can be attacked. If you compile it out, the code isn't there. It can't be exploited because it doesn't exist. One engineer, describing what they did in production, said omitting unused features and integrations became an important part of their security posture.



Corn

So the honest pitch for source builds is smaller attack surface, not faster code.



Herman

That's the trade. You're reducing what's present on the system rather than making what's present run faster.



Corn

Give me a concrete version of that, because "smaller attack surface" is abstract.



Herman

Think about a media player. The distro build ships with support for every codec the maintainer could reasonably include, because they don't know what you'll play. Every one of those codecs is a parser for untrusted input, and parsers for untrusted input are where a huge fraction of real vulnerabilities live. If you only ever play two formats, you can build the player with support for two formats and nothing else. Now a malicious file in a format you don't use has nothing to exploit. The parser isn't installed. That's the whole idea in one example.



Corn

And the same logic applies to network services.



Herman

It applies everywhere. A web server built without a module you don't use has no code from that module to attack. A library built without an optional compression backend has no decompression code to exploit. The principle is uniform. Code that isn't there can't be broken into.



Corn

Now flip it. What do the distribution maintainers have that you don't?



Herman

Quite a lot, and this is the part self-compilers tend to underrate. Take Gentoo's official amd64 binaries. They're built with march set to generic x86-64, mtune generic, dash O2, dash pipe. Deliberately conservative, so a single binary runs on any x86-64 machine, whether it's five years old or brand new.



Corn

That's the compromise. They give up the five percent so the binary works everywhere.



Herman

They give it up on purpose. And in exchange they get things the individual builder can't easily reproduce. Reproducible builds, so the binary is auditable. GPG signing, so you know it's authentic. And since May, Portage requires valid OpenPGP signatures for any remote binary host by default. So the binary path is now cryptographically gated end to end.



Corn

That's a security posture too, just a different one. Signed binaries versus absent features.



Herman

Both are real. They just buy different things. The maintainers also have testing infrastructure you don't, and a subtle advantage, which is the ability to delay moving to a new stable GCC. The binhost holds off on a compiler upgrade until the binaries stay compatible with systems that haven't upgraded yet. If you're compiling locally and you pull a new GCC, you rebuilt the world with a compiler that might have new bugs.



Corn

There's a hidden cost that doesn't get mentioned either. Something from this month.



Herman

Michał Górny, one of the Gentoo developers, wrote a plea on the twelfth of this month, asking people not to use a particular Python build backend called uv-build. It's Rust-based, and the problem is that it drags in a one point two to one point five megabyte dependency that every Gentoo user has to rebuild, repeatedly. Twelve releases in thirty days. And he compares it to flit-core, a lighter backend that's forty-five kilobytes.



Corn

So the tax isn't your time. It's a heavy upstream dependency forcing everyone downstream to rebuild.



Herman

His line was that for every hundred milliseconds running that tool to install one release, Gentoo users will spend many hours rebuilding new versions of it over and over. That's the part of the source-build story nobody tells. You have chosen to run the compiler, and what you run it on is no longer entirely your decision.



Corn

That's the thing about choosing to build everything. You inherit everyone else's build choices.



Herman

All of them. Down to their choice of build backend language.



Corn

So let's deal with the practical objection, because I think this is the biggest misconception in the whole topic. The picture of a Gentoo user hand-compiling every package, one at a time.



Herman

It's a picture of something that doesn't exist. Portage is a full package manager. It resolves dependencies, handles upgrades, applies security patches, and rebuilds what needs rebuilding. The user doesn't hand-compile anything. You run emerge with the packages you want, and Portage does the rest.



Corn

So it's automated.



Herman

Largely automated. MAKEOPTS defaults to running parallel build jobs, one per core, with a load limit matched to the core count. So it's using your whole machine to compile. The process is not manual. What it is, is slow, and heavy on resources.



Corn

How slow, how heavy?



Herman

Depends on the package and the machine. Some packages need gigabytes of RAM during the build. Compile time is the real cost, and it scales with the machine. Gentoo's own documentation on minimizing build time just tells you to look at what you're dragging in and substitute prebuilt dash bin versions where they exist. Which tells you something. The source distribution wrote a page about how to compile less.



Corn

And that points at the pivot, which is fascinating to me.



Herman

This is the most interesting recent development. In December 2023, Gentoo launched an official binary package host. Over twenty gigabytes of packages for amd64 and arm64, updated daily. So the source-based distribution started shipping binaries.



Corn

And they were honest about the compromise.



Herman

Completely honest. Their news post says, and I love this, "But hey, that's not optimized for my CPU! Tough luck. You can still compile packages yourself just as before!" That's Gentoo telling you the argument you're about to make and pre-empting it. Tough luck.



Corn

And then the important part. They said they're still a source-based distribution and they are not aiming for a binary-only install with no compilation at all.



Herman

That's the line that matters. They're not abandoning the source model. They're putting a binary path alongside it, for people who want the base system fast and don't need every package built locally. In February 2024 they added x86-64-v3 binaries, the microarchitecture level above baseline, so you get a bit of the CPU-specific benefit without compiling.



Corn

How big has the binary side gotten?



Herman

Their 2025 retrospective, from January, has the numbers. Eighty-nine gigabytes of binary packages on the mirrors for amd64. A hundred and fifty-four distinct installation stages, built weekly. Thirty-one thousand ebuilds covering nineteen thousand packages. The binary infrastructure is now a substantial thing alongside the source tree, not a side project.



Corn

And that's the hybrid reality. You can mix them.



Herman

Portage mixes freely. There's a feature setting, getbinpkg, and with it Portage installs a binary when one matches your configuration and compiles locally otherwise. So you can be on source-based Gentoo and pull binaries for most things.



Corn

But there's a catch.



Herman

There's exactly one catch, and it's the whole tension in one line. If your USE flags differ from the profile default, Portage compiles from source. The binhost only carries binaries matching the default configuration. So the more you customize, the less the binary path helps you.



Corn

The more you want the thing that source builds are for, the more source builds you have to do.



Herman

Which is a clean little law. Customization forfeits the convenience. You don't get to have the customization and skip the compilation, because the compilation is the customization. If you want a binary, it's somebody else's choice of flags.



Corn

What about the middle ground for people not on Gentoo at all? Arch, say.



Herman

Arch has its build system, the PKGBUILD. It's a Bash script that a tool called makepkg reads. The script knows how to download the source, compile it, and package the result into an archive that pacman installs. So Arch is binary by default, but the build recipe for every official package is right there, and there's a user repository, the AUR, with thousands of contributed recipes for software the official repos don't carry.



Corn

So you get the source path without being on a source distribution.



Herman

Without being on one. Arch's own documentation lists the use cases, and it's candid about them. One of them is rebuilding your entire system with your own compiler flags, quoting, in the manner of FreeBSD. Building custom kernels. That's the ports model, inherited from the BSDs, available as an option on a binary distro.



Corn

What about Ubuntu? Because that's where Daniel lives, that's the whole premise of the question.



Herman

You can do it there too, and it's not obscure. You pull the build dependencies for a package with apt build-dep, then build a .deb from source with dpkg-buildpackage. Or use checkinstall to wrap a manual build into a package so it doesn't pollute your system. And then you keep APT for everything else.



Corn

So the answer to his middle-ground question is yes. You can compile one thing and install everything else as binaries.



Herman

You can, and it's the pragmatic hybrid. You get the one package you need, built the way you need it, and you don't sign up for compiling the world.



Corn

Where does compiling from source stop being optional and start being necessary?



Herman

Scientific computing and high-performance computing is the clearest case, and it's the one where the performance argument actually holds. It's not your CPU architecture. It's which math library you link against.



Corn

Explain that.



Herman

Numerical code spends its time in the linear algebra routines, the BLAS libraries. And different BLAS implementations run at very different speeds for the same operation. A tuned vendor library like Intel MKL, or OpenBLAS, can be substantially faster than a generic distro build. So if your whole computation is matrix operations, which library you link against is the difference that matters.



Corn

So for once the optimization argument is real.



Herman

For once it's real, and it's not about compiler flags, it's about library selection. Which is the third category of build-time decision, and the one that finally justifies the effort. And this is evolving, which is a good sign. Gentoo adopted FlexiBLAS as the primary way to switch BLAS implementations at runtime in their 2025 work. So you can pick the tuned library without rebuilding everything.



Corn

That's a nice resolution. They moved the choice from build time to run time.



Herman

They made the choice reversible. Which is the direction the whole field is moving.



Corn

What else needs source builds?



Herman

Security hardening, the USE flag argument again, in a setting where it's taken seriously. Newer versions than the distro ships. Optional features the maintainer didn't build in. Hardware-specific optimizations, which is real but niche. And building against a specific version of a dependency, which matters when your code has to match something else exactly.



Corn

There's a nice summary of the Gentoo appeal from the discussion when they went binary. Somebody said the appeal was never compiling everything. It was the freedom to install anything you want on nearly any hardware, and the selling point was the flexibility and the community, not the package manager.



Herman

That reframes it well. The compilation is the mechanism. The freedom is the point.



Corn

Now the trade-offs, and I want to push on the ones that cut against source builds, because the romantic version leaves them out.



Herman

They're real. The first is that you own everything. You own the patching, the reproducibility, the rebuilds. If a vulnerability drops and you're the one compiling, you're the one who has to notice and act and rebuild.



Corn

Which is a genuine security risk, because most people aren't watching.



Herman

Most people aren't watching. There's a real irony there. The USE flag argument says smaller attack surface is safer. But you've also removed the maintainer who's paid attention to the security list for you. If you're not diligent, your hardened system is more exposed than the boring one, because nobody's patching it on your behalf.



Corn

I ran the stock distribution kernel for years for exactly that reason. I never had to think about it.



Herman

And there's no shame in that. That's the deal. You gave up the control and you got the maintenance done for you.



Corn

What are the mechanical problems?



Herman

Dependency conflicts, particularly around USE flags. If package A wants a feature and package B is incompatible with it, you can get into a state the binary world wouldn't have let you reach, because the maintainer resolved it once and shipped the resolution. You can also hit circular dependencies, where two packages need each other to build. And rebuilds cascade. Change a flag on a common library and a lot of packages need



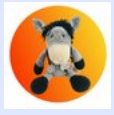
Herman (continued)

rebuilding.



Corn

And the storage and time costs are real. Compile time, build RAM, disk for the build tree.



Herman

Real, and worth stating plainly. You want to know why someone would compile from source. You should also know they spend a lot of CPU doing it. The binhost exists because building from source takes time, which is Gentoo's own explanation for why they built it.



Corn

Now the other direction. What's wrong with relying entirely on distro binaries?



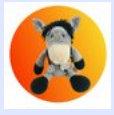
Herman

The version cadence is the big one. On a stable distribution, you get the version the maintainer chose, and you wait for the next one on their schedule, not yours. If your work needs something two years newer, the binary repos don't have it.



Corn

Or an optional feature the maintainer turned off.



Herman

That's the one that sends people to compile their first package. The flag you need is off in the distro build, and no binary exists with it on. So you build the one package, and you've joined the middle ground.



Corn

Hardware-specific optimization, which is real but small.



Herman

Real but small, and we've already measured it. Five percent at the absolute ceiling on some workloads, commonly much less, sometimes negative if you get aggressive. So if you're going to compile for performance, go in knowing the ceiling.



Corn

Let me make the case for the other side, because you've been fair to source builds and I should make the binary case properly.



Herman

Go ahead.



Corn

The binary path is the default because it's better for almost everyone. It's fast, it's tested, it's signed, it's reproducible, it's the same artifact a million people are running, so when it breaks it gets fixed. The maintainers have the compiler upgrade timing right, which an individual doesn't. And the thing you give up, the customization, is worth almost nothing to a desktop user.



Herman

It's worth almost nothing to most desktop users.



Corn

The honest answer to Daniel is that the source path is for the person who needs something the binary can't give, and knows exactly what it is. Everyone else is paying a tax for a benefit they can't measure.



Herman

Which is where I land, with one caveat.



Herman

The caveat is that the control is real even when the performance isn't. Choosing what's on your system, down to which code paths exist, is a legitimate thing to want, and it's not irrational just because it doesn't show up in a benchmark. That's the part the performance argument obscures. The performance is small. The control is the actual product.



Corn

Control is the product, performance is the marketing.



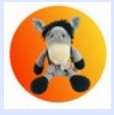
Herman

That's the lot in one line.



Corn

More and more of this is moving to the hybrid. Gentoo ships binaries. Arch ships a source recipe for everything. Ubuntu lets you compile one package. The line between the two sides is blurring.



Herman

It's blurring from both directions, which is the healthiest possible outcome. The binary distributions are adding source paths. The source distribution added a binary path and held the line on staying source-based. Everybody has decided the interesting question is how much you want to customize, not which camp you're in.



Corn

How much do you want to customize, and are you willing to pay for it in compile time.



Herman

The one thing that makes the choice clean, though, is that you're choosing control, not speed. If you want performance, the numbers say build at O2 and stop. If you want control, that's a different project with a different justification.



Corn

The numbers are the easy part. They point the wrong way.



Herman

The numbers point the wrong way, and the honest version is that most people who compile from source are doing it for the control, and should just say so.

SOURCES

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

1. **Gentoo, 2023-12-29 [primary]**
   https://www.gentoo.org/news/2023/12/29/Gentoo-binary.html
2. **Gentoo wiki [primary]**
   https://wiki.gentoo.org/wiki/Gentoo_Binary_Host_Quickstart
3. **Gentoo wiki [primary]**
   https://wiki.gentoo.org/wiki/Binary_package_guide
4. **Gentoo, 2026-01-05 [primary]**
   https://www.gentoo.org/news/2026/01/05/new-year.html
5. **Michał Górny, 2026-10-12 [primary]**
   https://blogs.gentoo.org/mgorny/2026/10/12/a-plea-not-to-use-uv-build/
6. **ArchWiki [primary]**
   https://wiki.archlinux.org/title/Arch_build_system
7. **Gentoo, 2024-02-04 [primary]**
   https://www.gentoo.org/news/2024/02/04/x86-64-v3.html
8. **Hacker News, 2023-12-29**
   https://news.ycombinator.com/item?id=38804135
9. **Linux Gazette #37, Feb 1999**
   https://ftp.gwdg.de/pub/linux/misc/gazette/issue37/martinez.html

10. **Modern C++ // dev, 2026-03-10**
https://moderncpp.dev/articles/profile-guided-optimization/