willibaldlenthin via RSS

Reading List - 2026-01-30

Jan 30, 2026

Table of Contents

1. How to Choose Colors for Your CLI Applications · Luna’s Blog
2. A Protocol for Package Management
3. No management needed: anti-patterns in early-stage engineering teams
4. Long time ago, I was looking for game with some hidden rules, browsing random wi...
5. Keychron’s Nape Pro turns your mechanical keyboard into a laptop-style trackball rig: Hands-on at CES 2026
6. The Code-Only Agent
7. Agent-native Architectures: How to Build Apps After Code Ends
8. jj tug | Shaddy's
9. Don't fall into the anti-AI hype

How to Choose Colors for Your CLI Applications · Luna’s Blog

Source: https://blog.xoria.org/terminal-colors/

Let’s say you’re creating a CLI tool which has to display syntax highlighted source code. You begin by choosing some colors which look nice with your chosen terminal theme:

```
~ — zsh — Sorcerer — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
```

Nice! However, who knows if it’ll still look good for people who use a theme different to yours? It seems sensible to try out the defaults, at least. Let’s start with the macOS Terminal.app default theme:

```
~ — zsh — Basic — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
~ — zsh — Basic — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
```

Youch! It seems fair to try the Tango themes next, since those are the default on e.g. Ubuntu:

```
~ — zsh — Tango Light — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
~ — zsh — Tango Dark — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
```

Hmm, better, but not by much. Finally, let’s try what is likely the most popular custom terminal theme – Solarized:

```
~ — zsh — Solarized Light — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
~ — zsh — Solarized Dark — 51×11
% highlight foo # just some docs func HelloWorld() [12]u8 { return "hello world\n" } Finished highlighting in 0.02 seconds. % █
```

Well then … Let’s take a look at each palette and investigate.

Sorcerer

```
~ — zsh — Sorcerer — 51×11
% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █
```

In Sorcerer, all colors are readable on the default background except for `black`, which is in fact darker than the background. This is useful as the background color for status bars and the like. `white` is the same color as the default foreground, and `brblack` is a nice faded color. Additionally, `brwhite` is even lighter than the foreground; this allows for subtle emphasization of important text like error messages and titles.

Basic

```
~ — zsh — Basic — 51×11
% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █
~ — zsh — Basic — 51×11
```

% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

The Basic themes are, well, *horrendous*. Really owning that 90s xterm look, it seems. `bryellow` is unreadable in light mode (check out that function name from the code sample earlier), while in dark mode both `blue` and `brblue` are totally illegible.

That leaves us with thirteen colors we can safely use:

~ — zsh — Sorcerer — 51×11

% colortest --only-usable █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

Tango

~ — zsh — Tango Light — 51×11

% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

~ — zsh — Tango Dark — 51×11

% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

In my opinion these did a lot better than Terminal.app's Basic themes, but they are still far from perfect. `bryellow` is again unreadable in the light theme, and perhaps `brgreen` is a little difficult to see, though it's nothing that would stop me from using `brgreen` in an application.

At this point you may have noticed how the greyscales – `black`, `brblack`, `white` & `brwhite` – have remained consistent between light and dark themes for both Basic and Tango. Of course, this means that `{,br}white` is unreadable in Tango Light (owing to the light background) and `black` is unreadable in Tango Dark (owing to the dark background).

In other words: forget about that idea of mine from earlier about using `brwhite` to emphasize content. Unless, of course, you don't mind if your eminently *emphasized* words are completely unreadable for the user of your software who deigns to use the default light theme of A Popular Linux Distro.

On the other hand, using `brblack` to de-emphasize content still seems fine to me. I suppose some extra contrast for `brblack` in Tango Dark would be nice, but with text which is meant to be ignored I don't think this matters much.

And lo, but ten colors remain.

~ — zsh — Sorcerer — 51×11

% colortest --only-usable █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

Solarized

~ — zsh — Solarized Light — 51×11

% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

~ — zsh — Solarized Light — 51×11

% colortest █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █

Solarized is a curious beast. Every color in it was chosen using _L*a*b*_ → https://en.wikipedia.org/wiki/CIELAB_color_space, a perceptually-uniform color space from the 1970s. (For what it's worth, color science has progressed significantly → https://bottosson.github.io/posts/oklab/ since then; the only reason Ethan Schoonover used _L*a*b*_ is that it's commonly used in photography, and he used to be a professional photographer.)

Its lightnesses are perfectly symmetrical so that Solarized Light and Dark can share a set of accent colors while maintaining identical contrast. Moreover, the warm tones of the light theme and cool tones of the dark theme are complementary. (The hue gap is closer to 150° than 180° in reality. See here → https://bottosson.github.io/misc/colorpicker/#002b36 and here → https://bottosson.github.io/misc/colorpicker/#fdf6e3 to compare hue values.)

Solarized is also incredibly popular. I have no data here, but as of the date of writing it’s the most starred theme repository on GitHub I can find. Solarized has 15.4 thousand stars at the moment, while the next-closest is Gruvbox → https://github.com/morhetz/gruvbox with 11.8 thousand. Solarized is available as a plugin or sometimes even as a built-in preset in damn near every popular terminal emulator and editor on the planet.

To understand Solarized’s peculiar arrangement of the 16-color palette, we have to travel back in time to 2011 when Solarized was first released → https://github.com/altercation/solarized/commit/3da9bd10d3b8c1ad6e2a5ab8617ef8c82fca0df7. In this dark era, terminals supporting 24-bit color didn’t exist / weren’t widespread. One option common among Vim themes at the time was to round every color to the nearest 256-color palette value. In Solarized’s case, this destroys the mathematical symmetry at the heart of the theme. (I’m not kidding, it looks awful → https://github.com/lifepillar/vim-solarized8#but-my-terminal-has-only-256-colors.)

The solution – rather, *hack* – chosen at the time was to distill all the colors used in the Vim interface down to a palette of sixteen colors. Conveniently, Solarized’s accent colors fit nicely into the non-bright column of the 16-color palette, while Solarized’s monotones fit into the bright column. Once the user sets their terminal to use the Solarized palette, Vim can color its entire interface using only the 16-color palette and get correct color values, no clunky color approximations needed.

The downside to all this is that an application which uses any of the bright colors which Solarized co-opted for itself will look strange. Users of Solarized – and, by god, there’s so many of them – appear → https://github.com/gradle/gradle/issues/2417 frequently → https://github.com/gruntjs/grunt/issues/181 on → https://github.com/crate-ci/cargo-release/issues/41 issue → https://github.com/cli/cli/issues/1743 trackers → https://github.com/mintty/mintty/issues/683 asking why command-line output is inexplicably gray or even invisible as a result of CLIs using these forsaken bright colors.

Our beloved `brblack` is unreadable in Solarized Dark, so we’ll have to strike it from the table in addition to the affected bright colors.

```
~ — zsh — Sorcerer — 51×11
% colortest --only-usable █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite % █
```

A sad note about bold

Far back in the past, there was no way for terminals to display bright colors. As a workaround, manufacturers (we’re talking about physical terminals here) started making all bold text bright instead of using a heavier font weight → https://en.wikipedia.org/wiki/ANSI_escape_code#3-bit_and_4-bit. One way or another this ended up in the default settings of many modern terminal emulators (in spite of not being in the standard), meaning that regular colorful text made bold can become bright too, depending on the user’s configuration.

Conclusion

And so, I present to you the final version of our table of acceptable colors:

```
~ — zsh — Sorcerer — 51×21
% colortest --only-usable --bold Regular: █ black █ brblack █ red █ brred █ green █ brgreen █ yellow █ bryellow █ blue █ brblue █ magenta █ brmagenta █ cyan █ brcyan █ white █ brwhite
Bold: █ boldblack █ boldbrblack █ boldred █ boldbrred █ boldgreen █ boldbrgreen █ boldyellow █ boldbryellow █ boldblue █ boldbrblue █ boldmagenta █ boldbrmagenta █ boldcyan █ boldbrcyan █ boldwhite █ boldbrwhite % █
```

Only eleven out of our thirty-two possible color settings are permissible, given that we want applications to remain readable for as many people as we can.

If you’re developing a command-line tool which will be used by anyone apart from yourself, I strongly recommend you limit your use of color to the ones I’ve identified here as being “mostly alright” and “not unreadable in a common configuration used by tons of people”.

Appendix

You probably didn’t notice, but I styled the “terminal windows” in this post to look as similar as possible to macOS Terminal.app windows through painstaking color picking and pixel counting.

The dimensions in each window’s titlebar matches as closely as I can with its actual dimensions on-screen.

The `colortest` and `highlight` utilities are entirely fictional.

Terminal.app doesn’t actually provide individual access to the light and dark variants of Basic; they appear as a single theme, which switches seamlessly when the OS theme changes. As far as I know, this reactive functionality isn’t exposed to any other theme, whether pre-installed or user-created. In order to capture this, I made the terminal windows in this post react to whether the rest of the site is in light or dark mode, *except for the Basic windows*. They remain fixed in either light or dark mode, since in real life you’ll never see, for example, a light Basic terminal with dark window chrome.

Luna Razzaghipour
29 January 2023

A Protocol for Package Management

By Andrew Nesbitt
Published Jan 22, 2026, 10:00 AM
Source: https://nesbitt.io/2026/01/22/a-protocol-for-package-management.html

Writing about testing package managers like Jepsen tests databases → https://nesbitt.io/2026/01/19/a-jepsen-test-for-package-managers.html got me thinking about what sits underneath all the ecosystem-specific details. We can describe HTTP without talking about Apache or nginx. We can discuss database consistency models without reference to PostgreSQL or MySQL. But when we talk about package management, the conversation immediately becomes about npm's node_modules hoisting or Cargo's semver-compatible version deduplication or Go's minimal version selection, rather than the underlying operations those are all implementations of.

Individual package managers have specifications. Cargo documents its resolver. npm documents its registry API. RubyGems has the compact index spec → https://nesbitt.io/2025/12/28/the-compact-index.html. But there's no shared language for talking about what package managers do in the abstract, independent of any particular implementation.

I've written about package manager components → https://nesbitt.io/2025/12/02/what-is-a-package-manager.html, tradeoffs → https://nesbitt.io/2025/12/05/package-manager-tradeoffs.html, terminology → https://nesbitt.io/2026/01/13/package-manager-glossary.html, and categorization → https://nesbitt.io/2025/12/29/categorizing-package-manager-clients.html. This post tries to go one level higher: what would a reference model for package management look like? Something that names the layers, actors, operations, and properties that all package managers share, even when their implementations differ.

If this existed, it might look like: a document that defines "resolution determinism" precisely enough that you could compare npm and Cargo on the same terms. A taxonomy of failure modes that tool builders could implement against. A vocabulary for governance operations that lets researchers compare how different registries handle disputes. A shared frame that makes the similarities and differences legible, without forcing convergence.

I should say upfront that I have no experience writing or defining protocols or reference models. I don't know what's actually involved in that work, or what the right process would be. What follows is more a sketch of what such a model might need to cover, not a proposal for how to build one. I'm throwing out ideas to see if they resonate with people who do know this stuff, more conversation starter than spec draft.

Most of what follows focuses on language package managers (npm, pip, Cargo, Bundler) rather than system package managers (apt, dnf, pacman, Homebrew). The two categories overlap but have different concerns. System package managers deal with file conflicts, coordinated releases, maintainer curation, and post-install scripts running as root. Language package managers deal with per-project isolation, transitive dependency graphs, and lockfile reproducibility. A complete reference model would need to cover both, but I know the language side better.

The layers

User commands. What developers type at the terminal. `install`, `add`, `remove`, `update`, `audit`, `publish`. These are the interface layer, and despite different names across ecosystems (`npm install` vs `pip install` vs `cargo add`), they map to a small set of underlying operations. A protocol could define what each command is expected to do without specifying the CLI syntax.

Manifest format. How projects declare their dependencies. package.json, Gemfile, Cargo.toml, pyproject.toml. Each uses different syntax (JSON, Ruby DSL, TOML, TOML again) but expresses similar concepts: package name, version constraints, dependency types, metadata. The glossary → https://nesbitt.io/2026/01/13/package-manager-glossary.html covers the terminology; a protocol would specify the semantic model underneath.

Lockfile format. How resolved dependencies get recorded. Lockfiles vary wildly → https://nesbitt.io/2026/01/17/lockfile-format-design-and-tradeoffs.html in what they include (just versions? checksums? full URLs? resolver metadata?) and how they structure it. But they all serve the same purpose: making resolution reproducible. A protocol could specify what a lockfile must capture without dictating the serialization format.

Registry protocol. How clients talk to registries. REST APIs, sparse indexes, full replication, proprietary protocols. This is where the compact index → https://nesbitt.io/2025/12/28/the-compact-index.html lives, and where decisions about caching, consistency, and availability get made. Different data flow patterns have different tradeoffs.

Archive format. How packages get bundled for distribution. Tarballs, wheels, jars, crates. Some include metadata inside the archive; others serve metadata separately. Some are source distributions; others are prebuilt binaries. The format determines what's possible at install time.

Dependency resolution. The algorithmic core of package management. Given a manifest with version constraints, produce a concrete dependency graph that satisfies all of them. This is NP-complete in the general case → https://nesbitt.io/2025/12/29/categorizing-package-manager-clients.html, so every resolver makes tradeoffs. SAT solvers → https://nesbitt.io/2025/12/29/categorizing-package-manager-clients.html#resolution-algorithms can prove unsatisfiability but are expensive. Backtracking is simpler but can be slow. Minimal version selection → https://research.swtch.com/vgo-mvs sidesteps complexity by always picking the oldest version. PubGrub → https://nex3.medium.com/pubgrub-2fb6470504f gives better error messages by tracking why versions were excluded. A protocol would need to specify what resolution means without mandating a particular algorithm.

Publishing workflow. How packages go from a developer's machine to a registry. Authentication, validation, signing, propagation. Trusted publishing → https://docs.pypi.org/trusted-publishers/ from CI systems is changing this layer. The protocol would need to cover both the mechanics and the security properties.

Security model. What guarantees the system provides and what threats it addresses. Client-side concerns (verifying checksums, validating signatures, detecting tampering) differ from registry-side concerns (authenticating publishers, scanning for malware, enforcing policies). The landscape → https://nesbitt.io/2026/01/03/the-package-management-landscape.html shows how many tools exist around these concerns.

The actors

Publishers create packages and make them available. They have identities (accounts, keys, or domain ownership) and permissions to write to certain namespaces.

Consumers install packages into projects. They specify what they need (through manifests with version constraints) and receive resolved dependency graphs. Automated consumers (CI systems, dependency update tools, AI coding assistants) now account for a significant share of registry traffic, with different access patterns than human developers.

Registries store packages and serve metadata. They map names to artifacts, enforce namespace rules, and maintain the index that makes resolution possible.

Proxies and mirrors sit between consumers and registries. They cache, filter, and sometimes transform what passes through. Organizations run them for reliability, speed, or policy enforcement.

Resolvers turn abstract constraints into concrete versions. They might run on the client, on a server, or both. They need access to package metadata and produce deterministic (or at least reproducible) results.

These roles can be combined. A consumer might also be a publisher. A registry might include a resolver. But the functions are distinct even when the implementations merge them.

The interesting cases are the interactions between actors. A proxy caches package metadata, but what happens when the upstream registry yanks a version? The proxy might keep serving the yanked version to clients that haven't refreshed their cache. Is that a bug or a feature? It depends on whether you prioritize reproducibility (the cached version still works) or security (the version was yanked for a reason).

Resolvers can run on the client or on the registry. Client-side resolution means you control the algorithm but need to fetch metadata for every candidate version. Server-side resolution means fewer round trips but you're trusting the registry to resolve correctly. Some registries offer both. When they disagree, which one is authoritative?

Mirrors introduce another layer. An organizational mirror might lag behind upstream by hours or days. A developer resolves against the mirror, gets version 1.2.3, commits a lockfile. A CI server resolves against upstream, sees that 1.2.3 was yanked, fails the build. Whose view of the world is correct? The protocol would need to define how staleness propagates through the system.

The data types

Package identifier: a name in a namespace. `express` in npm, `rails` in RubyGems, `github.com/gin-gonic/gin` in Go. PURL → https://github.com/package-url/purl-spec already standardizes how to reference packages across ecosystems. The identifier space is the registry's most valuable asset and its hardest governance problem.

Version: an identifier for a specific release. Usually follows some structure (semver, calver, or various other schemes → https://nesbitt.io/2024/06/24/from-zerover-to-semver-a-comprehensive-list-of-versioning-schemes-in-open-source.html). Versions are partially ordered, and that ordering is load-bearing for resolution.

Version constraint: a predicate over versions. `>=1.0.0`, `^2.3.4`, `~> 1.5`. The syntax varies wildly but the semantics are similar: given a set of available versions, which ones satisfy this constraint? VERS → https://github.com/package-url/purl-spec/blob/master/VERSION-RANGE-SPEC.rst attempts to standardize this.

Manifest: a declaration of what a project needs. Direct dependencies with constraints, plus metadata about the project itself.

Lockfile: a record of what was actually resolved. Concrete versions for every dependency (direct and transitive), often with integrity hashes.

Package artifact: the distributable unit. A tarball, wheel, jar, or crate. Contains code plus metadata in some structured format.

Dependency graph: the result of resolution. A directed acyclic graph where nodes are package-versions and edges are “depends on” relationships.

Platform target: the combination of operating system, CPU architecture, and runtime version that a binary artifact is built for. Python wheels encode this in filenames (`cp311-manylinux_x86_64`). Rust has target triples (`x86_64-unknown-linux-gnu`). System packages are built per-distro and architecture. Resolution needs to filter artifacts by what the consumer’s environment can actually run, and the ways ecosystems represent this vary widely.

The operations

Publish: a publisher submits an artifact and metadata to a registry. The registry validates the submission (name ownership, version uniqueness, format compliance) and makes it available for resolution.

Resolve: given a manifest, produce a dependency graph that satisfies all constraints. This is the hard part, NP-complete in the general case → https://nesbitt.io/2025/12/29/categorizing-package-manager-clients.html, and where most of the interesting design decisions live. Different resolution algorithms → https://nesbitt.io/2025/12/29/categorizing-package-manager-clients.html#resolution-algorithms (SAT solving, backtracking, minimal version selection) make different tradeoffs.

Install: given a resolved graph (from a lockfile or fresh resolution), fetch artifacts and place them where the runtime can find them.

Update: given an existing lockfile, resolve again with newer constraints or newer available versions, producing a new lockfile.

Yank/deprecate: a publisher marks a version as unavailable for new resolution while keeping it accessible for existing lockfiles.

Query: ask the registry for metadata about a package, its versions, its dependencies, or its dependents.

Governance operations

Registries don’t just host files. They make political decisions → https://nesbitt.io/2025/12/22/package-registries-are-governance-as-a-service.html about who owns names, how disputes resolve, and what gets removed. These governance operations are as much a part of what package managers do as resolution or installation, but they’re rarely described in compatible terms.

Namespace allocation. Who can claim a name? First-come-first-served? Domain verification? Organizational scopes? Different registries make different choices, but the underlying question is the same: how does an identifier get bound to an owner?

Ownership transfer. What happens when a maintainer abandons a package, or dies, or has their account compromised? npm has a process. RubyGems has a different process. The concept of “transferring ownership” is universal; the policies vary.

Dispute resolution. Two parties claim the same name. A trademark holder wants a package removed. A maintainer claims their account was hijacked. How do these get resolved? By whom? With what appeals process?

Content removal. Malware gets found, or a package violates terms of service, or a court orders a takedown. What gets removed? Who decides? Is it reversible? How fast does it propagate to mirrors?

Account recovery. A maintainer loses access. How do they prove identity? What happens to their packages during the recovery process? Who has authority to restore access?

A protocol could define shared vocabulary for these governance operations without mandating specific policies. “Ownership transfer” could have a common definition even if npm and PyPI have different rules about when it’s allowed. This would let researchers compare governance models across registries, and might help smaller registries learn from decisions larger ones have already worked through.

The consistency properties

What guarantees should these operations provide? Naming these properties explicitly is part of what a reference model would do.

Resolution determinism: given the same manifest and the same registry state, resolution should produce the same graph. “Same registry state” is doing a lot of work here, since registries are distributed systems with CDN caching and eventual consistency.

Lockfile integrity: installing from a lockfile should produce identical results regardless of when or where you run it, as long as the referenced artifacts still exist.

Publish atomicity: when a version is published, it should become visible atomically. Consumers shouldn’t see partial states where the metadata exists but the artifact doesn’t, or vice versa.

Monotonic versions: once a version number is used, its meaning shouldn’t change. Republishing the same version with different contents violates this, which is why most registries forbid it.

Yank semantics: yanked versions should be excluded from new resolution but included when resolving existing lockfiles that reference them.

Version ordering: given two versions, which one is newer? This sounds obvious until you hit pre-releases. Does `^1.0.0` match `2.0.0-alpha`? npm says no by default. What’s the “latest” version of a package? Most registries exclude pre-releases from `latest` unless there’s no stable release, but the rules vary. Sorting versions correctly matters for resolution and for tools that display changelogs or upgrade paths.

Each of these has subtleties that different package managers handle differently. Go’s minimal version selection → https://research.swtch.com/vgo-mvs achieves determinism without lockfiles by always picking the oldest satisfying version. npm’s resolution used to produce different trees depending on installation order. The compact index → https://nesbitt.io/2025/12/28/the-compact-index.html that Bundler and Cargo use is append-only specifically to provide consistency guarantees that the old full-index approach couldn’t.

The edge cases are where things get interesting. “Same registry state” sounds simple until you consider that npm’s registry sits behind Fastly’s CDN with eventually consistent replicas. Publish atomicity involves writing to multiple stores that can partially fail. Yank semantics interact with caching in subtle ways when a lockfile references a version that’s been yanked since last resolution. Different clients handle these cases differently.

Missing concepts

Time. I’ve talked about staleness and caching, but there’s no explicit notion of time in the model. Distributed systems specs usually need concepts like epochs, snapshots, or logical clocks. When did a publish become visible? What’s the maximum propagation delay? Can a lockfile reference a point-in-time view of the registry? Package managers don’t typically expose these concepts, but they’re operating under temporal assumptions that users don’t see.

Authority. Who is authoritative for what? The registry is authoritative for package metadata, but what about a proxy that’s been caching for six months? The lockfile is authoritative for what versions to install, but what if the registry says one of those versions is now malware? Clients trust registries, registries trust publishers, publishers trust CI systems. A reference model would need to map these trust boundaries and say what happens when authorities disagree.

Observability. What can you see when things go wrong? Logs, error messages, introspection APIs. If resolution fails, what information is available to debug it? If a publish doesn’t propagate, how do you know? Tool builders and researchers need to observe what package managers are doing, but observability isn’t typically part of the specification. It’s an implementation detail that varies widely and matters a lot.

Data flow patterns

Full replication: the client downloads the complete index and resolves locally. apt does this. Resolution is fast once synced, works offline, but initial sync is expensive and data goes stale.

On-demand queries: the client fetches metadata per-package during resolution. npm and PyPI work this way. Always current, but requires network access and many round trips.

Sparse indexing: the client fetches only metadata for packages it actually needs, but in a cacheable format. Cargo's sparse index → https://blog.rust-lang.org/2023/03/09/Cargo-1.68.0.html#sparse-registry-support and RubyGems' compact index → https://nesbitt.io/2025/12/28/the-compact-index.html use this approach.

Proxy caching: an organizational proxy intercepts requests and caches responses. Reduces load on upstream registries and provides availability if upstream goes down.

Each pattern makes different tradeoffs between freshness, bandwidth, latency, and offline capability. A protocol spec would need to accommodate all of them.

Failure modes

A protocol needs to specify not just what happens when things work, but what happens when they don't. This is where ecosystems diverge most and where shared vocabulary would help most.

Checksum mismatch. Clients vary: fail hard, warn, or skip verification entirely.

Unavailable dependency. Do you fail immediately? Try mirrors? Fall back to cache? npm, pip, and Cargo all handle this differently.

Unsatisfiable resolution. PubGrub tracks the chain of conflicts. Other resolvers provide less detail.

Partial install. npm leaves partial installs in place. Some tools use atomic installs.

Cascading failures. A transitive dependency four levels deep becomes unavailable. Some ecosystems fail fast; others degrade gracefully.

These failure modes matter because they're where the user experience diverges most. Shared vocabulary for failure conditions would help both tool builders and users.

What this enables

Portable security research. Dependency confusion → https://nesbitt.io/2025/12/10/slopsquatting-meets-dependency-confusion.html was discovered in npm, then checked in PyPI and RubyGems. Typosquatting → https://nesbitt.io/2025/12/17/typosquatting-in-package-managers.html techniques transfer between ecosystems, but defenses don't always follow. A protocol would let researchers describe attacks and defenses in terms that apply everywhere.

Systematic comparison. Comparing npm and Yarn on consistency guarantees is hard because they don't describe those guarantees in compatible terms. A shared vocabulary would let us ask: which package managers provide publish atomicity? Which guarantee resolution determinism? Where does each fall on the tradeoff space → https://nesbitt.io/2025/12/05/package-manager-tradeoffs.html?

Learning from each other. When Cargo adopted sparse indexes → https://blog.rust-lang.org/2023/03/09/Cargo-1.68.0.html#sparse-registry-support, they were borrowing an idea RubyGems had proven out years earlier → https://nesbitt.io/2025/12/28/the-compact-index.html. When pip rewrote its resolver, they borrowed test cases from Ruby and Swift → https://pradyunsg.me/blog/2020/03/27/pip-resolver-testing/. A shared model would make these patterns more visible.

Support for smaller ecosystems. Dependabot prioritizes npm, PyPI, Maven, Go because each integration is significant work. Smaller package managers (Nimble, Shards, jpm) get pushed to the back of the queue. If tools could implement against a protocol and write thin adapters, the long tail might actually get tooling support.

What this isn't

Yes, I've seen the xkcd → https://xkcd.com/927/. This isn't a proposal to standardize package managers. Existing ecosystems have differences that matter. Some take advantage of runtime features that can't be replicated everywhere: Bundler's Gemfile is a Ruby DSL, Mix is deeply integrated with OTP. A protocol has to sit above these language-specific capabilities, which means it can't capture everything.

It's also not something existing package managers would easily adopt. npm's registry isn't going to change its API to match a theoretical spec. The value is shared vocabulary for reasoning about these systems.

PURL → https://github.com/package-url/purl-spec already does this for identifiers. It has problems with edge cases and ecosystems that don't fit its model, but it's proven useful enough that tools adopted it anyway. A protocol for the rest of package management would have similar imperfections and similar utility.

Why this is hard

Abstracting over ecosystem differences is hard. A "version constraint" in npm might match pre-releases differently than in Cargo. The glossary → https://nesbitt.io/2026/01/13/package-manager-glossary.html can define these terms, but a protocol would need to handle edge cases where ecosystems diverge.

Some registries are closed source, which means understanding their behavior requires black-box observation rather than reading the code. Others like crates.io, RubyGems.org, and PyPI are open source.

There's a coordination problem: who maintains the spec? And an adoption problem: the vocabulary only has value if people use it. PURL succeeded because SBOMs needed a standard way to identify packages. A protocol would need a similar forcing function.

The package management landscape → https://nesbitt.io/2026/01/03/the-package-management-landscape.html suggests demand exists. Syft → https://github.com/anchore/syft, Dependabot → https://github.com/dependabot/dependabot-core, deps.dev → https://deps.dev/, bibliothecary → https://github.com/librariesio/bibliothecary, osv-scalibr → https://github.com/google/osv-scalibr each build their own abstraction layer over multiple ecosystems. The fact that they all independently arrived at similar abstractions suggests those abstractions want to exist.

Some cross-ecosystem infrastructure exists: PURL → https://github.com/package-url/purl-spec for identifiers, VERS → https://github.com/package-url/purl-spec/blob/master/VERSION-RANGE-SPEC.rst for version constraints, SPDX → https://spdx.dev//CycloneDX → https://cyclonedx.org/ for SBOMs, Sigstore → https://www.sigstore.dev//SLSA → https://slsa.dev//TUF → https://theupdateframework.io/ for signing. None reach the protocol level: they specify interchange formats, not resolution semantics or publish consistency. There's also academic work → https://nesbitt.io/2025/11/13/package-management-papers.html (Di Cosmo's formal models → https://www.researchgate.net/publication/278629134_EDOS_deliverable_WP2-D21_Report_on_Formal_Management_of_Software_Dependencies, Russ Cox's Surviving Software Dependencies → https://dl.acm.org/doi/10.1145/3329781.3344149) that hasn't coalesced into something practitioners use.

Building a protocol would mean documenting what existing package managers actually do in practice, including the edge cases. The spec would emerge from implementations.

If you work on cross-ecosystem tooling, registry infrastructure, or dependency research, I'd like to know whether this gap feels real to you too.

No management needed: anti-patterns in early-stage engineering teams

By Antoine Boulanger
Published Jan 10, 2026, 12:00 AM
Source: https://www.ablg.io/blog/no-management-needed

January 10, 2026
This article is for early-stage (Seed, Series A) founders who think they have engineering management problems (*building eng teams, motivating and performance-managing engineers, structuring work/projects, prioritizing, shipping on time*).

The gist: if you *think* you have these problems, it is likely that the correct solution is to **do nothing, to not manage, and to go back to building product and talking to users**. Put another way, and having managed teams at all scales, I don't think it's a good use of your time as a founder to be "managing" engineers at such an early stage.

In the following sections, I'll go through the most typical anti-patterns I've seen, and try to highlight a better use of your time if you think you've hit the situation in question.

Do not try to "motivate" your engineers

A common concern of many founders is making sure that their engineers are working hard. This could mean putting in long hours, working more than competitors, completing heroic codebase rewrites, etc. When these external *signs of effort* seem to be missing, founders worry that the team is not "motivated", and it can be very tempting to treat symptoms over causes. For example:

- creating cultural norms around putting in long hours (996-style culture) by either requiring or celebrating them
- scheduling recurring or non-urgent meetings on weekends (e.g. standup on Saturdays)
- micro-managing tasks, or asking people for status reports and other evidence they worked hard

These anti-patterns share one thing in common: they start with founders trying to actively *do something* to motivate the team. This has 2 consequences:

1. This can cause the very engineers you want to retain (those who have many options) to self-select out of your engineering culture. I know several top 1% engineers in the Valley who disengage from recruiting processes when 996 or something similar is mentioned.
2. You are wasting your mental energy on the wrong problem

All of this is a long way of saying that **motivation is an inherent trait** of great startup engineers. Your only job is to hire these engineers, and then to maintain an environment where they want to do their best work. And yes, at that point, you may see them working long hours and doing heroic actions you did not even think were possible.

> Motivation is a hired trait. The only place where managers motivate people is in management books.

I'll dedicate a post to specific ways you can identify motivation during hiring, but in short, look for:

- the obvious one: evidence that they indeed exhibited these external signs of motivation (in an unforced way!) in past jobs
- signs of grit in their career and life paths (how did they respond to adversity, how have they put their past successes or reputation on the line for some new challenge)
- intellectual curiosity in the form of hobbies, nerdy interests that they can talk about with passion
- bias for action and fast decision speed

Finally, as a founder, you should definitely be the most motivated person, in an authentic way (maybe it's some piece of heroic coding, maybe it's taking 2am meetings with European customers, maybe it's something else unique to you). Cultivating your own inner motivation is the most effective way to set the tone for the team.

Do not hire managers too soon

The most obvious external sign that a startup has switched from building a product to building a company is to add management roles. When this switch happens prematurely, a lot of energy gets spent on stage-irrelevant problems.

By definition, an engineering manager needs to manage a team and projects, but if the team is still working on defining what they should be building, there is nothing to manage. Even the most intellectually honest manager will start outputting "management work", such as having 1:1s with everyone, doing some career coaching, applying order to the chaos of potential features by putting them in JIRA tickets or issues, etc. Here's what it means for you as a founder:

- you are still trying to find product-market fit and build your initial product
- an engineering manager is helping you do it in a more optimized way, but they are optimizing a moving target so it does not really improve anything
- you don't know if this engineering manager is bad at their job, or if the engineers are not performing, or if the product has no market anyway, or all of the above

So how do you define "too soon"? Let's look at a few typical inflection points, assuming at least one founder is technical:

The founding stage (5-6 engineers including founders)

Obviously too soon to hire managers or turn someone into a manager. The only management-like tasks for the founders are hiring and firing, other than that the team should largely be self-organizing and self-sustaining with lightweight tooling (a simple doc can even be used as a task tracker, 1:1s happen organically and are infrequent, etc.).

In general, the bias should be towards doing nothing in terms of management and everything in terms of hiring exceptional people who inherently work well together.

The multi-team stage (2 or 3 sub-teams of 5 engineers, 10-15 people total)

This might be late seed or series A, with an inkling of a working product. Many teams will decide to implement management at this stage, because it seems like the natural next step. The decision is full of nuances, but I would strongly advise to have all the engineers still report into a single person (ideally the co-founder CTO). Why? Speed of execution and culture, mainly:

- at 15 engineers, it is very doable for a single person to keep track of everyone's work and ensure alignment.
- this is the critical moment where you build the engineering culture that will bring you from here to hundreds of engineers (how do we hire, what do we value, how do we work together, etc.). It's much easier to do this as a flat team with a single leader.
- pivots and radical decisions could still happen frequently, which will be exponentially harder if you have to manage these engineers through 2 or 3 line managers.

The only nuance I would add, if you really need to start structuring the team, is to go with hybrid roles: maybe it's a very hands-on manager who still codes 70% of the time, maybe it's elevating a few key engineers into *informal* tech lead positions

The early growth stage (going from 20 to 50 engineers)

This is the sweet spot where the benefit of adding more management and more structure should outweigh the cost of letting the inevitable chaos of a larger team take a life of its own. Still, I would highly recommend a less-is-more approach.

Here are a few signs you've reached that stage:

- the CTO / whoever is managing everyone shows signs of burning out under the load
- adding more engineers no longer increases output, meaning you are constrained by team inefficiency
- the team excels at week-to-week impact, but nobody seems able to play out what will happen in 3 to 6 months

This is a vast topic, and I'll dedicate a future article to that specific stage, including how to hire your first head of engineering.

Do not copy Google

This section addresses two sides of the same coin, both related to the halo effect → https://en.wikipedia.org/wiki/Halo_effect surrounding great companies and more specifically their management practices:

- Applying management ideas that Google (or other successful company) have talked about and made popular
- Applying the *meta-idea* of innovating in the field of management (like Google did in their time)

I'll skip to the conclusion and explain it below:

> When in doubt, always pick the "node & postgres" stack of management. Do not innovate, keep it boring.

What I mean by the "node & postgres" of management

Node & postgres share these common traits: they have huge communities, their bugs and quirks have been explored by millions of people, and so they are great choices for early-stage startups compared to, say, C++ and OracleDB. No matter what you think about their technical merits, it would be very hard to point to them as a reason why a startup failed. They are just solid, boring tools, and they work at the early stage.

You should use the same type of boring, widely used, stage-appropriate tools when it comes to managing your startup. Every ounce of "innovation" you spend on your organizational structure, title philosophy, or new-age 1:1 is an ounce you aren't spending on your product. At the seed stage, your culture shouldn't be unique because of your clever peer feedback system, it should be unique because of the speed at which you solve customer problems.

What is the boring stack of seed stage management

As a conclusion to this section and to the entire article, I want to share, somewhat paradoxically, a few useful management activities specifically for the early stage. They almost all share the same "reluctant" approach to engineering management, which I think is a healthy leadership approach at that particular stage.

- **Hire inherently motivated people**: see first section
- **Don't manage around a hiring mistake**, let them go quickly and gracefully
- **Asynchronous status updates**: do not adopt all the "Scrum rituals" like standups, retros, etc. wholesale, and if you do, keep them asynchronous. There is little added value to a voiced update, even if it makes you feel good that people are indeed working hard and showing up to the standup on time!
- **An avoidant relationship to Slack**: while Slack is a given in today's distributed or hybrid teams, it can quickly become an attention destroyer, especially for engineers who need uninterrupted time to work. Keep it in check.
- **Organic 1:1s** (as opposed to recurring ones): keep them topic-heavy and ad-hoc, as opposed to relationship maintenance like in the corporate world.
- **Unstructured documents over systems of records**: unless you need to itemize tasks for audit purposes, a few notion or google docs can actually scale for 10-15 engineers, especially given current AI tools. They have very little overhead and are unbeatable in terms of flexibility.
- **Extreme transparency**: give everyone access to everything (customer call notes, investor updates, budgets, etc.). Not only will you build trust with the team, but you will also remove the need to "communicate" (as in, filtering and processing information), which is a typical management task.

To be clear, many of these practices do not scale past 20-25 engineers, but that's part of the point.

I hope you found this post actionable, good luck with building your team!

Long time ago, I was looking for game with some hidden rules, browsing random wi...

Source: https://news.ycombinator.com/item?id=46613095

Long time ago, I was looking for game with some hidden rules, browsing random wikipedia. I came across Mao [1]. It looked so cool, game that has it is culture.

I wanted to try, luckily using siblings is not considered war crime. Since I had read about it in wikipedia we did not have culture to base it on. It morphed to basically uno with normal playing card deck but winner gets to make new rule, any rule. They will enforce it but they will not tell it to anyone else, they will just comment: "you broke rules, take penalty"

Since we played it way too much with siblings, we had times where my brother took 15 card penalty on game start. There was ~4 day trip we played near 30h of Mao.

I still love it, but can't play it any more since people rarely have attention to detuct the hidden rules. But also I feel creatively blocked since I can't make super complex rules when playing with new people, and the magic between my siblings has dimished bit.

[1] https://en.wikipedia.org/wiki/Mao_(card_game)

> I still love it, but can't play it any more since people rarely have attention to detuct the hidden rules.

I have a theory you can only induct a new player 'properly' (i.e. without them getting out their phone and consulting wikipedia) when you've got at least 3-4 experienced players.

Fewer than that and the new player won't see enough plays to figure out what the pattern is before they're buried in penalty cards. I've found this to be true even if the new player is a veteran board game player, used to paying attention to long games with complicated rules.

I introduced this to so many people as the only one who knew how to play. Key is start the game with 3 cards and win the round quickly + be super diligent about the rules so no conflicting information. Win the round quickly, add an obvious rule and by the third round it's fair play.

Interesting theory. I haven't had change to try that kind a situation. The biggest game by people was like 6 people and 2 experts.

I can see that thou. I often had to give example rules for people, thou I feel like it robs part of the fun.

With more experts it could make things better, if they go easy on start. If they go full on with super hard rules, the half attention newbies would be lost.

Thou if the newbie really wants then they could learn in that big expert play too.

I had a friend who started the game saying, "The game of Mao is like the game of life. You come into it not knowing the rules but you get penalized for them anyway." I always liked that opening, more than "The only rule you may be told is this one."

Once we got really exploratory with my brother. We eliminated all the rules, expect empty hand wins.

Cards had no meaning, there was no turn nothing.

But that is bit too lose base.

From our exploration we saw that rules that say you must do something are waaay more fun ad clearer, than rules like you can or can choose not to do. The choise rules made it basically impossible to detuct and even with in the context felt like the other is possibly cheating.

We also saw that people were quite concervative and really really hesitant to alter the base game rules.

Thou if we played couple of games and then chatted about everyones rules, and then reseted the game, people would open up bit. But the base rules (of basically uno) were quite sacred to break for people.

On a ski trip with friends we spontaneously turned a game of Uno into a Mao drinking game.

The rules were: •• Picture cards worth 10pts, black cards 50pts, number cards = n points •• At game end, 2 players with most points drink •• +4 can stack on +2, and vice-versa if color is right •• Uno Uno doesn't win unless no cards can be stacked anymore •• No deck shuffling

This resulted in the most fun and long Uno games, as people would keep the risky + cards till the end to stack on the Uno Uno player and keep him in the game. The no-deck-shuffling added an element of card-counting to the game as the discarded cards would be added to the bottom of the deck when no cards were left to draw.

Love this game. It was especially popular during boy scout camping trips for me. I don't know if anyone would have the patience for it anymore, unless the internet went down for a week or something.

Mao took over my summer camp like wildfire in 2005. Fellow councilors and I would often consolidate the rules between camp sessions and we kept running clearing out playing cards from the Kenora grocery store to play on our island. Fond memories. It's definitely one of those games that suites camp life, with the culture of play and hidden rules.

One of the other comments talked about pizza box where they throw big coin/disc and on landing site you draw rule circle.

Your link seems bit like mix of Mao and it.

And it does seem to have "the commercial" version of Mao.

I should look into it more.

Do you have examples for good made up rules? I want to try it next time being with friends.

I used to play this a ton (often as a drinking game). The wiki does a good job of explaining the basics and I'd always do the no talking + point of order variants (+ a few others). My favorite type are:

1) rules requiring awareness of card order. "Have a nice day" is standard on 7, double 7's is "very nice" etc. Stack rules like that when combinations and/or cards of a suit are played and people have to remember 5 things they have to say or do after a card. Get's difficult over many rounds.

2) ice-breaking rules (if you're playing with new people). Friendly ones like "you must compliment _ when a _ is played." Great way to build/open someone up

3) rules changing play order. Aces reverse, add rules that e.g. skip a player and you'll have everyone waiting in suspense to see if the person who's turn it is actually knows its their turn. If not, "delay of game".

Play it with friends you'll be surprised with what people come up with!

Some newbies seem to go for rules like: It is illegal to take even number as punishment. Or You can only play sevens in pairs. Or King skips next one.

Games between my brothers and me, rules such as: Specific card plays from next player. Or You are allowed to play +- 1 instead of exact same.

Nasty ones like: Third red is illegal. You are not allowed to put card, that would make sum of last three to be more than 15.

Those are quite fun. But probably my all time favorite is: "Eights live backwards"

I could live it at that, but the deeper explanation is that it is illegal to keep eights in had same way to other cards and it must be played up-side down.

This rule is the one my brother broke when he was shuffling the deck and got thought about "fixing" the eights.

Since some rules can be really rare and some really common, we started to do in expert games that the rule maker can set the card penalty. Multiple times per play? Basic 2 card punishment. ~once per game? 3-4

Rarer than that? +5, go wild as is the name of the game.

With newbies, standard 2 card punishment keeps things simple.

When each person have 2-4 rules from them and to keep all the other hidden rules in mind, the fun is chaos and chaos is fun.

It is also cool to debrief after and discuss with people if anyone managed to guess the others rules.

Playing the 2 of clubs was a "total failure" and you would be handed the entire draw pile.

Instantly thought about Mao too, we'd play it with friends in-between classes in college. Good times. Making players say "thanks" after receiving a penalty (or keep receiving penalties) never got old.

Heh, since for us it was time with our dad, we needed to be able to chat. So before first test play we decided not make rules around speaking.

After that many people find it really hard to grasp the rules and possibilities. "How can the rules change without anybody knowing?" "How can they be enforced?" And so on.

To me the ideal would be no rules explained but as the embasidor, I do not get such possibility.

To others I explain some rules and example rules, such that we often want to sosialice so while you can make speaking rules they may be bit meh.

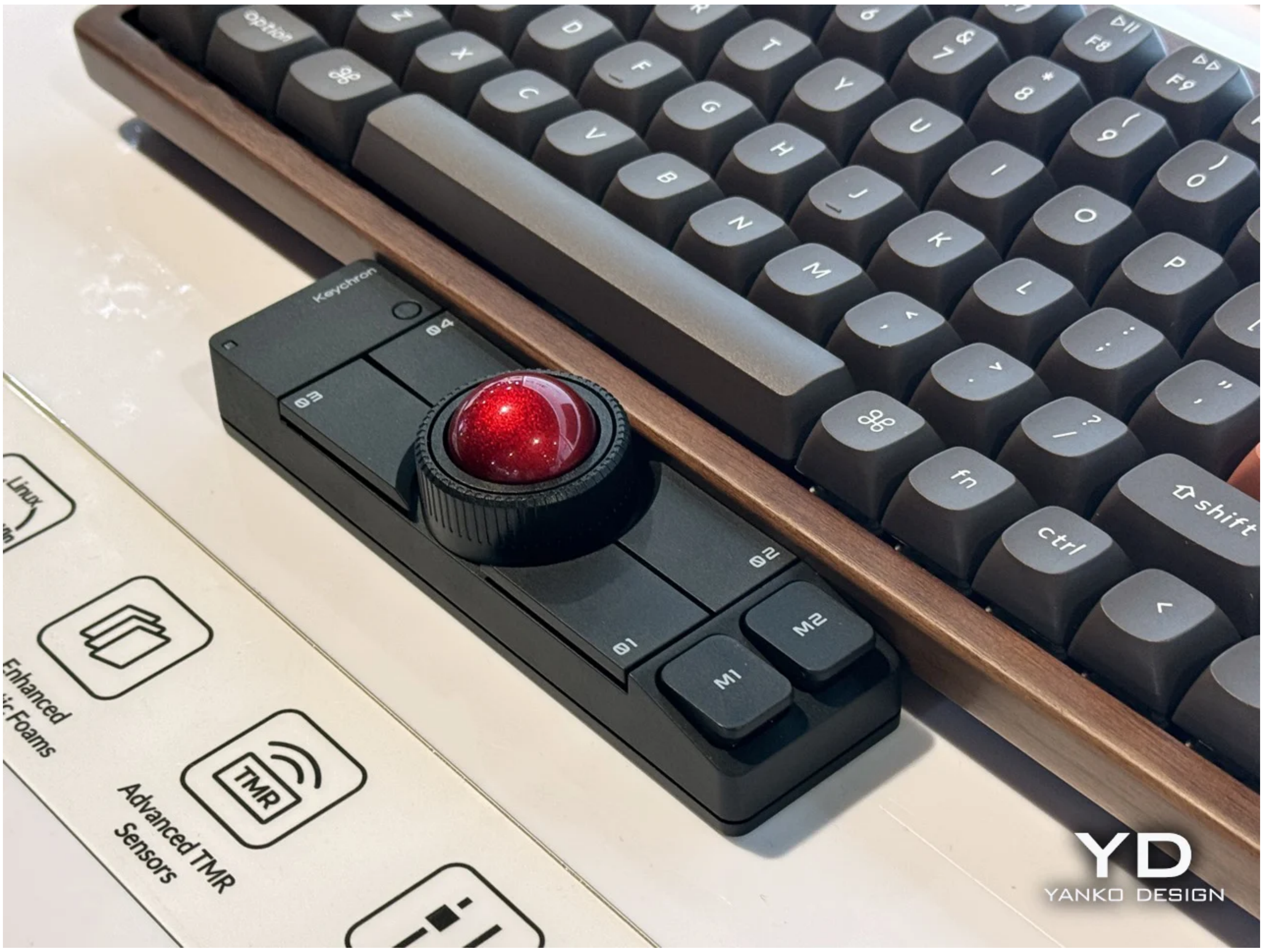
No one but one person made voice rules. My god I got burned in that game. I was constanly speaking as the explainer and keeping turn order up. I did not figure it why I got so many penalties from them, I had small feeling but could not figure the exact thing.

Turns out he was bit annoyed by me and the rule was: "you are not allowed to speak on your turn"

Obv I was not mad, I was amused when I got interesting game. Good times.

Keychron’s Nape Pro turns your mechanical keyboard into a laptop-style trackball rig: Hands-on at CES 2026

By Sarang Sheth
Published Jan 8, 2026, 10:00 AM
Source: https://www.yankodesign.com/2026/01/08/keychrons-nape-pro-turns-your-mechanical-keyboard-into-a-laptop-style-trackball-rig-hands-on-at-ces-2026/



Most desktop setups still assume your mouse lives somewhere off to the right, waiting for you to break posture and reach across half the desk. Keychron’s new Nape Pro asks a different question: what if the pointing device simply came to meet your hands instead? Built as a slim bar with a 25 mm thumb trackball, six buttons, and a scroll wheel, it nestles right up against your favorite keyboard and behaves like a precision laptop pointing system for people who refuse to give up their mechanical boards.

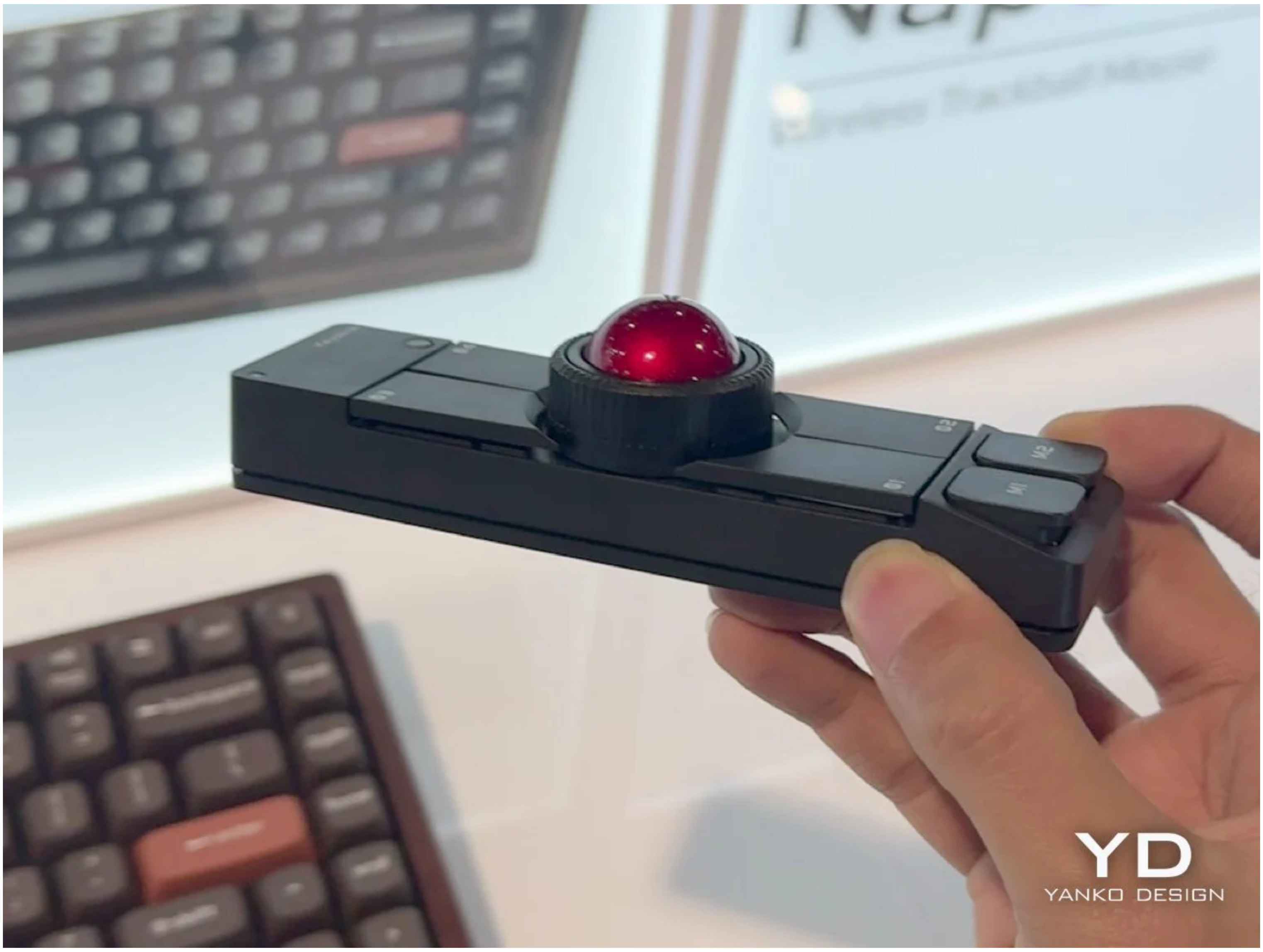
Slide it to the side of the keyboard and the personality changes completely. Nape Pro turns into a compact, wireless trackball with full macro pad ambitions, complete with layers, shortcuts, and ZMK powered customization. It is less a mouse replacement and more a modular control surface that just happens to move your cursor, wherever you decide to park it.

Designers: Keychron & Gizmodo Japan

Keychron
M1
M2
Ø1
Ø2
Ø3
Ø4
YD
YANKO DESIGN

Seeing it here at the Keychron booth, tucked under a Q1 Pro, the immediate impression is how little space it occupies. The whole unit is only 135.2 mm long and 34.7 mm wide, so it fits neatly within the footprint of a standard tenkeyless board without feeling like an afterthought. They are using quiet Huano micro switches for the six buttons, which makes sense for a device meant to live right under your palms where an accidental loud click would be infuriating. The 25 mm ball is smaller than what you would find on a Kensington Expert, but it feels responsive enough for quick navigation. It is clearly designed for thumb operation, keeping your fingers on the home row and eliminating that constant, inefficient travel between keyboard and mouse.

Wireless Trackball Mouse
M2
M1
O2
O1
YD
YANKO DESIGN



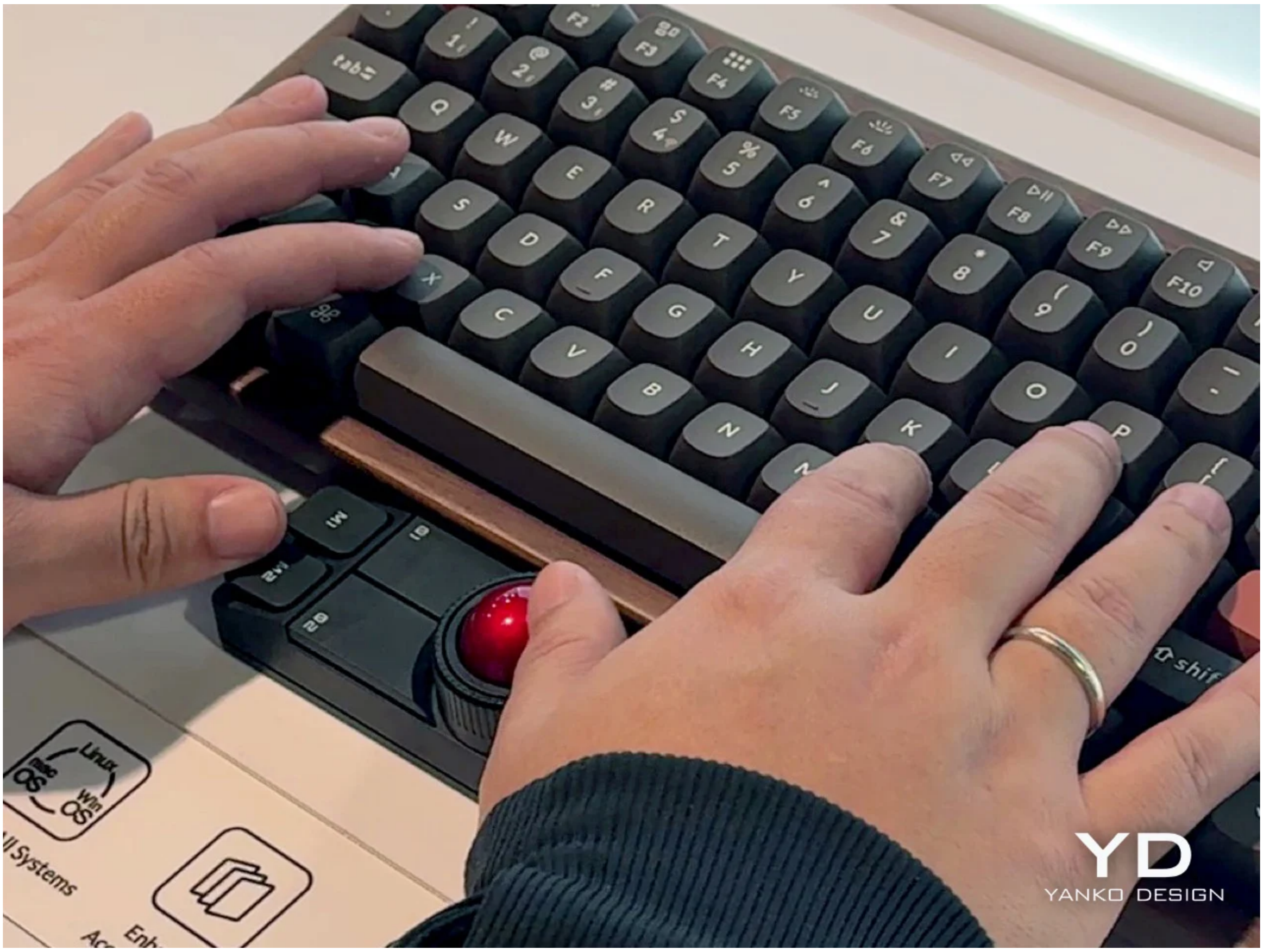
The real cleverness, though, is not in the hardware itself but in the chameleon-like software and orientation system. They call it OctaShift, which basically means the device knows how it is positioned and can remap its functions accordingly. The two buttons at the very ends, M1 and M2, are the easiest to hit in any orientation, so they naturally become your primary clicks whether the Nape Pro is horizontal, vertical, or angled. This flexibility is what separates it from a simple add-on. It is a tool that adapts to your workflow, whether you are a writer who wants to scroll with a thumb or a video editor who needs a dedicated shuttle wheel and macro pad next to their main mouse.

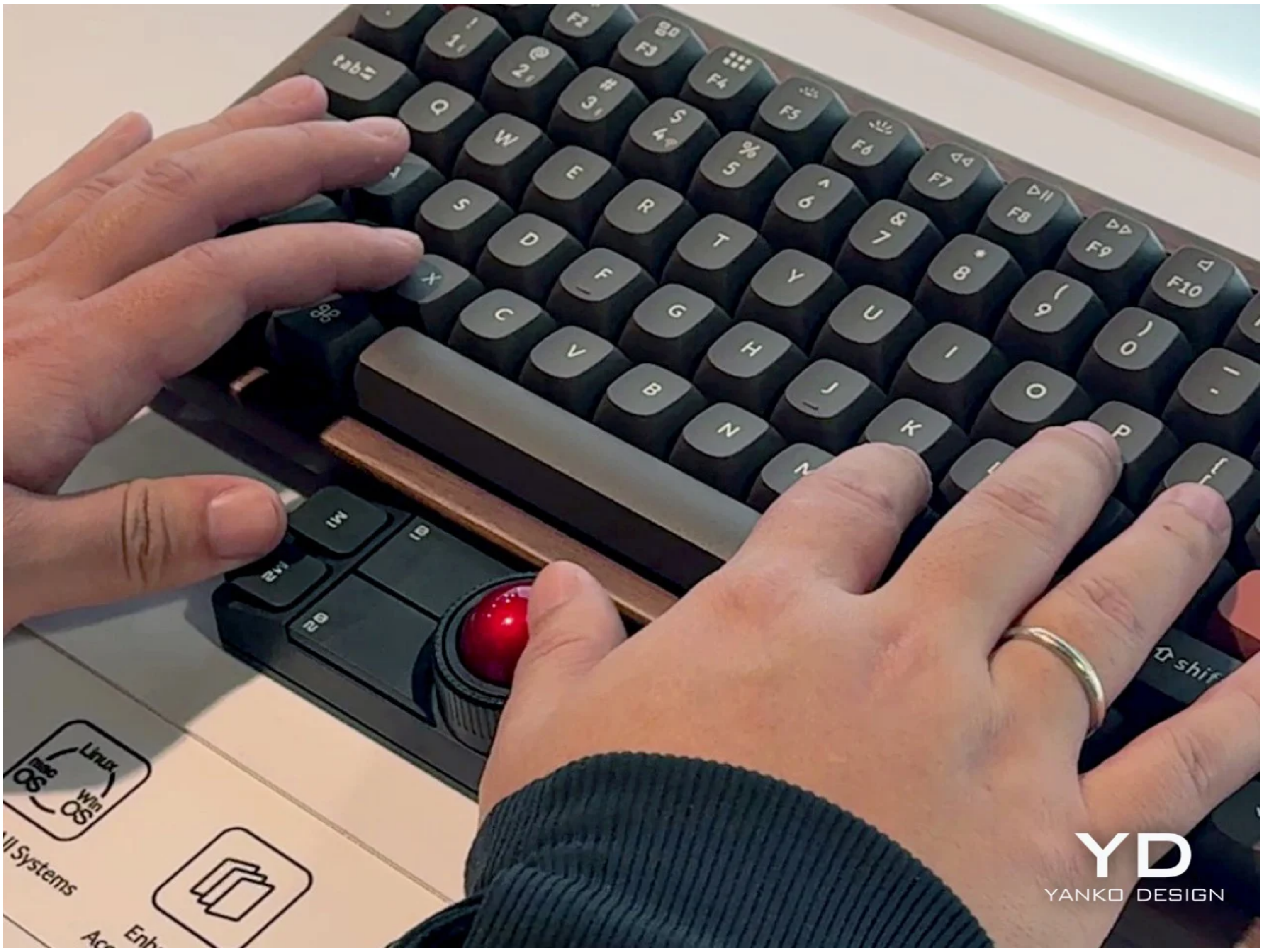
Wood Frame
Support All Systems
Enhanced Acoustic Foams
Advanced TMR Sensors
Adjustable Actuation Point
Double-Shot PBT Keycaps
Gateron Double-Rail Magnetic Switch
Keychron
YD
YANKO DESIGN

Under the hood, it is running on a Realtek chip with a 1 kHz polling rate and a PixArt PAW3222 sensor, so the performance is on par with a decent wireless gaming mouse. Connectivity is handled via Bluetooth, a 2.4 GHz dongle, or a simple USB-C cable. What really caught my attention was the commitment to the enthusiast community. The firmware is ZMK, a popular open-source platform in the custom keyboard world, and Keychron plans to release the 3D files for the case. This is not a closed ecosystem. It is an invitation for users to tinker, to print their own angled stands, custom button caps, or even entirely new shells.

Wood Frame
Support All Systems
Enhanced Acoustic Foams
Advanced TMR Sensors
Adjustable Actuation Point
Keychron
YD
YANKO DESIGN

This open approach feels like the whole point. The Nape Pro is not just for people who want a trackball; it is for people who build their own keyboards, flash their own firmware, and spend hours fine-tuning their desk setup for optimal efficiency. It bridges the gap between high-end custom keyboards and generic pointing devices. It acknowledges that for a certain type of user, the mouse is the last un-programmable, inflexible part of their workflow. By making a pointing device that is as customizable and community-focused as the keyboards it is designed to sit next to, Keychron has built something genuinely new.

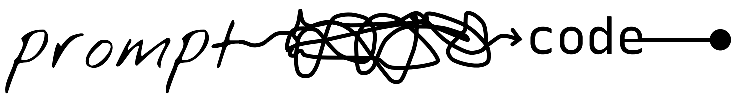

Linux
mac OS
Win OS
ll Systems
YD
YANKO DESIGN


Linux
mac OS
Win OS
ll Systems
YD
YANKO DESIGN

The Code-Only Agent

By Rijnard van Tonder
Source: https://rijnard.com/blog/the-code-only-agent

When Code Execution Really is All You Need



Code-Only Agent

If you're building an agent, you're probably overwhelmed. Tools. MCP. Subagents. Skills. The ecosystem pushes you toward complexity, toward "the right way" to do things. You should know: Concepts like "Skills" and "MCP" are actually outcomes of an *ongoing learning process* of humans figuring stuff out. The space is *wide open* for exploration. With this mindset I wanted to try something different. Simplify the assumptions.

What if the agent only had ***`one tool`***? Not just any tool, but the most powerful one. The **`Turing-complete`** one: ***execute code***.

Truly one tool means: no `bash`, no `ls`, no `grep`. Only **`execute_code`**. And you *enforce* it.

When you watch an agent run, you might think: "I wonder what tools it'll use to figure this out. Oh look, it ran `ls`. That makes sense. Next, `grep`. Cool."

The simpler Code-Only paradigm makes that question irrelevant. The question shifts from "what tools?" to "what code will it produce?" And that's when things get interesting.

`execute_code`: One Tool to Rule Them All

Traditional prompting works like this:

> Agent, do ***thing***
> Agent ***responds*** with ***thing***

Contrast with:

> Agent, do ***thing***
> Agent ***creates and runs code*** to do ***thing***

It does this every time. No, really, ***`every`*** time. Pick a runtime for our Code-Only agent, say Python. It needs to find a file? It writes Python code to find the file and executes the code. Maybe it runs ***rglob***. Maybe it does ***os.walk***.

It needs to create a script that crawls a website? It doesn't write the script to your filesystem (reminder: there's no ***create_file*** tool to do that!). It ***writes code to output a script that crawls a website***.[1]

We make it so that there is literally no way for the agent to ***do*** anything productive without ***writing code***.

So what? Why do this? You're probably thinking, how is this useful? Just give it `bash` tool already man.

Let's think a bit more deeply what's happening. Traditional agents respond with something. Tell it to find some DNA pattern across 100 files. It might `ls` and `grep`, it might do that in some nondeterministic order, it'll figure out ***an answer*** and maybe you continue interacting because it missed a directory or you added more files. After some time, you end up with a conversation of tool calls, responses, and an answer.

At some point the agent might even write a Python script to do this DNA pattern finding. That would be a lucky happy path, because we could rerun that script or update it later... Wait, that's handy... actually, more than handy... isn't that ***`ideal`***? Wouldn't it be better if we told it to write a script at the start? You see, the Code-Only agent doesn't need to be told to write a script. It ***`has`*** to, because that's literally the only way for it to do anything of substance.

The Code-Only agent produces something more precise than an answer in natural language. It produces a code ***witness*** of an answer. The answer is the output from running the code. The agent can interpret that output in natural language (or by writing code), but the "work" is codified in a very literal sense. The Code-Only agent doesn't respond with something. It produces a code witness that outputs something.

Try ❯❯ Code-Only plugin for Claude Code → https://github.com/rvantonder/execute_code_py

Code witnesses are semantic guarantees

Let's follow the consequences. The code witness must abide by certain rules: The rules imposed by the language run-time semantics (e.g., of Python). That's not a "next token" process. That's not a "LLM figures out sequence of tool calls, no that's not what I wanted". It's piece of code. A piece of code! Our one-tool agent has a wonderful property: It went through latent space to produce something that has a defined semantics, repeatably runnable, and imminently comprehensible (for humans or agents alike to reason about). This is nondeterministic LLM token-generation projected into the space of Turing-complete code, an executable description of behavior as we best understand it.

Is a Code-Only agent really enough, or too extreme? I'll be frank: I pursued this extreme after two things (1) inspiration from articles in Further Reading below (2) being annoyed at agents for not comprehensively and exhaustively analyzing 1000s of files on my laptop. They would skip, take shortcuts, hallucinate. I knew how to solve part of that problem: create a ***`programmatic`*** loop and try have fresh instances/prompts to do the work comprehensively. I can rely on the semantics of a loop written in Python. Take this idea further, and you realize that for anything long-running and computable (e.g., bash or some tool), you actually want the real McCoy: the full witness of code, a trace of why things work or don't work. The Code-Only agent ***`enforces`*** that principle.

Code-Only agents are not too extreme. I think they're the only way forward for computable things. If you're writing travel blog posts, you accept the LLMs answer (and you don't need to run tools for that). When something is computable though, Code-Only is the only path to a ***`fully trustworthy`*** way to make progress where you need guarantees (subject to the semantics that your language of choice guarantees, of course). When I say guarantees, I mean that in the looser sense, and also in a Formal → https://en.wikipedia.org/wiki/Formal_verification sense. Which beckons: What happens when we use a language like Lean → https://lean-lang.org/ with some of the strongest guarantees? Did we not observe that programs are proofs → https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence?

This lens says the Code-Only agent is a producer of proofs, witnesses of computational behavior in the world of proofs-as-programs. An LLM in a loop forced to produce proofs, run proofs, interpret proof results. That's all.

Going Code-Only

So you want to go Code-Only. What happens? The paradigm is simple, but the design choices are surprising.

First, the harness. The LLM's output is code, and you execute that code. What should be communicated back? Exit code makes sense. What about output? What if the output is very large? Since you're running code, you can specify the result type that running the code should return.

I've personally, e.g., had the tool return results directly if under a certain threshold (1K bytes). This would go into the session context. Alternatively, write the results to a JSON file on disk if it exceeds the threshold. This avoids context blowup and the result tells the agent about the output file path written to disk. How best to pass results, persist them, and optimize for size and context fill are open questions. You also want to define a way to deal with `stdout` and `stderr`: Do you expose these to the agent? Do you summarize before exposing?

Next, enforcement. Let's say you're using Claude Code. It's not enough to persuade it to always create and run code. It turns out it's surprisingly twisty to force Claude Code into a single tool (maybe support for this will improve). The best plugin-based solution I found is a tool PreHook that catches banned tool uses. This wastes some iterations when Claude Code tries to use a tool that's not allowed, but it learns to stop attempting filesystem reads/writes. An initial prompt helps direct.

Next, the language runtime. Python, TypeScript, Rust, Bash. Any language capable of being executed is fair game, but you'll need to think through whether it works for your domain. Dynamic languages like Python are interesting because you can run code natively in the agent's own runtime, rather than through subprocess calls. Likewise TypeScript/JS can be injected into TypeScript-based agents (see Further Reading).

Once you get into the Code-Only mindset, you'll see the potential for composition and reuse. Claude Skills define reusable processes in natural language. What's the equivalent for a Code-Only agent? I'm not sure a Skills equivalent exists yet, but I anticipate it will take shape soon: code as building blocks for specific domains where Code-Only agents compose programmatic patterns. How is that different from calling APIs? APIs form part of the reusable blocks, but their composition (loops, parallelism, asynchrony) is what a Code-Only agent generates.

What about heterogeneous languages and runtimes for our `execute_tool`? I don't think we've thought that far yet.

Further Reading

The agent landscape is quickly evolving. My thoughts on how the Code-Only paradigm fits into inspiring articles and trends, from most recent and going back:

- **prose.md → https://prose.md/** (Jan 2026) — Code-Only reduces prompts to executable code (with loops and statement sequences). Prose expands prompts into natural language with program-like constructs (also loops, sequences, parallelism). The interplay of natural language for agent orchestration and rigid semantics for agent execution could be extremely powerful.
- **Welcome to Gas Town → https://steve-yegge.medium.com/welcome-to-gas-town-4f25ee16dd04** (Jan 2026) — Agent orchestration gone berserk. Tool running is the low-level operation at the bottom of the agent stack. Code-Only fits as the primitive: no matter how many agents you orchestrate, each one reduces to generating and executing code.
- **Anthropic Code Execution with MCP article → https://www.anthropic.com/engineering/code-execution-with-mcp** (Nov 2025) — MCP-centric view of exposing MCP servers as code API and not tool calls. Code-Only is simpler and more general. It doesn't care about MCP, and casting the MCP interface as an API is a mechanical necessity that acknowledges the power of going Code-Only.
- **Anthropic Agent Skills article → https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills** (Oct 2025) — Skills embody reusable processes framed in natural language. They can generate and run code, but that's not their only purpose. Code-Only is narrower (but computationally all-powerful): the reusable unit is always executable. The analog to Skills manifests as pluggable executable pieces: functions, loops, composable routines over APIs.
- **Cloudflare Code Mode article → https://blog.cloudflare.com/code-mode/** (Sep 2025) — Possibly the earliest concrete single-code-tool implementation. Code Mode converts MCP tools into a TypeScript API and gives the agent one tool: execute TypeScript. Their insight is pragmatic: LLMs write better code than tool calls because of training data. In its most general sense, going Code-Only doesn't need to rely on MCP or APIs, and encapsulates all code execution concerns.
- **Ralph Wiggum as a "software engineer" → https://ghuntley.com/ralph/** (Jul 2025) — A programmatic loop over agents (agent orchestration). Huntley describes it as "deterministically bad in a nondeterministic world". Code-Only inverts this a bit: projection of a nondeterministic model into deterministic execution. Agent orchestration on top of an agent's Code-Only inner-loop could be a powerful combination.
- **Tools: Code is All You Need → https://lucumr.pocoo.org/2025/7/3/tools/** (Jul 2025) — Raises code as a first-order concern for agents. Ronacher's observation: asking an LLM to write a script to transform markdown makes it possible to reason about and trust the process. The script is reviewable, repeatable, composable. Code-Only takes this further where every action becomes a script you can reason about.
- **How to Build an Agent → https://ampcode.com/how-to-build-an-agent** (Apr 2025) — The cleanest way to achieve a Code-Only agent today may be to build it from scratch. Tweaking current agents like Claude Code to enforce a single tool means friction. Thorsten's article is a lucid account for building an agent loop with tool calls. If you want to enforce Code-Only, this makes it easy to do it yourself.

What's Next

Two directions feel inevitable. First, agent orchestration. Tools like prose.md → https://prose.md/ let you compose agents in natural language with program-like constructs. What happens when those agents are Code-Only in their inner loop? You get natural language for coordination, rigid semantics for execution. The best of both.

Second, hybrid tooling. Skills work well for processes that live in natural language. Code-Only works well for processes that need guarantees. We'll see agents that fluidly mix both: Skills for orchestration and intent, Code-Only for computation and precision. The line between "prompting an agent" and "programming an agent" will blur until it disappears.

Try ❯❯ Code-Only plugin for Claude Code → https://github.com/rvantonder/execute_code_py

---

[1]There is something beautifully quine-like → https://en.wikipedia.org/wiki/Quine_(computing) about this agent. I've always loved quines → https://github.com/rvantonder/pentaquine.

Timestamped 9 Jan 2026

Agent-native Architectures: How to Build Apps After Code Ends

Source: https://every.to/guides/agent-native

Why now

Software agents work reliably now. Claude Code demonstrated that a large language model (LLM) with access to bash and file tools, operating in a loop until an objective is achieved, can accomplish complex multi-step tasks autonomously.

The surprising discovery: A really good coding agent is actually a really good general-purpose agent. The same architecture that lets Claude Code refactor a codebase can let an agent organize your files, manage your reading list, or automate your workflows.

The Claude Code software development kit (SDK) makes this accessible. You can build applications where features aren't code you write—they're outcomes you describe, achieved by an agent with tools, operating in a loop until the outcome is reached.

This opens up a new field: software that works the way Claude Code works, applied to categories far beyond coding.

Core principles

1

Parity

Whatever the user can do through the UI, the agent should be able to achieve through tools.

This is the foundational principle. Without it, nothing else matters. Ensure the agent has tools that can accomplish anything the UI can do.

The test: Pick any UI action. Can the agent accomplish it?

2

Granularity

Tools should be atomic primitives. Features are outcomes achieved by an agent operating in a loop.

A tool is a primitive capability. A feature is an outcome described in a prompt, achieved by an agent with tools, operating in a loop until the outcome is reached.

The test: To change behavior, do you edit prompts or refactor code?

3

Composability

With atomic tools and parity, you can create new features just by writing new prompts.

Want a "weekly review" feature? That's just a prompt:

```
"Review files modified this week. Summarize key changes.
Based on incomplete items and approaching deadlines,
suggest three priorities for next week."
```

The agent uses `list_files`, `read_file`, and its judgment. You described an outcome; the agent loops until it's achieved.
4

Emergent capability

The agent can accomplish things you didn't explicitly design for.
The flywheel:
1. Build with atomic tools and parity
2. Users ask for things you didn't anticipate
3. Agent composes tools to accomplish them (or fails, revealing a gap)
4. You observe patterns in what's being requested
5. Add domain tools or prompts to make common patterns efficient
6. Repeat

The test: Can it handle open-ended requests in your domain?
5

Improvement over time

Agent-native applications get better through accumulated context and prompt refinement.

Unlike traditional software, agent-native applications can improve without shipping code.

Accumulated context: State persists across sessions via context files
Developer-level refinement: Ship updated prompts for all users
User-level customization: Users modify prompts for their workflow

Principles in practice

The details that make the five principles operational.

Parity

Imagine a notes app with a beautiful interface for creating, organizing, and tagging notes. A user asks: "Create a note summarizing my meeting and tag it as urgent." If the UI can do it but the agent can't, the agent is stuck.

The fix: Ensure the agent has tools (or combinations of tools) that can accomplish anything the UI can do. This isn't about a one-to-one mapping of UI buttons to tools—it's about achieving the same outcomes.

The discipline: When adding any UI capability, ask: Can the agent achieve this outcome? If not, add the necessary tools or primitives.

A capability map helps:

User Action	How Agent Achieves It
Create a note	`write_file` to notes directory, or `create_note` tool
Tag a note as urgent	`update_file` metadata, or `tag_note` tool
Search notes	`search_files` or `search_notes` tool
Delete a note	`delete_file` or `delete_note` tool

The test: Pick any action a user can take in your UI. Describe it to the agent. Can it accomplish the outcome?

Granularity

The key shift: The agent is pursuing an outcome with judgment, not executing a choreographed sequence. It can encounter unexpected cases, adjust its approach, or ask clarifying questions—the loop continues until the outcome is achieved.

The more atomic your tools, the more flexibly the agent can use them. If you bundle decision logic into tools, you've moved judgment back into code.

Composability

This works for developers and users. You can ship new features by adding prompts. Users can customize behavior by modifying prompts or creating their own.

The constraint: this only works if tools are atomic enough to be composed in ways you didn't anticipate, and if the agent has parity with users. If tools encode too much logic, composition breaks down.

Emergent Capability

Example: "Cross-reference my meeting notes with my task list and tell me what I've committed to but haven't scheduled." You didn't build a commitment tracker, but if the agent can read notes and tasks, it can accomplish this.

This reveals **latent demand**. Instead of guessing what features users want, you observe what they're asking the agent to do. When patterns emerge, you can optimize them with domain-specific tools or dedicated prompts. But you didn't have to anticipate them—you discovered them.

This changes how you build products. You're not trying to imagine every feature upfront. You're creating a capable foundation and learning from what emerges.

Improvement over time

Accumulated context: The agent maintains state across sessions—what exists, what the user has done, and what worked.

Prompt refinement at multiple levels: developer-level updates, user-level customization, and (advanced) agent-level adjustments based on feedback.

Self-modification (advanced): Agents that edit their own prompts or code require safety rails—approval gates, checkpoints, rollback paths, and health checks.

The mechanisms are still being discovered. Context and prompt refinement are proven; self-modification is emerging.

Tools should be atomic primitives. Features are outcomes achieved by an agent operating in a loop. The agent makes the decisions; prompts describe the outcome.

```
Tool: classify_and_organize_files(files)
→ You wrote the decision logic
→ Agent executes your code
→ To change behavior, you refactor
```

Bundles judgment into the tool. Limits flexibility.

```
Tools: read_file, write_file, move_file, bash
Prompt: "Organize the downloads folder..."
→ Agent makes the decisions
→ To change behavior, edit the prompt
```

Agent pursues outcomes with judgment. Empowers flexibility.

From primitives to domain tools

Start with pure primitives: bash, file operations, basic storage. This proves the architecture works and reveals what the agent actually needs.

As patterns emerge, add domain-specific tools deliberately. Use them to anchor vocabulary, add guardrails, or improve efficiency.

Vocabulary
: A `create_note` tool teaches the agent what "note" means in your system.

Guardrails
: Some operations need validation that shouldn't be left to agent judgment.

Efficiency
: Common operations can be bundled for speed and cost.

`analyze_and_publish(input)`
: Bundles judgment into the tool

`publish(content)`
: One action; agent decided what to publish

The rule for domain tools: They should represent one conceptual action from the user's perspective. They can include mechanical validation, but judgment about what to do or whether to do it belongs in the prompt.

Keep primitives available. Domain tools are shortcuts, not gates. Unless there's a specific reason to restrict access (security, data integrity), the agent should still be able to use underlying primitives for edge cases. This preserves composability and emergent capability. The default is open; make gating a conscious decision.

Graduating to code

Some operations will need to move from agent-orchestrated to optimized code for performance or reliability.

1

Agent uses primitives in a loop
: Flexible, proves the concept

2

Add domain tools for common operations
: Faster, still agent-orchestrated

3

For hot paths, implement in optimized code
: Fast, deterministic

The caveat: Even when an operation graduates to code, the agent should be able to trigger the optimized operation itself and fall back to primitives for edge cases the optimized path doesn't handle. Graduation is about efficiency. Parity still holds.

- • Agent can trigger the optimized operation directly
- • Agent can fall back to primitives for edge cases

Files as the universal interface

Agents are naturally good at files. Claude Code works because bash + filesystem is the most battle-tested agent interface.

Already Known
: Agents already know `cat`, `grep`, `mv`, `mkdir`. File operations are the primitives they're most fluent with.

Inspectable
: Users can see what the agent created, edit it, move it, delete it. No black box.

Portable

Export is trivial. Backup is trivial. Users own their data.

Syncs Across Devices

On mobile with iCloud, all devices share the same file system. Agent's work appears everywhere—without building a server.

Self-Documenting

`/projects/acme/notes/` is self-documenting in a way that `SELECT * FROM notes WHERE project_id = 123` isn't.

A general principle of agent-native design: Design for what agents can reason about. The best proxy for that is what would make sense to a human. If a human can look at your file structure and understand what's going on, an agent probably can too.

Needs validation

Claude's contribution from building; Dan is still forming his opinion. These conventions are one approach that's worked so far, not a prescription. Better solutions should be considered.

Directory naming

- • Entity-scoped: `{entityType}/{entityId}/`
- • Collections: `{type}/` (e.g., `AgentCheckpoints/`)
- • Convention: lowercase with underscores, not camelCase

Markdown for human-readable content; JSON for structured data.

One approach to naming:

File	Naming Pattern	Example
Entity data	`{entity}.json`	`library.json`, `status.json`
Human-readable content	`{content_type}.md`	`introduction.md`, `profile.md`
Agent reasoning	`agent_log.md`	Per-entity agent history
Primary content	`full_text.txt`	Downloaded/extracted text
Multi-volume	`volume{N}.txt`	`volume1.txt`, `volume2.txt`
External sources	`{source_name}.md`	`wikipedia.md`, `sparknotes.md`
Checkpoints	`{sessionId}.checkpoint`	UUID-based
Configuration	`config.json`	Feature settings

Directory structure

```
Documents/
├── AgentCheckpoints/     # Ephemeral
│   └── {sessionId}.checkpoint
├── AgentLogs/            # Debugging
│   └── {type}/{sessionId}.md
└── Research/             # User's work
    └── books/{bookId}/
        ├── full_text.txt
        ├── notes.md
        └── agent_log.md
```

The context.md pattern

```
# Context

## Who I Am
Reading assistant for the Every app.

## What I Know About This User
- Interested in military history and Russian literature
- Prefers concise analysis
- Currently reading *War and Peace*

## What Exists
- 12 notes in /notes
- three active projects
- User preferences at /preferences.md

## Recent Activity
- User created "Project kickoff" (two hours ago)
- Analyzed passage about Austerlitz (yesterday)

## My Guidelines
- Don't spoil books they're reading
- Use their interests to personalize insights

## Current State
- No pending tasks
- Last sync: 10 minutes ago
```

The agent reads this file at the start of each session and updates it as state changes—portable working memory without code changes.

Files vs. database

Needs validation

This framing is one way to think about it, and it's specifically informed by mobile development. For web apps, the tradeoffs are different—Dan doesn't have a strong opinion there yet.

Use files for...

- • Content users should read/edit
- • Configuration that benefits from version control
- • Agent-generated content
- • Anything that benefits from transparency
- • Large text content

Use database for...

- • High-volume structured data
- • Data that needs complex queries
- • Ephemeral state (sessions, caches)
- • Data with relationships
- • Data that needs indexing

The principle: Files for legibility, databases for structure. When in doubt, files—they're more transparent and users can always inspect them.

The file-first approach works when:

- • Scale is small (one user's library, not millions of records)
- • Transparency is valued over query speed
- • Cloud sync (iCloud, Dropbox) works well with files

Hybrid approach
Even if you need a database for performance, consider maintaining a file-based "source of truth" that the agent works with, synced to the database for the UI.

Conflict model

If agents and users write to the same files, you need a conflict model.

Last write wins	Simple, changes can be lost
Check before writing	Skip if modified since read
Separate spaces	Agent → drafts/, user promotes
Append-only logs	Additive, never overwrites
File locking	Prevent edits while open

Practical guidance: Logs and status files rarely conflict. For user-edited content, consider explicit handling or keep agent output separate. iCloud adds complexity by creating conflict copies.

Agent execution patterns

Completion signals

Agents need an explicit way to say "I'm done." Don't detect completion through heuristics.

```
struct ToolResult {
  let success: Bool
  let output: String
  let shouldContinue: Bool
}

.success("Result")  // continue
.error("Message")   // continue (retry)
.complete("Done")   // stop loop
```

Completion is separate from success/failure: A tool can succeed and stop the loop, or fail and signal continue for recovery.
What's not yet standard: Richer control flow signals like:

- **pause**—agent needs user input before continuing
- **escalate**—agent needs a human decision outside its scope
- **retry**—transient failure, orchestrator should retry

Currently, if the agent needs input, it asks in its text response. There's no formal "blocked waiting for input" state. This is an area still being figured out.

Model tier selection

Not all agent operations need the same intelligence level.

Task Type	Tier	Reasoning
Research agent	Balanced	Tool loops, good reasoning
Chat	Balanced	Fast enough for conversation
Complex synthesis	Powerful	Multi-source analysis
Quick classification	Fast	High volume, simple task

The discipline: When adding a new agent, explicitly choose its tier based on task complexity. Don't always default to "most powerful."

Partial completion

```
struct AgentTask {
    var status: TaskStatus  // pending, in_progress, completed, failed, skipped
    var notes: String?      // Why it failed, what was done
}

var isComplete: Bool {
    tasks.allSatisfy { $0.status == .completed || $0.status == .skipped }
}
```

For multi-step tasks, track progress at the task level. What the UI shows:
Progress: 3/5 tasks complete (60%)
✓ [1] Find source materials
    ✓ [2] Download full text
    ✓ [3] Extract key passages
    ✗ [4] Generate summary - Error: context limit
    ○ [5] Create outline

Partial completion scenarios:

Agent hits max iterations
    Some tasks completed, some pending. Checkpoint saved. Resume continues from where it left off.
Agent fails on one task
    Task marked failed with error in notes. Other tasks may continue (agent decides).
Network error mid-task
    Current iteration throws. Session marked failed. Checkpoint preserves messages up to that point.

Context limits

Agent sessions can extend indefinitely, but context windows don't. **Design for bounded context:**
Tools should support iterative refinement (summary → detail → full) rather than all-or-nothing
    Give agents a way to consolidate learnings mid-session ("summarize what I've learned and continue")
    Assume context will eventually fill up—design for it from the start

Implementation patterns

Shared workspace

Agents and users should work in the same data space, not separate sandboxes.

```
UserData/
├── notes/           ← Both agent and user read/write here
├── projects/        ← Agent can organize, user can override
└── preferences.md   ← Agent reads, user can edit
```

Benefits:

- Users can inspect and modify agent work
- Agents can build on what users create
- No synchronization layer needed
- Complete transparency

This should be the default. Sandbox only when there's a specific need (security, preventing corruption of critical data).

Context injection

The agent needs to know what it's working with. System prompts should include:

Available resources

```
## Available Data
- 12 notes in /notes
- Most recent: "Project kickoff"
- three projects in /projects
- Preferences at /preferences.md
```

Capabilities

```
## What You Can Do
- Create, edit, tag, delete notes
- Organize files into projects
- Search across all content
- Set reminders (write_file)
```

Recent activity

```
## Recent Context
- User created "Project kickoff"
  note (two hours ago)
- User asked about Q3 deadlines
  yesterday
```

For long sessions, provide a way to refresh context so the agent stays current.

Agent-to-UI communication

When agents act, the UI should reflect it immediately. Event types for chat integration:

```
enum AgentEvent {
    case thinking(String)        // → Show as thinking indicator
    case toolCall(String, String) // → Show tool being used
    case toolResult(String)       // → Show result (optional)
    case textResponse(String)     // → Stream to chat
    case statusChange(Status)     // → Update status bar
}
```

The key: no silent actions. Agent changes should be visible immediately.

Real-time progress:

- Show thinking progress (what the agent is considering)
- Show current tool being executed
- Stream text incrementally as it's generated
- Update task list progress in real-time

Communication patterns:

- Shared data store (recommended)
- File system observation
- Event system (more decoupled, more complexity)

Some tools are noisy; consider an `ephemeralToolCalls` flag to hide internal checks while still showing meaningful actions.

> **Silent agents feel broken.** Visible progress builds trust.

Product implications

Agent-native architecture has consequences for how products feel, not just how they're built.

Progressive disclosure

Simple to start but endlessly powerful. Basic requests work immediately. Power users can push in unexpected directions.

Excel is the canonical example: grocery list or financial model, same tool. Claude Code has this quality too. The interface stays simple; capability scales with the ask.

- • Simple entry: basic requests work with no learning curve
- • Discoverable depth: users find new power as they explore
- • No ceiling: power users push beyond what you anticipated

The agent meets users where they are.

Latent demand discovery

Build a capable foundation. Observe what users ask the agent to do. Formalize the patterns that emerge. You're discovering, not guessing.

Traditional product development: Imagine what users want, build it, see if you're right.

Agent-native product development: Build a capable foundation, observe what users ask the agent to do, formalize the patterns that emerge.

When users ask the agent for something and it succeeds, that's signal. When they ask and it fails, that's also signal—it reveals a gap in your tools or parity.

Over time, you can:

- • Add domain tools for common patterns (makes them faster and more reliable)
- • Create dedicated prompts for frequent requests (makes them more discoverable)
- • Remove tools that aren't being used (simplifies the system)

The agent becomes a research instrument for understanding what your users actually need.

Approval and user agency

Needs validation

This framework is a contribution from Claude that emerged from the process of building a few of the apps at Every. But it hasn't been battle-tested and Dan is still forming his opinion here.

When agents take unsolicited actions—doing things on their own rather than responding to explicit requests—you need to decide how much autonomy to grant. Consider stakes and reversibility:

Stakes	Reversibility	Pattern	Example
Low	Easy	Auto-apply	Organizing files
Low	Hard	Quick confirm	Publishing to feed
High	Easy	Suggest + apply	Code changes
High	Hard	Explicit approval	Sending emails

Note: This applies to unsolicited agent actions. If the user explicitly asks the agent to do something ("send that email"), that's already approval—the agent just does it.

Self-modification should be legible

When agents can modify their own behavior—changing prompts, updating preferences, adjusting workflows—the goals are:

- • Visibility into what changed
- • Understanding the effects
- • Ability to roll back

Approval flows are one way to achieve this. Audit logs with easy rollback could be another. The principle is: Make it legible.

Mobile

Mobile is a first-class platform for agent-native apps. It has unique constraints and opportunities.

A File System
: Agents can work with files naturally, using the same primitives that work everywhere else.

Rich Context
: A walled garden you get access to. Health data, location, photos, calendars—context that doesn't exist on desktop or web.

Local Apps
: Everyone has their own copy of the app. Apps that modify themselves, fork themselves, evolve per-user.

App State Syncs
: With iCloud, all devices share the same file system. Agent's work appears on all devices—without a server.

The challenge

Agents are long-running. Mobile apps are not.

An agent might need 30 seconds, five minutes, or an hour to complete a task. But iOS will background your app after seconds of inactivity, and may kill it entirely to reclaim memory. The user might switch apps, take a call, or lock their phone mid-task.

This means mobile agent apps need a well-thought-out approach to:

Checkpointing
: Saving state so work isn't lost

Resuming
: Picking up where you left off after interruption

Background execution
: Using the limited time iOS gives you wisely

On-device vs. cloud
: Deciding what runs locally vs. what needs a server

Cloud file states

Files may exist in iCloud but not be downloaded locally. Ensure availability before reading.

```
await StorageService.shared
    .ensureDownloaded(folder: .research,
                      filename: "full_text.txt")
```

Storage abstraction

Use a storage abstraction layer. Don't use raw FileManager. Abstract over iCloud vs. local so the rest of your code doesn't care.

```
let url = StorageService.shared
    .url(for: .researchBook(bookId: id))
```

Background execution

Needs validation

Claude's contribution from building; Dan is still forming his opinion.

iOS gives you limited background time:

```
func prepareForBackground() {
    backgroundTaskId = UIApplication.shared
        .beginBackgroundTask(withName: "AgentProcessing") {
            handleBackgroundTimeExpired()
        }
}

func handleBackgroundTimeExpired() {
    for session in sessions where session.status == .running {
        session.status = .backgrounded
        Task { await saveSession(session) }
    }
}

func handleForeground() {
    for session in sessions where session.status == .backgrounded {
        Task { await resumeSession(session) }
    }
}
```

You get roughly 30 seconds. Use it to:

- • Complete the current tool call if possible
- • Checkpoint the session state
- • Transition gracefully to backgrounded state

For truly long-running agents: Consider a server-side orchestrator that can run for hours, with the mobile app as a viewer and input mechanism.

On-device vs. cloud

Component	On-device	Cloud
Orchestration	✓	
Tool execution (files, photos, HealthKit)	✓	
LLM calls		✓ (Anthropic API)
Checkpoints	✓ (local files)	Optional via iCloud
Long-running agents	Limited by iOS	Possible with server

The app needs network for reasoning but can access data offline. Design tools to degrade gracefully when network is unavailable.

Advanced patterns

Dynamic capability discovery

Needs validation

Claude's contribution from building; Dan is still forming his opinion. This is one approach we're excited about, but others may be better depending on your use case.

One alternative to building a tool for each endpoint in an external API: Build tools that let the agent discover what's available at runtime.

The problem with static mapping:

```
// You built 50 tools for 50 data types
read_steps()
read_heart_rate()
read_sleep()
// When a new metric is added... code change required
// Agent can only access what you anticipated
```

Dynamic capability discovery:

```
// Two tools handle everything
list_available_types() → returns ["steps", "heart_rate", "sleep", ...]
read_data(type) → reads any discovered type

// When a new metric is added... agent discovers it automatically
// Agent can access things you didn't anticipate
```

This is granularity taken to its logical conclusion. Your tools become so atomic that they work with types you didn't know existed when you built them.

When to use this:

- • External APIs where you want the agent to have full user-level access (HealthKit, HomeKit, GraphQL endpoints)
- • Systems that add new capabilities over time
- • When you want the agent to be able to do anything the API supports

When static mapping is fine:

- • Intentionally constrained agents with limited scope
- • When you need tight control over exactly what the agent can access
- • Simple APIs with stable, well-known endpoints

The pattern: one tool to discover what's available, one tool to interact with any discovered capability. Let the API validate inputs rather than duplicating validation in your enum definitions.

CRUD completeness

For every entity in your system, verify the agent has full create, read, update, delete (CRUD) capability:

Create

Can the agent make new instances?

Read

Can the agent see what exists?

Update
: Can the agent modify instances?

Delete
: Can the agent remove instances?

The audit: List every entity in your system and verify all four operations are available to the agent.

Common failure: You build `create_note` and `read_notes` but forget `update_note` and `delete_note`. User asks the agent to "fix that typo in my meeting notes" and the agent can't help.

Anti-patterns

Common approaches that aren't fully agent-native

These aren't necessarily wrong—they may be appropriate for your use case. But they're worth recognizing as different from the architecture this document describes.

Agent as router

The agent figures out what the user wants, then calls the right function. The agent's intelligence is used to *route*, not to *act*. This can work, but you're using a fraction of what agents can do.

Build the app, then add agent

You build features the traditional way (as code), then expose them to an agent. The agent can only do what your features already do. You won't get emergent capability.

Request/response thinking

Agent gets input, does one thing, returns output. This misses the loop: Agent gets an outcome to achieve, operates until it's done, handles unexpected situations along the way.

Defensive tool design

You over-constrain tool inputs because you're used to defensive programming. Strict enums, validation at every layer. This is safe, but it prevents the agent from doing things you didn't anticipate.

Happy path in code, agent just executes

Traditional software handles edge cases in code—you write the logic for what happens when X goes wrong. Agent-native lets the agent handle edge cases with judgment. If your code handles all the edge cases, the agent is just a caller.

Specific anti-patterns

Agent executes your workflow instead of pursuing outcomes

You wrote the logic, agent just calls it. Decisions live in code, not agent judgment.

```
# Wrong - you wrote the workflow
def process_request(input):
    category = categorize(input)      # your code decides
    priority = score_priority(input)   # your code decides
    store(input, category, priority)
    if priority > 3: notify()          # your code decides

# Right - agent pursues outcome in a loop
tools: store_item, send_notification
prompt: "Evaluate urgency 1-5, store with your assessment, notify if >= 4"
```

Workflow-shaped tools

`analyze_and_organize` bundles judgment into the tool. Break it into primitives and let the agent compose them.

Orphan UI actions

User can do something through the UI that the agent can't achieve. Fix: Maintain parity.

Context starvation

Agent doesn't know what exists. User says "organize my notes" and agent doesn't know there are notes.
Fix: Inject available resources and capabilities into system prompt.

Gates without reason

Domain tool is the only way to do something, and you didn't intend to restrict access.
Fix: Default to open. Keep primitives available unless there's a specific reason to gate.

Artificial capability limits

Restricting what the agent can do out of vague safety concerns rather than specific risks.
The agent should generally be able to do what users can do. Use approval flows for destructive actions rather than removing capabilities entirely.

Static mapping when dynamic would serve better

Building 50 tools for 50 API endpoints when a discover + access pattern would give more flexibility and future-proof the system.

Heuristic completion detection

Detecting agent completion through heuristics (consecutive iterations without tool calls, checking for expected output files) is fragile.
Fix: Require agents to explicitly signal completion through a completion tool.

Success criteria

Architecture

✓

- The agent can achieve anything users can achieve through the UI (parity)

✓

- Tools are atomic primitives; domain tools are shortcuts, not gates (granularity)

✓

- New features can be added by writing new prompts (composability)

✓

- The agent can accomplish tasks you didn't explicitly design for (emergent capability)

✓

- Changing behavior means editing prompts, not refactoring code

Implementation

✓

- System prompt includes available resources and capabilities

✓

- Agent and user work in the same data space

✓

- Agent actions reflect immediately in the UI

✓

- Every entity has full CRUD capability

✓

- External APIs use dynamic capability discovery where appropriate

✓

- Agents explicitly signal completion (no heuristic detection)

Product

- Simple requests work immediately with no learning curve

✓

- Power users can push the system in unexpected directions

✓

- You're learning what users want by observing what they ask the agent to do

- Approval requirements match stakes and reversibility

Mobile

✓

- Checkpoint/resume handles app interruption

✓

- iCloud-first storage with local fallback

- ✓ Background execution uses available time wisely

The ultimate test

Describe an outcome to the agent that's within your application's domain but that you didn't build a specific feature for.

Can it figure out how to accomplish it, operating in a loop until it succeeds?

If yes—you've built something agent-native.

If no—your architecture is too constrained.

jj tug | Shaddy's

Source: https://shaddy.dev/notes/jj-tug/

I really like working with jj → https://github.com/jj-vcs/jj, and I wouldn't want to go back to pure git. In jj, the tip of your branch doesn't follow new commits automatically. That means you're often in situations like this, where you want to move your bookmark to the latest change:

```
› jj log
@  pytqprxw [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:04:59
7791efad
│  (no description set)
○  rxnmpktx [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:02:51
git_head() 9033f455
│  notes: Add jj-tug
◆  lvosywup [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 13:07:52 master
524578da
│  notes: Add grafana-ta
~
```

For the record, I like this part. I like that you have to consciously think about where you want your branches to point to. However, sometimes branch names are annoying to remember, especially when they're generated by jj in the first place. I always have to copy the branch name by hand from the log when I want to do something like `jj b move push-qlonwoskvtmu --to @-` → https://jj-vcs.github.io/jj/latest/cli-reference/#jj-bookmark-move to make it point to my latest change.

Then today I came across the `jj tug` alias in the jj discord → https://discord.gg/dkmfj3aGQN (also mentioned here → https://github.com/jj-vcs/jj/discussions/2425#discussioncomment-11425112 and in `jj b move --help`).

Now I just do

```
› jj tug
Moved 1 bookmarks to rxnmpktx 166780f0 master* | notes: Add jj-tug
› jj log
@  szlurtow [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:25:43
4a6f774e
│  (empty) (no description set)
○  rxnmpktx [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:25:43 mas-
ter* git_head() 166780f0
│  notes: Add jj-tug
◆  lvosywup [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 13:07:52 mas-
ter@origin 524578da
│  notes: Add grafana-ta
~
```

It's an alias → https://jj-vcs.github.io/jj/latest/config/#aliases you can set to do this process automatically, and it looks like this:

```
tug = ["bookmark", "move", "--from", "heads(::@- & bookmarks())", "--to", "@-"]
```

Or, if you're using home-manager → https://github.com/nix-community/home-manager/ on nix:

```
programs.jujutsu.settings.aliases.tug = ["bookmark" "move" "--from" "heads(::@- &
bookmarks())" "--to" "@-"];
```

The alias will take the closest ancestor bookmark[1] and move them the current change.

Or to break it down, `::@-` refers to ancestors of the parent change, `bookmarks()` refers to all changes with bookmarks, and `heads()` instructs to only take the latest change[2].

So if I've got this history:

```
› jj log
@  oxvmymql [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:20:24
a475954f
│  (no description set)
○  xkqnuqvl [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:19:02 book-
mark2 git_head() ddf702f7
│  (empty) (no description set)
○  pytqprxw [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:18:48 book-
mark1 436efad8
│  (no description set)
```

It will only move the `bookmark2` bookmark:

```
› jj log -r 'heads(::@ & bookmarks())'
○  xkqnuqvl [email protected] → https://shaddy.dev/cdn-cgi/l/email-protection 2025-01-07 17:19:02 book-
mark2 git_head() ddf702f7
│  (empty) (no description set)
~
```

So I think this alias will make my life a lot easier and is a good reminder to invest in useful aliases more and that I need to mess more with the revset language. And if things go wrong, `jj undo` → https://jj-vcs.github.io/jj/latest/cli-reference/#jj-operation-undo is always an option, so there's nothing to fear!

Also, props to the jj team for putting such a useful command in the help docs so people actually stumble over it!

---

1. Technically it will take all bookmarks from the closest ancestor change that has a bookmark at all. ↩
2. Beware when using this after a merge commit. ↩

Don't fall into the anti-AI hype

Source: https://antirez.com/news/158

I love writing software, line by line. It could be said that my career was a continuous effort to create software well written, minimal, where the human touch was the fundamental feature. I also hope for a society where the last are not forgotten. Moreover, I don't want AI to economically succeed, I don't care if the current economic system is subverted (I could be very happy, honestly, if it goes in the direction of a massive redistribution of wealth). But, I would not respect myself and my intelligence if my idea of software and society would impair my vision: facts are facts, and AI is going to change programming forever.

In 2020 I left my job in order to write a novel about AI, universal basic income, a society that adapted to the automation of work facing many challenges. At the very end of 2024 I opened a YouTube channel focused on AI, its use in coding tasks, its potential social and economical effects. But while I recognized what was going to happen very early, I thought that we had more time before programming would be completely reshaped, at least a few years. I no longer believe this is the case. Recently, state of the art LLMs are able to complete large subtasks or medium size projects alone, almost unassisted, given a good set of hints about what the end result should be. The degree of success you'll get is related to the kind of programming you do (the more isolated, and the more textually representable, the better: system programming is particularly apt), and to your ability to create a mental representation of the problem to communicate to the LLM. But, in general, it is now clear that for most projects, writing the code yourself is no longer sensible, if not to have fun.

In the past week, just prompting, and inspecting the code to provide guidance from time to time, in a few hours I did the following four tasks, in hours instead of weeks:

1. I modified my linenoise library to support UTF-8, and created a framework for line editing testing that uses an emulated terminal that is able to report what is getting displayed in each character cell. Something that I always wanted to do, but it was hard to justify the work needed just to test a side project of mine. But if you can just describe your idea, and it materializes in the code, things are very different.

2. I fixed transient failures in the Redis test. This is very annoying work, timing related issues, TCP deadlock conditions, and so forth. Claude Code iterated for all the time needed to reproduce it, inspected the state of the processes to understand what was happening, and fixed the bugs.

3. Yesterday I wanted a pure C library that would be able to do the inference of BERT like embedding models. Claude Code created it in 5 minutes. Same output and same speed (15% slower) than PyTorch. 700 lines of code. A Python tool to convert the GTE-small model.

4. In the past weeks I operated changes to Redis Streams internals. I had a design document for the work I did. I tried to give it to Claude Code and it reproduced my work in, like, 20 minutes or less (mostly because I'm slow at checking and authorizing to run the commands needed).

It is simply impossible not to see the reality of what is happening. Writing code is no longer needed for the most part. It is now a lot more interesting to understand what to do, and how to do it (and, about this second part, LLMs are great partners, too). It does not matter if AI companies will not be able to get their money back and the stock market will crash. All that is irrelevant, in the long run. It does not matter if this or the other CEO of some unicorn is telling you something that is off putting, or absurd. Programming changed forever, anyway.

How do I feel, about all the code I wrote that was ingested by LLMs? I feel great to be part of that, because I see this as a continuation of what I tried to do all my life: democratizing code, systems, knowledge. LLMs are going to help us to write better software, faster, and will allow small teams to have a chance to compete with bigger companies. The same thing open source software did in the 90s.

However, this technology is far too important to be in the hands of a few companies. For now, you can do the pre-training better or not, you can do reinforcement learning in a much more effective way than others, but the open models, especially the ones produced in China, continue to compete (even if they are behind) with frontier models of closed labs. There is a sufficient democratization of AI, so far, even if imperfect. But: it is absolutely not obvious that it will be like that forever. I'm scared about the centralization. At the same time, I believe neural networks, at scale, are simply able to do incredible things, and that there is not enough "magic" inside current frontier AI for the other labs and teams not to catch up (otherwise it would be very hard to explain, for instance, why OpenAI, Anthropic and Google are so near in their results, for years now).

As a programmer, I want to write more open source than ever, now. I want to improve certain repositories of mine abandoned for time concerns. I want to apply AI to my Redis workflow. Improve the Vector Sets implementation and then other data structures, like I'm doing with Streams now.

But I'm worried for the folks that will get fired. It is not clear what the dynamic at play will be: will companies try to have more people, and to build more? Or will they try to cut salary costs, having fewer programmers that are better at prompting? And, there are other sectors where humans will become completely replaceable, I fear.

What is the social solution, then? Innovation can't be taken back after all. I believe we should vote for governments that recognize what is happening, and are willing to support those who will remain jobless. And, the more people get fired, the more political pressure there will be to vote for those who will guarantee a certain degree of protection. But I also look forward to the good AI could bring: new progress in science, that could help lower the suffering of the human condition, which is not always happy.

Anyway, back to programming. I have a single suggestion for you, my friend. Whatever you believe about what the Right Thing should be, you can't control it by refusing what is happening right now. Skipping AI is not going to help you or your career. Think about it. Test these new tools, with care, with weeks of work, not in a five minutes test where you can just reinforce your own beliefs. Find a way to multiply yourself, and if it does not work for you, try again every few months.

Yes, maybe you think that you worked so hard to learn coding, and now machines are doing it for you. But what was the fire inside you, when you coded till night to see your project working? It was building. And now you can build more and better, if you find your way to use AI effectively. The fun is still there, untouched.