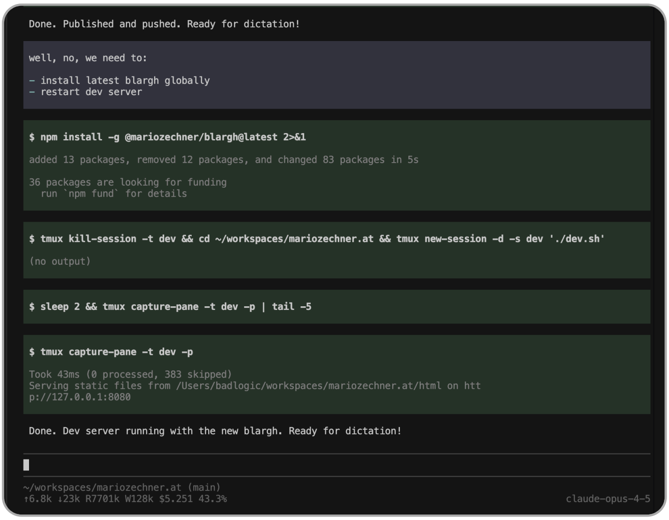
# What I learned building an opinionated and minimal coding agent

*2025-11-30*



```
Done. Published and pushed. Ready for dictation!

well, no, we need to:

- install latest blargh globally
- restart dev server

$ npm install -g @mariozechner/blargh@latest 2>&1

added 13 packages, removed 12 packages, and changed 83 packages in 5s

36 packages are looking for funding
  run `npm fund` for details

$ tmux kill-session -t dev && cd ~/workspaces/mariozechner.at && tmux new-session -d -s dev './dev.sh'

(no output)

$ sleep 2 && tmux capture-pane -t dev -p | tail -5

$ tmux capture-pane -t dev -p

Took 43ms (0 processed, 383 skipped)
Serving static files from /Users/badlogic/workspaces/mariozechner.at/html on htt
p://127.0.0.1:8080

Done. Dev server running with the new blargh. Ready for dictation!

█

~/workspaces/mariozechner.at (main)
↑6.8k ↓23k R7701k W128k $5.251 43.3%                                    claude-opus-4-5
```

## Table of contents

In the past three years, I've been using LLMs for assisted coding. If you read this, you probably went through the same evolution: from copying and pasting code into ChatGPT, to Copilot auto-completions (which never worked for me), to Cursor, and finally the new breed of coding agent harnesses like Claude Code, Codex, Amp, Droid, and opencode that became our daily drivers in 2025.

I preferred Claude Code for most of my work. It was the first thing I tried back in April after using Cursor for a year and a half. Back then, it was much more basic. That fit my workflow perfectly, because I'm a simple boy who likes simple, predictable tools. Over the past few months, Claude Code has turned into a spaceship with 80% of functionality I have no use for. The system prompt and tools also change on every release, which breaks my workflows and changes model behavior. I hate that. Also, it flickers.

I've also built a bunch of agents over the years, of various complexity. For example, Sitegeist, my little browser-use agent, is essentially a coding agent that lives inside the browser. In all that work, I learned that context engineering is paramount. Exactly controlling what goes into the model's context yields better outputs, especially when it's writing code. Existing harnesses make this extremely hard or impossible by injecting stuff behind your back that isn't even surfaced in the UI.

Speaking of surfacing things, I want to inspect every aspect of my interactions with the model. Basically no harness allows that. I also want a cleanly documented session format I can post-process automatically, and a simple way to build alternative UIs on top of the agent core. While some of this is possible with existing harnesses, the APIs smell like organic evolution. These solutions accumulated baggage along the way, which shows in the developer experience. I'm not blaming anyone for this. If tons of people use your shit and you need some sort of backwards compatibility, that's the price you pay.

I've also dabbled in self-hosting, both locally and on DataCrunch. While some harnesses like opencode support self-hosted models, it usually doesn't work well. Mostly because they rely on libraries like the Vercel AI SDK, which doesn't play nice with self-hosted models for some reason, specifically when it comes to tool calling.

So what's an old guy yelling at Claudes going to do? He's going to write his own coding agent harness and give it a name that's entirely un-Google-able, so there will never be any users. Which means there will also never be any issues on the GitHub issue tracker. How hard can it be?

To make this work, I needed to build:

- **pi-ai**: A unified LLM API with multi-provider support (Anthropic, OpenAI, Google, xAI, Groq, Cerebras, OpenRouter, and any OpenAI-compatible endpoint), streaming, tool calling with TypeBox schemas, thinking/reasoning support, seamless cross-provider context handoffs, and token and cost tracking.
- **pi-agent-core**: An agent loop that handles tool execution,

validation, and event streaming.

- **pi-tui**: A minimal terminal UI framework with differential rendering, synchronized output for (almost) flicker-free updates, and components like editors with autocomplete and markdown rendering.
- **pi-coding-agent**: The actual CLI that wires it all together with session management, custom tools, themes, and project context files.

My philosophy in all of this was: if I don't need it, it won't be built. And I don't need a lot of things.

## pi-ai and pi-agent-core

I'm not going to bore you with the API specifics of this package. You can read it all in the README.md. Instead, I want to document the problems I ran into while creating a unified LLM API and how I resolved them. I'm not claiming my solutions are the best, but they've been working pretty well throughout various agentic and non-agentic LLM projects.

## There. Are. Four. Ligh... APIs

There's really only four APIs you need to speak to talk to pretty much any LLM provider: OpenAI's Completions API, their newer Responses API, Anthropic's Messages API, and Google's Generative AI API.

They're all pretty similar in features, so building an abstraction on top of them isn't rocket science. There are, of course, provider-specific peculiarities you have to care for. That's especially true for the Completions API, which is spoken by pretty much all providers, but each of them has a different understanding of what this API should do. For example, while

OpenAI doesn't support reasoning traces in their Completions API, other providers do in their version of the Completions API. This is also true for inference engines like llama.cpp, Ollama, vLLM, and LM Studio.

For example, in openai-completions.ts:

- Cerebras, xAI, Mistral, and Chutes don't like the `store` field
- Mistral and Chutes use `max_tokens` instead of `max_completion_tokens`
- Cerebras, xAI, Mistral, and Chutes don't support the `developer` role for system prompts
- Grok models don't like `reasoning_effort`
- Different providers return reasoning content in different fields (`reasoning_content` vs `reasoning`)

To ensure all features actually work across the gazillion of providers, pi-ai has a pretty extensive test suite covering image inputs, reasoning traces, tool calling, and other features you'd expect from an LLM API. Tests run across all supported providers and popular models. While this is a good effort, it still won't guarantee that new models and providers will just work out of the box.

Another big difference is how providers report tokens and cache reads/writes. Anthropic has the sanest approach, but generally it's the Wild West. Some report token counts at the start of the SSE stream, others only at the end, making accurate cost tracking impossible if a request is aborted. To add insult to injury, you can't provide a unique ID to later correlate with their billing APIs and figure out which of your users consumed how many tokens. So pi-ai does token and cache tracking on a best-effort basis. Good enough for personal use, but not for accurate billing if you have end users consuming tokens through your service.

Special shout out to Google who to this date seem to not support tool call streaming which is extremely Google.

pi-ai also works in the browser, which is useful for building web-based interfaces. Some providers make this especially easy by supporting CORS, specifically Anthropic and xAI.

## Context handoff

Context handoff between providers was a feature pi-ai was designed for from the start. Since each provider has their own way of tracking tool calls and thinking traces, this can only be a best-effort thing. For example, if you switch from Anthropic to

OpenAI mid-session, Anthropic thinking traces are converted to content blocks inside assistant messages, delimited by `<thinking></thinking>` tags. This may or may not be sensible, because the thinking traces returned by Anthropic and OpenAI don't actually represent what's happening behind the scenes.

These providers also insert signed blobs into the event stream that you have to replay on subsequent requests containing the same messages. This also applies when switching models within a provider. It makes for a cumbersome abstraction and transformation pipeline in the background.

I'm happy to report that cross-provider context handoff and context serialization/deserialization work pretty well in pi-ai:

```
import { getModel, complete, Conte

// Start with Claude
const claude = getModel('anthropic
const context: Context = {
  messages: []
};

context.messages.push({ role: 'use
const claudeResponse = await compl
  thinkingEnabled: true
});
context.messages.push(claudeRespon

// Switch to GPT - it will see Cla
const gpt = getModel('openai', 'gp
context.messages.push({ role: 'use
const gptResponse = await complete
context.messages.push(gptResponse)

// Switch to Gemini
const gemini = getModel('google',
context.messages.push({ role: 'use
const geminiResponse = await compl

// Serialize context to JSON (for
const serialized = JSON.stringify(

// Later: deserialize and continue
const restored: Context = JSON.par
```

```
restored.messages.push({ role: 'us
const continuation = await complet
```

## We live in a multi-model world

Speaking of models, I wanted a type-safe way of specifying them in the `getModel` call. For that I needed a model registry that I could turn into TypeScript types. I'm parsing data from both OpenRouter and models.dev (created by the opencode folks, thanks for that, it's super useful) into models.generated.ts. This includes token costs and capabilities like image inputs and thinking support.

And if I ever need to add a model that's not in the registry, I wanted a type system that makes it easy to create new ones. This is especially useful when working with self-hosted models, new releases that aren't yet on models.dev or OpenRouter, or trying out one of the more obscure LLM providers:

```
import { Model, stream } from '@ma

const ollamaModel: Model<'openai-c
  id: 'llama-3.1-8b',
  name: 'Llama 3.1 8B (Ollama)',
  api: 'openai-completions',
  provider: 'ollama',
  baseUrl: 'http://localhost:11434
  reasoning: false,
  input: ['text'],
  cost: { input: 0, output: 0, cac
  contextWindow: 128000,
  maxTokens: 32000
};

const response = await stream(olla
  apiKey: 'dummy' // Ollama doesn'
});
```

Many unified LLM APIs completely ignore providing a way to abort requests. This is entirely unacceptable if you want to integrate your LLM into any kind of production system. Many unified LLM APIs also don't return partial results to you, which is kind of ridiculous. pi-ai was designed from the beginning to support aborts throughout the entire pipeline, including tool calls. Here's how it works:

```
import { getModel, stream } from '

const model = getModel('openai', '
const controller = new AbortContro

// Abort after 2 seconds
setTimeout(() => controller.abort(

const s = stream(model, {
  messages: [{ role: 'user', conte
}, {
  signal: controller.signal
});

for await (const event of s) {
  if (event.type === 'text_delta')
    process.stdout.write(event.del
  } else if (event.type === 'error
    console.log(`${event.reason ==
  }
}

// Get results (may be partial if
const response = await s.result();
if (response.stopReason === 'abort
  console.log('Partial content:',
}
```

## Structured split tool results

Another abstraction I haven't seen in any unified LLM API is splitting tool results into a portion handed to the LLM and a portion for UI display. The LLM portion is generally just text or JSON, which doesn't necessarily contain all the information you'd want to show in a UI. It also sucks hard to parse textual tool outputs and restructure them for display in a UI. pi-ai's tool implementation allows returning both content blocks for the LLM and separate content blocks for UI rendering. Tools can also return attachments like images that get attached in the native format of the respective provider. Tool arguments are automatically validated using

TypeBox schemas and AJV, with detailed error messages when validation fails:

```
import { Type, AgentTool } from '@

const weatherSchema = Type.Object(
  city: Type.String({ minLength: 1
});

const weatherTool: AgentTool<typeo
  name: 'get_weather',
  description: 'Get current weathe
  parameters: weatherSchema,
  execute: async (toolCallId, args
    const temp = Math.round(Math.r
    return {
      // Text for the LLM
      output: `Temperature in ${ar
      // Structured data for the U
      details: { temp }
    };
  }
};

// Tools can also return images
const chartTool: AgentTool = {
  name: 'generate_chart',
  description: 'Generate a chart f
  parameters: Type.Object({ data:
```

```
  execute: async (toolCallId, args
    const chartImage = await gener
    return {
      content: [
        { type: 'text', text: `Gen
        { type: 'image', data: cha
      ]
    };
  }
};
```

What's still lacking is tool result streaming. Imagine a bash tool where you want to display ANSI sequences as they come in. That's currently not possible, but it's a simple fix that will eventually make it into the package.

Partial JSON parsing during tool call streaming is essential for good UX. As the LLM streams tool call arguments, pi-ai progressively parses them so you can show partial results in the UI before the call completes. For example, you can display a diff streaming in as the agent rewrites a file.

## Minimal agent scaffold

Finally, pi-ai provides an agent loop that handles the full orchestration: processing user messages, executing tool calls, feeding results back to the LLM, and repeating until the model produces a response without tool calls. The loop also supports message queuing via a callback: after each turn, it asks for queued messages and injects them before the next assistant response. The loop emits events for everything, making it easy to build reactive UIs.

The agent loop doesn't let you specify max steps or similar knobs you'd find in other unified LLM APIs. I never found a use case for that, so why add it? The loop just loops until the agent says it's done. On top of the loop, however, pi-agent-core provides an `Agent` class with actually useful stuff: state management, simplified event subscriptions, message queuing with two modes (one-at-a-time or all-at-once), attachment handling (images, documents), and a transport abstraction that lets you run the agent either directly or through a proxy.

Am I happy with pi-ai? For the most part, yes. Like any unifying API, it can never be perfect due to leaky abstractions. But it's been used in seven different production projects and has served me extremely well.

Why build this instead of using the Vercel AI SDK? Armin's blog post mirrors my experience. Building on top of the provider SDKs directly gives me full control and lets me design the APIs exactly as I want, with a much smaller surface area. Armin's blog gives you a more in-depth treatise on the reasons for building your own. Go read that.

## pi-tui

I grew up in the DOS era, so terminal user interfaces are what I grew up with. From the fancy setup programs for Doom to Borland products, TUIs were with me until the end of the 90s. And boy was I fucking happy when I eventually switched to a GUI operating system. While TUIs are mostly portable and easily streamable, they also suck at information density. Having said all that, I thought starting with a terminal user interface for pi makes the most sense. I could strap on a GUI later whenever I felt like I needed to.

So why build my own TUI framework? I've looked into the alternatives like Ink, Blessed, OpenTUI, and so on. I'm sure they're all fine in their own way, but I definitely don't want to write my TUI like a React app. Blessed seems to be mostly unmaintained, and OpenTUI is explicitly not production ready. Also, writing my own TUI framework on top of Node.js seemed like a fun little challenge.

## Two kinds of TUIs

Writing a terminal user interface is not rocket science per se. You just have to pick your poison. There's basically two ways to do it. One is to take ownership of the terminal viewport (the portion of the terminal contents you can actually see) and treat it like a pixel buffer. Instead of pixels you have cells that contain characters with background color, foreground color, and styling like italic and bold. I call these full screen TUIs. Amp and opencode use this approach.

The drawback is that you lose the scrollback buffer, which means you have to implement custom search. You also lose scrolling, which means you have to simulate scrolling within the viewport yourself. While this is not hard to implement, it means you have to re-implement all the functionality your terminal emulator already provides. Mouse scrolling specifically always feels kind of off in such TUIs.

The second approach is to just write to the terminal like any CLI program, appending content to the scrollback buffer, only occasionally moving the "rendering cursor" back up a little within the visible viewport to redraw things like animated spinners or a text edit field. It's not exactly that simple, but you get the idea. This is what Claude Code, Codex, and Droid do.

Coding agents have this nice property that they're basically a chat interface. The user writes a prompt, followed by replies from the agent and tool calls and their results. Everything is nicely linear, which lends itself well to working with the "native" terminal emulator. You get to use all the built-in functionality like natural scrolling and search within the scrollback buffer. It also limits what your TUI can do to some degree, which I find charming because constraints make for minimal programs that just do what they're supposed to do without superfluous fluff. This is the direction I picked for pi-tui.

## Retained mode UI

If you've done any GUI programming, you've probably heard of retained mode vs immediate mode. In a retained mode UI, you build up a tree of components that persist across frames. Each component knows how to render itself and can cache its output if nothing changed. In an immediate mode UI, you redraw everything from scratch each frame (though in practice, immediate mode UIs also do caching, otherwise they'd fall apart).

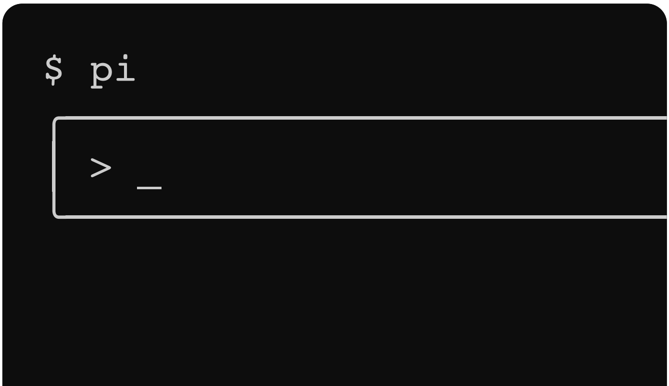
pi-tui uses a simple retained mode approach. A `Component` is just an object with a `render(width)` method that returns an array of strings (lines that fit the viewport horizontally, with ANSI escape codes for colors and styling) and an optional `handleInput(data)` method for keyboard input. A `Container` holds a list of components arranged vertically and collects all their rendered lines. The `TUI` class is itself a container that orchestrates everything.

When the TUI needs to update the screen, it asks each component to render. Components can cache their output: an assistant message that's fully streamed doesn't need to re-parse markdown and re-render ANSI sequences every time. It just returns the cached lines. Containers collect lines from all children. The TUI gathers all these lines and compares them to the lines it previously rendered for the previous component tree. It keeps a backbuffer of sorts, remembering what was written to the scrollback buffer.

Then it only redraws what changed, using a method I call differential rendering. I'm very bad with names, and this likely has an official name.

## Differential rendering

Here's a simplified demo that illustrates what exactly gets redrawn.



```
$ pi
  > _
```

```
▶ Click to start  | Lines
redrawn: 0/10
```

The algorithm is simple:

1. **First render**: Just output all lines to the terminal
2. **Width changed**: Clear screen completely and re-render everything (soft wrapping changes)
3. **Normal update**: Find the first line that differs from what's on screen, move the cursor to that line, and re-render from there to the end

There's one catch: if the first changed line is above the visible viewport (the user scrolled up), we have to do a full clear and re-render. The terminal doesn't let you write to the scrollback buffer above the viewport.

To prevent flicker during updates, pi-tui wraps all rendering in synchronized output escape sequences (`CSI ?2026h` and `CSI ?2026l`). This tells the terminal to buffer all the output and display it atomically. Most modern terminals support this.

How well does it work and how much does it flicker? In any capable terminal like Ghostty or iTerm2, this works brilliantly and you never see any flicker. In less fortunate terminal implementations like VS Code's built-in terminal, you will get some flicker depending on the time of day, your display size, your window size, and so on. Given that I'm very accustomed to Claude Code, I haven't spent any more time optimizing this. I'm happy with the little flicker I get in VS Code. I wouldn't feel at home otherwise.

And it still flickers less than Claude Code.

How wasteful is this approach? We store an entire scrollback buffer worth of previously rendered lines, and we re-render lines every time the TUI is asked to render itself. That's alleviated with the caching I described above, so the re-rendering isn't a big deal. We still have to compare a lot of lines with each other. Realistically, on computers younger than 25 years, this is not a big deal, both in terms of performance and memory use (a few hundred kilobytes for very large sessions). Thanks V8. What I get in return is a dead simple programming model that lets me iterate quickly.

## pi-coding-agent

I don't need to explain what features you should expect from a coding agent harness. pi comes with most creature comforts you're used to from other tools:

- Runs on Windows, Linux, and macOS (or anything with a Node.js runtime and a terminal)
- Multi-provider support with mid-session model switching
- Session management with continue, resume, and branching
- Project context files (AGENTS.md) loaded hierarchically from global to project-specific
- Slash commands for common operations
- Custom slash commands as markdown templates with argument support

- OAuth authentication for Claude Pro/Max subscriptions
- Custom model and provider configuration via JSON
- Customizable themes with live reload
- Editor with fuzzy file search, path completion, drag & drop, and multi-line paste
- Message queuing while the agent is working
- Image support for vision-capable models
- HTML export of sessions
- Headless operation via JSON streaming and RPC mode
- Full cost and token tracking

If you want the full rundown, read the README. What's more interesting is where pi deviates from other harness-es in philosophy and implementation.

## Minimal system prompt

Here's the system prompt:

```
You are an expert coding
assistant. You help users with
coding tasks by reading files,
executing commands, editing
code, and writing new files.

Available tools:
- read: Read file contents
- bash: Execute bash commands
- edit: Make surgical edits to
files
- write: Create or overwrite
files

Guidelines:
- Use bash for file operations
like ls, grep, find
- Use read to examine files
before editing
- Use edit for precise changes
(old text must match exactly)
- Use write only for new files
```

```
or complete rewrites
- When summarizing your actions,
output plain text directly - do
NOT use cat or bash to display
what you did
- Be concise in your responses
- Show file paths clearly when
working with files

Documentation:
- Your own documentation
(including custom model setup
and theme creation) is at:
/path/to/README.md
- Read it when users ask about
features, configuration, or
setup, and especially if the
user asks you to add a custom
model or provider, or create a
custom theme.
```

That's it. The only thing that gets injected at the bottom is your AGENTS.md file. Both the global one that applies to all your sessions and the project-specific one stored in your project directory. This is where you can customize pi to your liking. You can even replace the full system prompt if you want to. Compared to, for example, Claude Code's system prompt, Codex's system prompt, or opencode's model-specific prompts (the Claude one is a cut-down version of the original Claude Code prompt they copied).

You might think this is crazy. In all likelihood, the models have some training on their native coding harness. So using the native system prompt or something close to it like opencode would be most ideal. But it turns out that all the frontier models have been RL-trained up the wazoo, so they inherently understand what a coding agent is. There does not appear to be a need for 10,000 tokens of system prompt, as we'll find out later in the benchmark section, and as I've anecdotally found out by exclusively using pi for the past few weeks. Amp, while copying some parts of the native system prompts, seems to also do just fine with their own prompt.

## Minimal toolset

Here are the tool definitions:

```
read
  Read the contents of a file. Sup
  gif, webp). Images are sent as a
  first 2000 lines. Use offset/lim
  - path: Path to the file to read
  - offset: Line number to start r
  - limit: Maximum number of lines

write
  Write content to a file. Creates
  if it does. Automatically create
  - path: Path to the file to writ
  - content: Content to write to t

edit
  Edit a file by replacing exact t
  (including whitespace). Use this
  - path: Path to the file to edit
  - oldText: Exact text to find an
  - newText: New text to replace t

bash
```

```
    Execute a bash command in the cu
    and stderr. Optionally provide a
    - command: Bash command to execu
    - timeout: Timeout in seconds (o
```

There are additional read-only tools (grep, find, ls) if you want to restrict the agent from modifying files or running arbitrary commands. By default these are disabled, so the agent only gets the four tools above.

As it turns out, these four tools are all you need for an effective coding agent. Models know how to use bash and have been trained on the read, write, and edit tools with similar input schemas. Compare this to Claude Code's tool definitions or opencode's tool definitions (which are clearly derived from Claude Code's, same structure, same examples, same git commit flow). Notably, Codex's tool definitions are similarly minimal to pi's.

pi's system prompt and tool definitions together come in below 1000 tokens.

## YOLO by default

pi runs in full YOLO mode and assumes you know what you're doing. It has unrestricted access to your filesystem and can execute any command without permission checks or safety rails. No permission prompts for file operations or commands. No pre-checking of bash commands by Haiku for malicious content. Full filesystem access. Can execute any command with your user privileges.

If you look at the security measures in other coding agents, they're mostly security theater. As soon as your agent can write code and run code, it's pretty much game over. The only way you could prevent exfiltration of data would be to cut off all network access for the execution environment the agent runs in, which makes the agent mostly useless. An alternative is allow-listing domains, but this can also be worked around through other means.

Simon Willison has written extensively about this problem. His "dual LLM" pattern attempts to address confused deputy attacks and data exfiltration, but even he admits "this solution is pretty bad" and introduces enormous implementation complexity. The core issue remains: if an LLM has access to tools that can read private data and make network requests, you're playing whack-a-mole with attack vectors.

Since we cannot solve this trifecta of capabilities (read data, execute code, network access), pi just gives in. Everybody is running in YOLO mode anyways to get any productive work done, so why not make it the default and only option?

By default, pi has no web search or fetch tool. However, it can use `curl` or read files from disk, both of which provide ample surface area for prompt injection attacks. Malicious content in files or command outputs can influence behavior. If you're uncomfortable with full access, run pi inside a container or use a different tool if you need (faux) guardrails.

## No built-in to-dos

pi does not and will not support built-in to-dos. In my experience, to-do lists generally confuse models more than they help. They add state that the model has to track and update, which introduces more opportunities for things to go wrong.

If you need task tracking, make it externally stateful by writing to a file:

```
# TODO.md

- [x] Implement user authenticatio
- [x] Add database migrations
- [ ] Write API documentation
- [ ] Add rate limiting
```

The agent can read and update this file as needed. Using checkboxes keeps track of what's done and what remains. Simple, visible, and under your control.

## No plan mode

pi does not and will not have a built-in plan mode. Telling the agent to think through a problem together with you, without modifying files or executing commands, is generally sufficient.

If you need persistent planning across sessions, write it to a file:

```
# PLAN.md

## Goal
Refactor authentication system to

## Approach
1. Research OAuth 2.0 flows
2. Design token storage schema
3. Implement authorization server
4. Update client-side login flow
5. Add tests

## Current Step
Working on step 3 - authorization
```

The agent can read, update, and reference the plan as it works. Unlike ephemeral planning modes that only exist within a session, file-based plans can be shared across sessions, and can be versioned with your code.

Funnily enough, Claude Code now has a Plan Mode that's essentially read-only analysis, and it will eventually write a markdown file to disk. And you can basically not use plan mode without approving a shit ton of command invocations, because without that, planning is basically impossible.

The difference with pi is that I have full observability of everything. I get to see which sources the agent actually looked at and which ones it totally missed. In Claude Code, the orchestrating Claude instance usually spawns a sub-agent and you have zero visibility into what that sub-agent does. I get to see the markdown file immediately. I can edit it collaboratively with the agent. In short, I need observability for planning and I don't get that with Claude Code's plan mode.

If you must restrict the agent during planning, you can specify which tools it has access to via the CLI:

```
pi --tools read,grep,find,ls
```

This gives you read-only mode for exploration and planning without the agent modifying anything or being able to run bash commands. You won't be happy with that though.

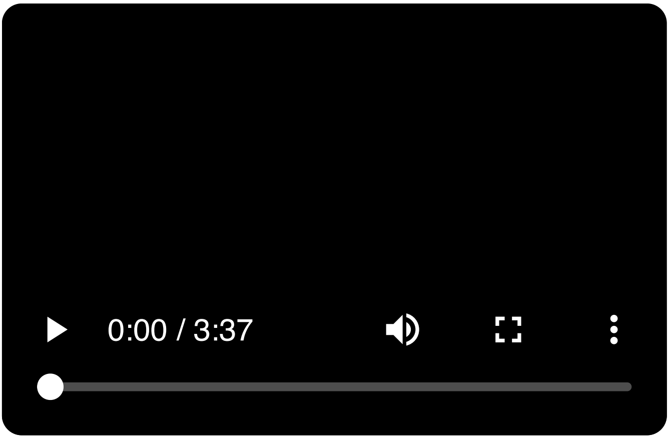
## No MCP support

pi does not and will not support MCP. I've written about this extensively, but the TL;DR is: MCP servers are overkill for most use cases, and they come with significant context overhead.

Popular MCP servers like Playwright MCP (21 tools, 13.7k tokens) or Chrome DevTools MCP (26 tools, 18k tokens) dump their entire tool descriptions into your context on every session. That's 7-9% of your context window gone before you even start working. Many of these tools you'll never use in a given session.

The alternative is simple: build CLI tools with README files. The agent reads the README when it needs the tool, pays the token cost only when necessary (progressive disclosure), and can use bash to invoke the tool. This approach is composable (pipe outputs, chain commands), easy to extend (just add another script), and token-efficient.

Here's how I add web search to pi:



I maintain a collection of these tools at github.com/badlogic/agent-tools. Each tool is a simple CLI with a README that the agent reads on demand.

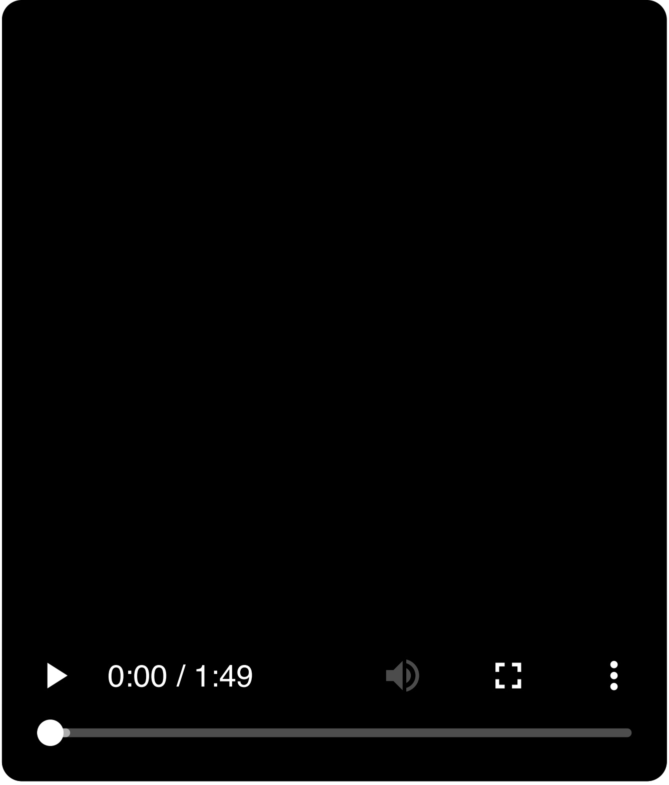
If you absolutely must use MCP servers, look into Peter Steinberger's mcporter tool that wraps MCP servers as CLI tools.

## No background bash

pi's bash tool runs commands synchronously. There's no built-in way to start a dev server, run tests in the background, or interact with a REPL while the command is still running.

This is intentional. Background process management adds complexity: you need process tracking, output buffering, cleanup on exit, and ways to send input to running processes. Claude Code handles some of this with their background bash feature, but it has poor observability (a common theme with Claude Code) and forces the agent to track running instances without providing a tool to query them. In earlier Claude Code versions, the agent forgot about all its background processes after context compaction and had no way to query them, so you had to manually kill them. This has since been fixed.

Use tmux instead. Here's pi debugging a crashing C program in LLDB:



How's that for observability? The same approach works for long-running dev servers, watching log output, and similar use cases. And if you wanted to, you could hop into that LLDB session above via tmux and co-debug with the agent. Tmux also gives you a CLI argument to list all active sessions. How nice.

There's simply no need for background bash. Claude Code can use tmux too, you know. Bash is all you need.

## No sub-agents

pi does not have a dedicated sub-agent tool. When Claude Code needs to do something complex, it often spawns a sub-agent to handle part of the task. You have zero visibility into what that sub-agent does. It's a black box within a black box. Context transfer between agents is also poor. The orchestrating agent decides what initial context to pass to the sub-agent, and you generally have little control over that. If the sub-agent makes a mistake, debugging is painful because you can't see the full conversation.

If you need pi to spawn itself, just ask it to run itself via bash. You could even have it spawn itself inside a tmux session for full observability and the ability to interact with that sub-agent directly.

---

But more importantly: fix your workflow, at least the ones that are all about context gathering. People use sub-agents within a session thinking they're saving context space, which is true. But that's the wrong way to think about sub-agents. Using a sub-agent mid-session for context gathering is a sign you didn't plan ahead. If you need to gather context, do that first in its own session. Create an artifact that you can later use in a fresh session to give your agent all the context it needs without polluting its context window with tool outputs. That artifact can be useful for the next feature too, and you get full observability and steerability, which is important during context gathering.

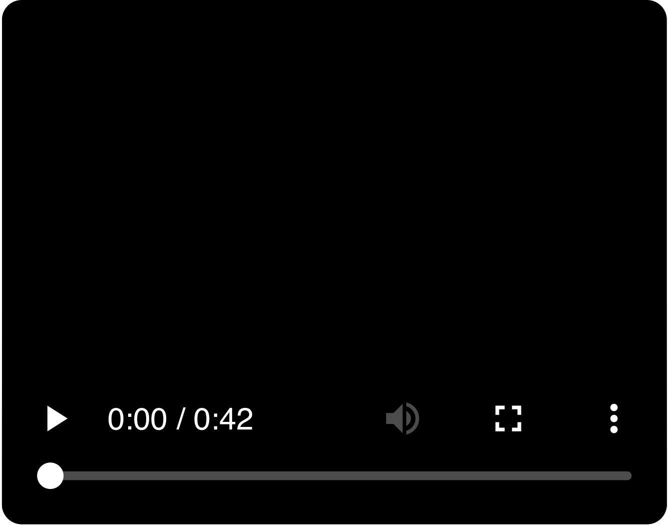
Because despite popular belief, models are still poor at finding all the context needed for implementing a new feature or fixing a bug. I attribute this to models being trained to only read parts of files rather than full files, so they're hesitant to read everything. Which means they miss important context and can't see what they need to properly complete the task.

Just look at the pi-mono issue tracker and the pull requests. Many get closed or revised because the agents couldn't fully grasp what's needed. That's not the fault of the contributors, which I truly appreciate because even incomplete PRs help me move faster. It just means we trust our agents too much.

I'm not dismissing sub-agents entirely. There are valid use cases. My most common one is code review: I tell pi to spawn itself with a code review prompt (via a custom slash command) and it gets the outputs.

```
---
description: Run a code review sub
---
Spawn yourself as a sub-agent via

Use `pi --print` with appropriate
use `--provider` and `--model` acc

Pass a prompt to the sub-agent ask
- Bugs and logic errors
- Security issues
- Error handling gaps

Do not read the code yourself. Let

Report the sub-agent's findings.
```

And here's how I use this to review a pull request on GitHub:



With a simple prompt, I can select what specific thing I want to review and what model to use. I could even set thinking levels if I wanted to. I can also save out the full review session to a file and hop into that in another pi session if I wanted. Or I can say this is an ephemeral session and it shouldn't be saved to disk. All of that gets translated into a prompt that the main agent reads and based on which it executes itself again via bash. And while I don't get full observability into the inner workings of the sub-agent, I get full observability on its output. Something other harnesses don't really provide, which makes no sense to me.

Of course, this is a bit of a simulated use case. In reality, I would just spawn a new pi session and ask it to review the pull request, possibly pull it into a branch locally. After I see its initial review, I give my own review and then we work on it together until it's good. That's the workflow I use to not merge garbage code.

Spawning multiple sub-agents to implement various features in parallel is an anti-pattern in my book and doesn't work, unless you don't care if your codebase devolves into a pile of garbage.

## Benchmarks

I make a lot of grandiose claims, but do I have numerical proof that all the contrarian things I say above actually work? I have my lived experience, but that's hard to transport in a blog post and you'd just have to believe me. So I created a Terminal-Bench 2.0 test run for pi with Claude Opus 4.5 and let it compete against Codex, Cursor, Windsurf, and other coding harnesses with their respective native models. Obviously, we all know benchmarks aren't representative of real-world performance, but it's the best I can provide you as a sort of proof that not everything I say is complete bullshit.

I performed a complete run with five trials per task, which makes the results eligible for submission to the leaderboard. I also started a second run that only runs during CET because I found that error rates (and consequently benchmark results) get worse once PST goes online. Here are the results for the first run:

---

And here's pi's placement on the current leaderboard as of December 2nd, 2025:

---

And here's the results.json file I've submitted to the Terminal-Bench folks for inclusion in the leaderboard. The bench runner for pi can be found in this repository if you want to reproduce the results. I suggest you use your Claude plan instead of pay-as-you-go.

Finally, here's a little glimpse into the CET-only run:

---

This is going to take another day or so to complete. I will update this blog post once that is done.

Also note the ranking of Terminus 2 on the leaderboard. Terminus 2 is the Terminal-Bench team's own minimal agent that just gives the model a tmux session. The model sends commands as text to tmux and parses the terminal output itself. No fancy tools, no file operations, just raw terminal interaction. And it's holding its own against agents with far more sophisticated tooling and works with a diverse set of models. More evidence that a minimal approach can do just as well.

## In summary

Benchmark results are hilarious, but the real proof is in the pudding. And my pudding is my day-to-day work, where pi has been performing admirably. Twitter is full of context engineering posts and blogs, but I feel like none of the harnesses we currently have actually let you do context engineering. pi is my attempt to build myself a tool where I'm in control as much as possible.

I'm pretty happy with where pi is. There are a few more features I'd like to add, like compaction or tool result streaming, but I don't think there's much more I'll personally need. Missing compaction hasn't been a problem for me personally. For some reason, I'm able to cram hundreds of exchanges between me and the agent into a single session, which I couldn't do with Claude Code without compaction.

That said, I welcome contributions. But as with all my open source projects, I tend to be dictatorial. A lesson I've learned the hard way over the years with my bigger projects. If I close an issue or PR you've sent in, I hope there are no hard feelings. I will also do my best to give you reasons why. I just want to keep this focused and maintainable. If pi doesn't fit your needs, I implore you to fork it. I truly mean it. And if you create something that even better fits my needs, I'll happily join your efforts.

I think some of the learnings above transfer to other harnesses as well. Let me know how that goes for you.