

pi profiles, part 1: a shelf for the agents you build

[Adrian Tame Jacobo](#), writing for Plicara Labs · 2026-10-13 ·

I have been lucky enough to use a lot of coding agents. I'd call myself a heavy user of all of them, but I definitely like some more than others. The one I keep coming back to as my main harness is Pi, and this article is about a small system I built on top of it, which I've called pi-profiles.

The short version: a profile is a separate Pi setup for a separate way of working. Think Chrome profiles, but for your coding agent. It is not only which tools are loaded. It is how much the agent does on its own, what it is allowed to touch and when it is supposed to stop. I have one profile for everyday work at the keyboard, one for long unattended runs and one I keep only for testing a single extension. I pick the one I want when I launch Pi, and my session history follows me between them.

The companion piece, [part 2: Agents, plural](#), is about why I ended up working this way and the philosophical mumbo jumbo that lead me here, and both sit next to an earlier experiment on what happens when an agent dies halfway (write-up coming soon), which is the problem `marathon` exists for.

what is pi?

Pi is a minimal coding agent harness. Out of the box it gives a model a small set of tools (read, write, edit, bash) and very little else. Everything beyond that comes from things you add: skills, slash commands, extensions and packages that other people publish (or yourself, obviously). The idea is that the people using the tool are the ones who decide what it should be. This is a very different philosophy from the more walled garden approach that tools like Claude Code or Codex take, and gives power back to power users.

The mainstream tools are moving towards more capability built natively into the product, and with it more lock-in. That makes sense for them. They are big companies and lock-in is part of the business. But it reminds me of what Apple did with AirPods. You can use AirPods with an Android phone, and they work, but the experience is nowhere near what you get with an iPhone, because a whole layer of capability sits on top that only works inside the ecosystem. Pi is going the other way: open, extensible and approachable.

It is very easy to create a tool right now. Code is cheap. What is hard is standing behind a product for a long time when everyone's capacity to experiment is basically infinite. The Pi team gave a talk at some point about, more or less, hating AI, which I found very funny and also a good way to understand the people behind this tool. They are down to earth, they keep things simple and they distrust these tools by default (the same ones they are building). In the harness world they feel almost like the counterculture.

If you have never used Pi, it also has its own tutorial, which is genuinely good and worth going through before anything else in this article.

what pi profiles are

Pi solves a simple problem very well: you get to build your own agent. What it doesn't give you is a shelf to put those agents on.

Say you had a really good thing going with a particular setup. Or you want to set aside a configuration formally for one kind of task. Or you want to try a new skill or extension without it touching everything else. In plain Pi, your configuration is one directory, so every experiment changes the one agent you use for everything. The nature of Pi is extensibility and modularity, and that is great, but if you have nowhere to put the things you build, they are all a bit ephemeral.

Profiles are that shelf. Each one is a set of three kinds of pieces that you combine like Lego: skills, slash commands (which are like skills, except only you can invoke them) and extensions. You take the pieces you want for a job and leave the ones you don't. Pi is the tool, and profiles are the toolbox. Some days I need a screwdriver, some days a wrench and some days a hammer.

FIGURE 1. THE PROFILES ON MY SHELF

AT THE KEYBOARD	LEFT ALONE	EXPERIMENTS
daily one task, everything visible	marathon hours unattended, gates in code	SoL-Pi one package, nothing el
basic stock Pi plus web and subagents		pi-profiles the wrapper, no package
		alterwiki local extension work

Six profiles, grouped by the kind of work each is for. Each one is a separate Pi setup, picked with one command when Pi starts.

These are the profiles I use right now:

PROFILE	WHAT IT'S FOR	WHAT'S IN IT
daily	Interactive, back-and-forth work at the keyboard, one task at a time.	Web access, subagents, a goal tracker, a visible to-do list and a few writing and document skills. No loops and no approval modes.
basic	Plain Pi, as it ships, with web search and subagents on top. Stock behaviour with the normal built-in tools, skills and prompts.	pi-web-access and pi-subagents . No lean extras from <code>daily</code> , no approval gating, no Ralph loop, and no default tools override.
marathon	Unattended work that runs for hours: deep research, long refactors, test coverage campaigns.	A goal workflow, Ralph loops, durable task plans, an approval gate and a code-review gate, plus written rules about worktrees, budgets and when to stop.
SoL-Pi	Testing NVIDIA's SoL-Pi on real work.	SoL-Pi and nothing else.
pi-profiles	Working on the pi-profiles wrapper itself: editing the code,	No packages.

testing a change, or running Pi
with no packages at all to see
what the base config does on its
own.

alterwiki

Developing the AlterWiki
extension against my local
checkout.

The AlterWiki extension, loaded
from a local directory. That path
only exists on my machine, so I
would publish it as an npm or git
package before packing this profile
for anyone else.

The difference between **daily** and **marathon** is not just the number of tools. It is how the agent behaves. With **daily** I am in the loop the whole time. With **marathon** I might say "go and find every piece of research on this subject and tell me which ones I should read", and that could be a multi-step, multi-hour job I come back to later. That needs things I do not want anywhere near my daily work. A Ralph loop, for example, runs each iteration as its own Pi session, so Pi sessions have to be able to spawn other Pi sessions. That level of permission is fine for a supervised overnight run in a worktree. It is not something I am comfortable with in **daily**.

Most of what makes **marathon** safe to leave alone is code, not instructions. The audit loop refuses to record a clean review unless it has seen the tests pass since the last change. The Ralph loop is configured to block **git push** and to protect secrets. The profile's instructions say to work in a git worktree, to stop when a budget runs out or progress stalls, and to never declare success without running the acceptance commands fresh. The rules are written down, but the parts I rely on at three in the morning are enforced by extensions.

It also works the other way. **marathon** has a deep research workflow, and if those skills were loaded in **daily** they could trigger when I don't want them to. Deep research is its own part of the stack for me. Keeping it out of the profiles where I never want it is part of the point.

why **sol-pi** has a profile of its own

The **SoL-Pi** profile exists because of a mistake. When NVIDIA released SoL-Pi, I wanted to test it on long sessions, and my first attempt was the marathon stack plus SoL-Pi. One of the marathon extensions is pi-nolo, a permissions extension I needed for the loops. To do

its job, pi-nolo re-registers Pi's built-in `edit` tool. SoL-Pi also extends `edit`, with a parameter that lets it fuse an edit and a follow-up command into one step. Whichever extension loads first keeps the `edit` slot, and pi-nolo was winning. So SoL-Pi's fusion feature never fired on a single edit for the whole trial.

Once SoL-Pi had a clean profile, I measured it again against stock Pi 0.85.1. With a natural prompt, the model never chose to use fusion: 0 of 6 runs. On small tasks that left the profile as ObservationPack plus about 14% extra prompt overhead. When the prompt asked for fusion, it fired every time and saved a turn (4 turns down to 3, with 21.6% and 12.7% fewer tokens). The large-context runs were too noisy to call with three pairs each. That is a small measurement, but it is a real one, and I could only get it once nothing else was in the way.

I didn't know that at the time. I actually opened an issue on SoL-Pi's GitHub thinking something was wrong with their system, when it was a compatibility problem between two extensions that are each fine on their own. That trial was useless, because I had changed more than one thing at once. Now SoL-Pi lives in a profile with nothing else in it, and the profile's description says not to add anything that re-registers `edit`, `write` or `bash`.

That is the third reason I keep profiles, next to working style and context: controlled experiments. If you want to know what one extension does, you need a setup where it is the only thing that changed.

using it

The launcher is a single Bash script. Install it by putting it on your path:

```
ln -s "$PWD/bin/pi-profile" ~/.local/bin/pi-profile
```

The everyday commands:

```
pi-profile list                # every profile, what it's  
for and what it installs  
pi-profile show marathon      # one profile's full  
description  
pi-profile create work         # a new, empty profile  
pi-profile create web --from work # start from an existing  
profile's settings  
pi-profile create min --from base # start from your normal Pi
```

```
settings
pi-profile daily          # launch Pi in the 'daily'
profile
pi-profile                # pick a profile
interactively (fzf if you have it)
```

Every profile carries a `DESCRIPTION.md` : what it is for and when to use it instead of another one. The first line is the summary that `list` and the picker show. Underneath that, `list` prints what the profile actually installs, read from the profile at that moment, so the description can never claim a package the profile doesn't have.

You can also share a profile. `pack` exports the portable parts into a folder you can push to GitHub, and `install` turns any such repository back into a local profile:

```
pi-profile pack marathon ./pi-marathon
pi-profile install you/pi-marathon
pi-profile install ./pi-marathon test  # local folders work
too, to test before pushing
```

Packages are not copied. They are fetched from npm or git the first time the profile launches, the same way Pi installs them normally. That also means installing someone else's profile runs their packages, so it carries the same trust as `npm install`. Read their `settings.json` first.

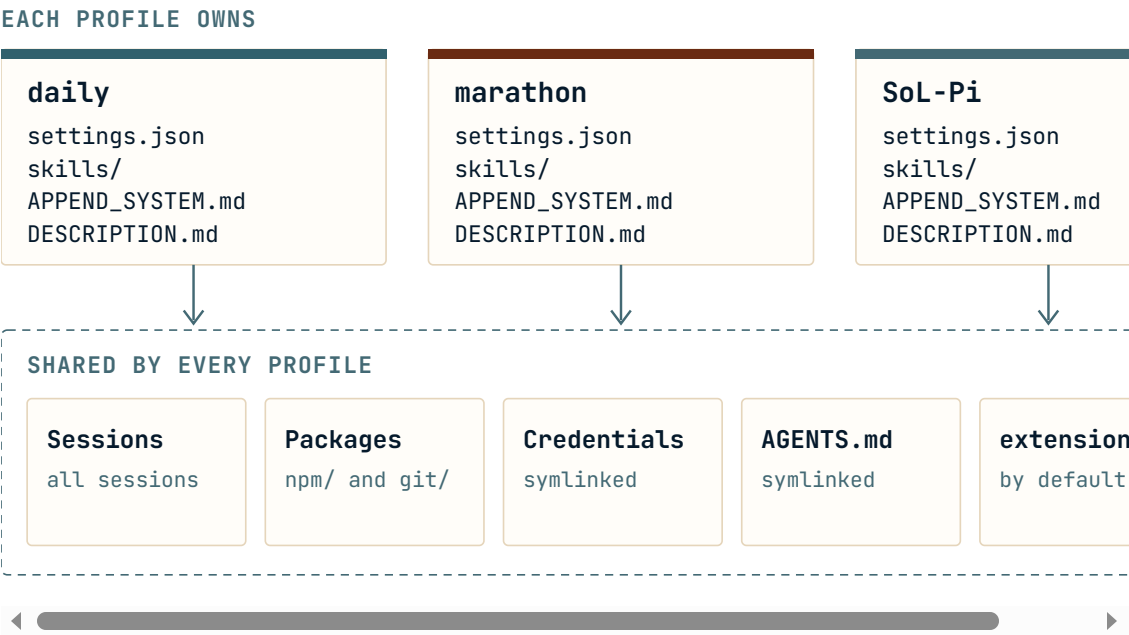
under the hood

This is the part I find most interesting, because the whole thing is small. Pi already has the two hooks the launcher needs. `PI_CODING_AGENT_DIR` tells Pi which directory to load its configuration from. `PI_CODING_AGENT_SESSION_DIR` tells it where to keep session history. Launching a profile is just:

```
export PI_CODING_AGENT_DIR="$HOME/.pi/profiles/$name"
export PI_CODING_AGENT_SESSION_DIR="$HOME/.pi/agent/sessions"
exec pi "$@"
```

Everything else in the script is about deciding which parts of a profile are its own and which parts are shared.

FIGURE 2. WHAT EACH PROFILE OWNS AND WHAT THEY SHARE



Pi reads its configuration from the directory the launcher points it at. Whatever lives in that directory is private to the profile. Everything inside the dashed box is shared by every profile, as one copy or a symlink. Extensions are shared by default because some tools install their own files there.

This is where each piece of a profile lives:

PIECE	WHERE IT LIVES	WHY
<code>settings.json</code> (packages, model, thinking level)	Each profile	This is what makes a profile different.
<code>skills/</code>	Each profile	Skills only load where you want them.
<code>APPEND_SYSTEM.md</code>	Each profile	Each profile's own instructions, added to the system prompt.

PIECE	WHERE IT LIVES	WHY
DESCRIPTION.md	Each profile	What it's for, shown by <code>list</code> and the picker.
Session history	Shared	<code>/resume</code> sees every session from every profile.
auth.json, models-store.json, trust.json	Shared, by symlink	Credentials live in one place and are never copied.
Global AGENTS.md	Shared, by symlink	My general instructions apply everywhere.
extensions/	Shared by default	Some tools install their own files here, and Pi only looks in the active directory.
Installed packages (<code>npm/</code> , <code>git/</code>)	One shared pool	One <code>pi update --extensions</code> updates every profile.

A few of those choices are worth explaining.

Sessions are shared. This is the main difference from most other profile tools, and it is an opinion, not a technicality. I am not trying to build separate agents that happen to run the same software. I want to change the working environment and keep the continuity of my work. If you would rather each profile be its own agent with its own memory, that is a reasonable choice too, and one of the packages below does exactly that. Pi keys session files by working directory, so in practice history is shared per project: I can start something in `daily`, decide it is a marathon job and resume the same thread there.

Extensions are shared by default. A private `extensions/` for each profile sounds cleaner, but it breaks things quietly. Some tools install their own files into Pi's base directory, and Pi only discovers extensions under the directory it is running from, so a profile with its own empty `extensions/` loaded none of them. If a profile really needs its own extensions, you replace the symlink with a real directory and link back in the ones you want.

Packages live in one pool. Pi installs packages under its agent directory. With several profiles that would mean several copies of everything and five updates every time

something changes. Instead, every profile's `npm/` and `git/` directories are symlinks into one pool at `~/.pi/profiles/.shared/`, and the base Pi directory points there too. Each profile still declares its own list of packages in `settings.json`; it just doesn't keep its own copy of them.

Sharing is a whitelist. `pack` copies an explicit list of files (`settings.json`, `APPEND_SYSTEM.md`, `DESCRIPTION.md`, `sol-pi.json`) and directories (`skills/`, `prompts/`, `extensions/`), and it never copies or follows a symlink. Credentials are always symlinks, so they can't end up in a bundle by accident. New config files get added to the list by name, never with a wildcard.

You can see where you are. New profiles get a tiny extension, `profile-badge`, that shows the profile name in Pi's footer. It reads `PI_CODING_AGENT_DIR`, so it stays quiet when Pi runs without the launcher.

It is also worth being clear about what the launcher does not do. It doesn't change Pi itself or override any of its behaviour. It doesn't switch anything inside a running session; you pick the profile when you start Pi. Skills in `~/.agents/skills/` are still loaded in every profile, because that is Pi's cross-harness skill location. A project's own `.pi/` folder still loads on top, as usual. And a profile is not a sandbox. It changes what the agent has available, not what it is allowed to touch on your machine. The limits in `marathon` come from its extensions and its rules, not from the profile.

other people are building this too

I'm not the only one who wanted this. There is a [Pi issue](#) proposing a built-in `--profile` flag on top of `PI_CODING_AGENT_DIR`, mostly to keep state such as credentials, sessions and settings separate between environments. People are still asking for it to be reconsidered, and as far as I can tell there is no plan for native profiles in Pi. And there are several packages that each mean something slightly different by a profile:

PROJECT	WHEN THE PROFILE CHANGES	SESSION HISTORY	WHAT IT FOCUSES ON
pi-profiles (mine)	At launch	Shared	Reusable setups for different kinds of work, and controlled experiments

PROJECT	WHEN THE PROFILE CHANGES	SESSION HISTORY	WHAT IT FOCUSES ON
<u>pi-profile-switch</u>	During a session, without restarting	Kept while switching	Choosing skills, extensions, MCP servers and tools per workflow
<u>pi-profile</u>	During a session	Kept while switching	Swapping model, skills, subagents and system prompt
<u>@sovern/pi-profile</u>	At launch	Separate per profile	Isolated profiles, each with its own memory

These come from each project's own description, and they will probably have changed by the time you read this. I don't think one of them is right and the others wrong. If you want to change modes in the middle of a session, a runtime switcher is more convenient. If you want a clear boundary around what is loaded and the same history everywhere, launching into a profile is simpler. That so many people built some version of this independently says something, though. It is in the nature of this tool that people extend it to fit the way they work.

If you want to try mine, the code and my profiles are at github.com/AdrianTJ/pi-profiles. Make it fit the way you work. That is the whole point of Pi.