Episode Book

EPISODE #6727

Keeping Private Config Safe in Public Stacks

PUBLISHED	21 November 2026 at 16:36 UTC
DURATION	19:24
LISTEN	myweirdprompts.com/episode/public-stacks-private-configuration-61433/
HOSTS	Corn, Herman, Larry, Hilbert
SCRIPT MODEL	z-ai/glm-5.3
VOICES	omnivoice
PIPELINE	V5.36
SOURCES	15 cited by the research (listed at the end)
BOOK GENERATED	21 November 2026 at 19:31 UTC

SYNOPSIS

Shipping public artifacts avoids registry logins and deployment friction, but managing private sidecars, credentials, and custom templates requires durable boundaries. A close look at Docker Compose reveals where override files quietly break down on subtraction and path resolution, how include elements offer composition over customization, and why drop-in directories survive updates. The discussion shifts to Claude Code plugins, contrasting Docker's convention-driven seams with platform-enforced data directory lifecycles and strictly validated user configuration.

TRANSCRIPT



Corn

The standard line about open-sourcing your home server project is that it's generosity. Daniel's take is better: his container image is public on GHCR, so he doesn't need authentication to pull it — the public artifact is the one that's easiest for the author himself to deploy. The catch is the rest of his life doesn't belong in it. He's built Canopy, this progressive skill discovery system, and he's asking about what he calls configuration carve-outs: shipping a public stack while leaving a sanctioned, durable place for the private stuff. Not just environment variables — say he adds a voice assistant sidecar to Canopy, that's a private addition to a public artifact. He's seen the Docker Compose override file as the traditional answer and wants to know what else exists and what actually holds up. And he's noticed the same shape in Claude Code plugins: ship the plugin publicly, but every user needs somewhere for their own templates and reference files that survives updates. Same problem or not?



Herman

And the reason it's a real question rather than a styling preference is that the two ecosystems answer it very differently. But hold on — before the mechanisms, the thing I want to defend is Daniel's premise, because it's the underdiscussed half. A public image on GHCR needs no credentials to pull. His own deployment scripts just... Work. No image pull secrets, no registry login in CI, nothing. Public-by-default is operationally cheaper for the author, not just generous to strangers. That's a odd incentive structure when you think about it — the path of least resistance and the path of maximum sharing are the same path.



Corn

Rare to find a free lunch in infrastructure.



Herman

Right, and the carve-out is what keeps that lunch safe. The moment you have private things in the stack, you need a seam between what you ship and what you keep. A configuration carve-out is a deliberate, documented boundary — not an accident of what git ignores, not "just use env vars and good luck." The real test of any carve-out pattern is whether it survives because the platform enforces the seam or only because the user is disciplined about it. Keep that split in mind, because it's the spine of the whole question.



Corn

So the override file. Why has it become the default answer?



Herman

Because it literally is the default. Compose reads compose dot yaml and then, automatically, with no flags, an optional compose dot override dot yaml sitting next to it. The user doesn't configure anything — they just create the file and Compose picks it up. Docker's own docs describe the second file as something that might live in a different git repository or be managed by a different team. They designed the private half of the split into the tool's default behavior. That zero-discipline quality is a large part of why the pattern survives at all — any carve-out that requires the user to remember a flag is already weaker than one that happens on its own.



Corn

You're describing a mechanism that works while I'm asleep, which I approve of on principle.



Herman

For additions, yes. Daniel's voice assistant sidecar is the textbook case: he writes a private override file declaring one new service, mounts it into the same network, done. The public stack is untouched, the private addition lives in his private repo, and if he publishes a new image tomorrow, nothing collides. Where it falls apart is subtraction. The merge rules concatenate multi-value options — ports, DNS, tmpfs. If the public compose file declares a port and you want it gone, the override file cannot remove it. You can add a port. You can't take one away.



Corn

The Roach Motel of configuration.



Herman

Ha — check in, never check out. Environment and labels merge by key, so your private value beats the public one for a given key, and volumes merge by container mount path, which is actually quite elegant — your local path replaces the public path for the same mount point inside the container, without touching other mounts. But the pattern holds up for adding a private sidecar, and quietly betrays you the moment your private layer needs



Herman (continued)

to modify a public service rather than extend it.



Corn

And there's the relative path thing, which is where everyone I know has been burned.



Herman

The most-cited limitation, yes. Paths in override files resolve relative to the base file, not the override file. Docker's reasoning is that override files need not be valid standalone Compose files — they're fragments, so they don't get their own directory context. Your override file could reference something like dot-slash dot-slash, and it resolves from wherever the base file lives. Their suggested mitigation is running the config command to review the merged result before you bring it up, which is the software equivalent of checking your work with a calculator you also have to trust.



Corn

At least it prints what you're about to get.



Herman

It does, and that's not nothing. But there's a newer mechanism that fixes the path problem outright, and it's worth separating from the override file because people keep conflating them. The include element. Each included file loads as its own Compose application model with its own project directory, so relative paths resolve correctly per file — it solves the exact problem override and extends create. And it can pull remote Compose files, from OCI artifacts or git repositories, so a public stack can be consumed as a versioned dependency, which is Daniel's whole public-artifact model taken to its logical end.



Corn

I was waiting for the catch.



Herman

The catch is that include refuses conflict. If any resource in an included file collides with one in your including file, Compose errors out — Docker says so explicitly, to protect the included author's resources. Include is composition, not customization. You can't use it to change the public stack; you use it to build on top of the public stack. Docker's own docs note that the global override file is what overrides the merged include model — so the two mechanisms stack, with include assembling and override adjusting.



Corn

So the pattern that actually survives subtraction is the oldest one in computing.



Herman

Say it.



Corn

The drop-in directory. nginx has been doing it forever — the main config includes everything in conf dot d, and the base config declares nothing in that directory. You drop your file in, the loader globs the directory, and there is nothing to collide with because the base never claimed the space.



Herman

And it outlived how many generations of config tooling? The base declaring nothing in the private directory is the deep design move — it means the user's layer can't conflict with the shipped layer by construction, not by discipline. Now, the gap Docker still hasn't shipped, as of today: there's no way to inject a single file into a container path without clobbering a directory. If you mount conf dot d as a volume, you destroy the config files already in it. The long-open feature request is for a copy key that injects files without that clobbering, and the workaround in the meantime is enumerating individual files as volumes — which the request itself notes gets you pages of file entries for a modest stack.



Corn

The DIY version of a feature the tool should have.

L

Larry

Friends, have you ever shipped a public artifact only to find your private configuration seam has been garbage collected, orphaned, and swept away about fourteen days later? Then you need the Canopy Carve-Out Caddy, the only sidecar-shaped storage crate with a documented merge seam that concatenates your ports, replaces your single values, and merges your volumes by container mount path, all while surviving every update. And listen, the price is dropping while I talk, it was two hundred dollars, now it is one hundred and fifty, now it is ninety nine, and I am throwing in a lifetime supply of environment variables, a decorative override file, a laminated copy of the path resolution rule, and a small donkey. But wait, the deal ends before I finish describing it, so it is over, it is gone, the offer has already expired, which is exactly the kind of durability your private layer deserves.



Herman

I own one of those, you know. It sits by the desk and holds precisely one override file and a cable. It has changed nothing about my life.



Corn

Mine holds leaves.



Herman

Where was I — the copy key gap. The point I'd add before we leave Docker: override holds up for addition, include is composition, conf dot d is the pattern that survives because the base declares nothing there. Three seams, three different survival strategies. Now watch what happens when the same problem arrives in Claude Code plugins, because the platform had to actually solve it.



Corn

This is the part Daniel's really poking at. Ship the plugin publicly, but each user needs templates and reference files that survive updates.



Herman

And Claude Code's answer is that the durable layer is a named directory with a lifecycle contract, not a convention. There are two variables, and the difference between them is the whole story. The plugin root variable points at the install cache — one directory per



Herman (continued)

installed version. The data directory variable resolves to a per-plugin path under your dot claude directory, and the contract on that one is explicit: it survives plugin updates. The install cache is orphaned and garbage collected on every update — orphaned version directories get swept about two weeks later. So a plugin author who writes user templates into the plugin root has built a machine that eats the user's templates every release.



Corn

And the data directory?



Herman

Only removed on the last uninstall of the plugin, across scopes — and even then there's a keep-data flag to preserve it. Created lazily, too, the first time the variable is referenced, so an install doesn't litter directories for features nobody used.



Corn

That's a named volume Compose guarantees the pull command won't touch.



Herman

That's exactly the analogy I'd reach for. Same moral structure, different platform. But there's a second half — user config. The plugin manifest can declare values Claude Code prompts the user for, with five types: string, number, boolean, directory, file. Declared once, and then the values get substituted into MCP configs, hook commands, skill content, and exported as environment variables the plugin can read. For the templates case, you declare a directory option, the user points it at their own folder, and the skill reads from there.



Corn

Which is the sidecar pattern again. Declare the shape publicly, fill it privately.



Herman

What makes it hold up is the strictness. The user config section is a strict object — an unrecognized key inside one of those options is an error, and the plugin doesn't load. Top-level unknown fields in the manifest are merely stripped with a warning, which is the forgiving choice in the right place. The validator even catches a reference to a user config key the manifest never declared. The tooling enforces the carve-out rather than trusting the author to be consistent — that's the platform shipping the contract Docker leaves to a documentation page.



Corn

Is that the real answer to Daniel's "is it the same problem," then? Same shape, better enforcement?



Herman

Same shape, and the plugin ecosystem solved it more completely because it had to. Plugins update far more often than container images. A home server image might get a new tag weekly; a plugin marketplace can push updates daily, and auto-update is on by default in places. A seam that's convention plus docs gets tested every update. A seam that's platform-enforced survives the update schedule. Daniel asked whether it's really the same problem — my answer is yes, but the stakes differ, and the stakes are what forced the better contract.



Corn

You're giving the platform credit for generosity. It's liability management.



Herman

Partly, sure. The underreported bit, though, is the trust asymmetry. A plugin that arrives from a repository gets a trust dialog before it loads. Its MCP servers go through per-server approval. Background monitors don't load at all. None of that — neither the trust check nor those limits — applies to a plugin sitting under your own skills directory in your home folder. Your private layer is trusted by virtue of being yours. That's a carve-out in the security model, not just the config model, and it's the same logic as your private override file on your own machine.



Corn

Private means accountable to exactly one person.



Herman

Who is also the one holding the backups. There's one more wrinkle I find unsettling, though, as a man who distrusts systems that report themselves fine. An administrator can set a strict customization policy that silently skips a plugin folder — no error, and the plugin listing shows nothing. A carve-out the platform can close without telling you is a different kind of seam than one you control. In Docker, the override file is yours; nobody's policy hides it from you. In the plugin world, your private layer can be vetoed by a setting you may not even know exists.



Hilbert

That's the truth of it.



Hilbert

Herman, you said "enforced." I'd go further. The seam has to be a different thing entirely, not the same thing with a switch on it. I have a label maker. Had it years. The one with the keyboard and the little cartridge. I keep two cartridges — the house cartridge and the lending cartridge. Anything in the house gets the house cartridge. Anything I lend out gets the lending cartridge, because I want it back and the label says whose it is.



Corn

Two labels per object, in effect.



Hilbert

One label, but the cartridge decides what the label means. They are not interchangeable. There's a notebook, small, spiral, and it lists every label the lending cartridge has ever made. Neighbor still has my long level. Seventeen months. It's in the book.



Herman

So the override file, in your version, would be...



Hilbert

The same cartridge with a switch on it. No. The switch can be flipped. The cartridge can't. You want my confidence in a seam, make it a different physical thing.



Corn

And the label maker itself? House cartridge or lending?



Hilbert

That's the trouble. I labeled it with the lending cartridge by mistake. Sat with that a while. Couldn't decide whether it was mine or on loan to myself, so it went in the drawer with the label on it. I use the old one now. Brother-in-law gave me the notebook habit, back when I drove for a living. The book works. Corn, you'd have used the override file, wouldn't you.



Corn

I'd have used the drawer.



Herman

The strictness thing stands, though — his instinct and the platform's are the same instinct. Two weeks, by the way. That's the sweep interval on the orphaned install cache. Fourteen days and the version directory is gone, notebook or no notebook.



Corn

Which brings me to the open end of this. The strongest carve-outs we found are the ones the platform enforces — the data directory contract, the strict manifest. The override file survives on convention and discipline. What does that mean for the long tail of self-hosted tools that will never have a platform behind them? Daniel's Canopy is exactly that — no vendor is going to ship him a lifecycle contract. He builds the seam himself, or borrows one Docker already maintains.



Herman

The pattern that keeps winning across generations is the one where the base declares nothing in the private directory. nginx's conf dot d outlived plenty of fancier config systems for exactly that reason — no collision is possible when the public layer never claims the space. If I were generalizing one lesson from today, it's that one, and it's available to anybody with a glob.



Corn

It's the oldest lesson in sharing a house. You don't negotiate over the shelf nobody uses.



Herman

And Daniel's original observation closes the loop — public by default is operationally cheaper for the author. The carve-out is what makes that cheapness safe. The seam is the product, really. Without a durable private layer, public-by-default just means your private decisions leak into the artifact.



Corn

Somewhere out there is a home server owner discovering his voice assistant ended up in the public repo.



Herman

During the show notes, by the way — Hilbert Flumingtop produces the show, and has for years, and the notebook habit explains the show notes formatting more than I expected it to.



Corn

If you've enjoyed this one, a review wherever you listen helps other people find us — takes a minute, and it makes a difference to a small show.



Herman

If you want more of this, try episode five, Fine-Tuning ASR For Maximal Usability; and episode twenty, Architectural AI. This has been My Weird Prompts. We'll be back soon.



Corn

See you then.

SOURCES

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

1. **Docker Docs, Merge Compose files, retrieved 2026-11-21 [primary]**
   https://docs.docker.com/compose/how-tos/multiple-compose-files/merge/

2. **Docker Docs, Include, retrieved 2026-11-21 [primary]**
   https://docs.docker.com/compose/how-tos/multiple-compose-files/include/

3. **Docker Docs, Extend your Compose file, retrieved 2026-11-21 [primary]**
   https://docs.docker.com/compose/how-tos/multiple-compose-files/extends/

4. **Docker Compose CLI reference (via Context7, library `/docker/compose`, versions v2.39.1 / v5.5.1 / v2.34.0, updated 2026-11-21) [primary]**
   https://github.com/docker/compose/blob/main/docs/reference/compose.md

5. **Claude Code Docs, Plugin manifest reference, retrieved 2026-11-21 [primary]**
   https://code.claude.com/docs/en/plugins/manifest-reference

6. **Claude Code Docs, Plugin loading reference, retrieved 2026-11-21 [primary]**
   https://code.claude.com/docs/en/plugins/loading

7. **Claude Code Docs, Marketplace reference, retrieved 2026-11-21 [primary]**
   https://code.claude.com/docs/en/plugins/marketplace-reference

8. **Claude Code Docs, Create a marketplace, retrieved 2026-11-21 [primary]**
   https://code.claude.com/docs/en/plugins/create-marketplace

9. **Docker-compose copy/add for single files/directories, retrieved 2026-11-21**
   https://github.com/docker/compose/issues/6643

10. **Thibaut Voirand, Setting up a web server with nginx and Docker Compose, 2025, retrieved 2026-11-21**
    https://thibautvoirand.com/blog/2025/nginx-compose-webserver/

11. **public mirror of Claude Code plugin directory utilities, retrieved 2026-11-21**
    https://github.com/leaf-kit/claude-analysis/blob/main/src/utils/plugins/pluginDirectories.ts

12. **Extension systems: Claude Code research notes, retrieved 2026-11-21**
https://github.com/orchidautomation/pluxx/blob/main/research/extension-systems/01-claude-code.md

13. **plugin-creator reference mirror, retrieved 2026-11-21**
https://github.com/krzysztofsurdy/code-virtuoso/blob/main/skills/tools/plugin-creator/references/plugin-manifest.md

14. **Claude Code Plugin Manifest in 2026: Full Reference, 2026, retrieved 2026-11-21**
https://skiln.co/blog/claude-code-plugin-manifest-reference-2026

15. **plugins-management reference, retrieved 2026-11-21**
https://github.com/codealive-ai/ai-driven-development/blob/main/skills/plugins-management/references/plugin-guide.md