Rijnard
van Tonder

𝕏 @rvtond

Home
← Back to all posts

The Code-Only Agent

When Code Execution Really is All You Need

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.***[I]

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

Code witnesses are semantic guarantees

Let's follow the consequences. The code witness must abide by certain rules: The rules imposed by the language runtime 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 sense. Which beckons: What happens when we use a language like Lean with some of the strongest guarantees? Did we not observe that programs are proofs?

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*** (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*** (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*** (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*** (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*** (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"*** (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*** (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*** (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 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

[1]There is something beautifully quine-like about this agent. I've always loved quines.

Timestamped 9 Jan 2026