Running pi Locally on Apple Silicon

A beginner's guide to Mario Zechner's minimal coding agent — local models via MLX and LM Studio, without the cloud

Thomas Bartz-Beielstein

2026-10-22

Table of contents

Preface **1**
Who this is for . . . 1
What is pi, in one sentence? . . . 1
Roadmap through the chapters . . . 1
How to read these notes . . . 2
A note on sources . . . 2
Colophon . . . 2

1 Introduction to PI **3**
1.1 What pi is . . . 3
1.2 Why it exists: a world of slop . . . 3
1.3 Why this matters for *local* models . . . 3
1.4 How pi, the model server, and the model fit together . . . 4
1.5 The mental model . . . 5
1.6 Where pi runs . . . 5

2 Installing a local backend **7**
2.1 Why Apple Silicon . . . 7
2.2 MLX-LM — running Hugging Face models . . . 8
2.2.1 Generate and chat directly . . . 8
2.2.2 Serve an OpenAI-compatible endpoint . . . 9
2.3 LM Studio — MLX with tool-calling for pi . . . 9

3 Installing & configuring pi **13**
3.1 Prerequisites . . . 13
3.2 Install . . . 13
3.3 What you just installed . . . 13
3.4 The configuration directory . . . 14
3.5 The bundled documentation . . . 14
3.6 Keeping pi up to date . . . 14

4 Pointing pi at a local model **17**
4.1 The idea . . . 17
4.2 Minimal configuration . . . 17
4.3 Adding the MLX provider . . . 18
4.4 Confirm pi sees the model . . . 18
4.5 Choosing and switching models . . . 19

5 Your first agentic session — the four tools **21**
5.1 The whole agent in four tools . . . 21
5.2 Two ways to run . . . 21
5.3 A verified first task . . . 21
5.4 Proving the agent really runs commands . . . 22
5.5 A safety word before you go further . . . 22
5.6 Handy input syntax . . . 22

6 Context files & sessions **25**
6.1 Telling pi how to work: `AGENTS.md` . . . 25
6.2 Replacing the system prompt: `SYSTEM.md` . . . 25
6.3 Sessions: save, continue, resume, branch . . . 25
6.4 Extending pi (briefly) . . . 26

7 Headless "server" mode (RPC) 27
7.1 "PI Server" demystified . . . 27
7.2 The crucial gotcha . . . 27
7.3 A mental model for driving rpc . . . 27
7.4 When to use which . . . 28

8 Troubleshooting: successes & failures 29
8.1 What worked well . . . 29
8.2 Failures and fixes . . . 29
8.2.1 `--mode json` / `--mode rpc` produces no output . . . 29
8.2.2 Long sessions get truncated / the model "forgets" . . . 29
8.2.3 A reasoning model errors or behaves oddly . . . 29
8.2.4 The first request is very slow . . . 30
8.2.5 The model ignores the tools entirely . . . 30
8.2.6 `mlx_lm.server` runs but pi's tools never fire . . . 30
8.2.7 LM Studio: `Invalid passkey`, or `lms get` cannot find a model . . . 30
8.2.8 Picking a model for your RAM . . . 30
8.3 A note on security . . . 30

References 33

Preface

These notes are a hands-on, beginner-friendly guide to running pi, Mario Zechner's deliberately minimal terminal AI (artificial intelligence) coding agent, against a local large language model (LLM) served by LM Studio or the MLX-LM server, on an Apple-Silicon Mac. The whole point is to drive a capable coding agent without sending your code or prompts to a cloud provider and without an API (application programming interface) key or subscription: the model runs on your own machine.

Everything in this book was verified on a real machine (an Apple M4 Max with 128 GB of unified memory running macOS, arm64) using pi `0.79.9`, LM Studio with its `lms` command-line interface (CLI), and mlx-lm `0.31.3`. When these notes were ported into the spotregen repository (2026-10-22), the current releases were pi `1.4.0` and mlx-lm `0.32.2`; the commands were not re-run against those versions. Where a command or configuration is shown, it is the command that actually worked, and the troubleshooting chapter records what went wrong along the way.

Who this is for

You are comfortable in a terminal and have used Git, but you have no prior experience with pi and perhaps none with local LLMs. You do not need to know TypeScript (pi is written in it, but you will not write any). You do need a Mac with Apple Silicon: both backends in this book build on Apple's MLX framework, which targets exactly that hardware.

What is pi, in one sentence?

pi is a small, transparent agent harness: it gives a language model exactly four tools (`read`, `write`, `edit`, and `bash`) and a short system prompt, then lets the model do the work in a loop. That is deliberately less than tools like Claude Code offer; Chapter 1 explains why "less" is the point.

Roadmap through the chapters

The chapters follow the order you would actually take to get from zero to a working local coding agent.

Chapter	You will…	Outcome
01	Understand what pi is and why it exists	The mental model
02	Install a local backend (MLX-LM or LM Studio), download a model	A local model server pi can talk to
03	Install the pi CLI	`pi --version` works
04	Point pi at your local model via `~/.pi/agent/models.json`	pi lists your local models
05	Run your first agentic session	pi writes & runs code locally
06	Steer pi with `AGENTS.md` / `SYSTEM.md` and use sessions	Reproducible, project-aware runs
07	Drive pi headlessly via remote procedure call (RPC), its "server" mode	Programmatic and editor integration
08	Diagnose the common failure modes	A working, debuggable setup

How to read these notes

Each chapter opens with a short learning-objective box and ends with a small set of exercises. Commands are shown in fenced blocks you can paste into your terminal; the agent-facing examples assume the local setup from chapters 02–04.

Conventions

- `$` at the start of a line denotes a shell prompt; do not type the `$`.
- Local model commands assume LM Studio is serving on `http://localhost:1234` (and the MLX server, where used, on `http://localhost:8080`).
- The primary example model is `qwen3-coder-30b-a3b-instruct` (an MLX model, good at tool-calling); `openai/gpt-oss-20b` is shown as a reasoning-model alternative.

A note on sources

pi is young and moving fast, so this book leans on primary sources: the pi repository and the docs bundled with the npm package, the author's blog at https://mariozechner.at, the talk *"Building pi in a World of Slop"* (Zechner 2026a), and the MLX-LM and LM Studio documentation. Academic background on agentic coding assistants and local inference is collected in the References.

Colophon

These notes are maintained inside the spotregen repository under `docs/pi-documentation` and are rendered with `make pi-doc` (HTML and PDF) from its Makefile. They are prose with displayed, non-executed command snippets, so no Python kernel is involved in the build.

1 Introduction to PI

> **Learning objective**
>
> You can explain what pi is, why its author built it, and the mental model behind it, enough to know *why* the later chapters do what they do.

1.1 What pi is

pi is a small command-line AI (artificial intelligence) coding agent, written by Mario Zechner (known online as *badlogic*, the creator of libGDX). You run it in your terminal, tell it what you want in plain language, and it reads your files, writes and edits code, and runs shell commands to get the job done. In that sense it belongs to the same family as tools like Claude Code or Codex.

What sets pi apart is what it leaves out. pi is built around a single conviction: a modern coding agent does not need to be large to be good. It ships with only four tools (`read`, `write`, `edit`, and `bash`) and a system prompt under a thousand tokens (Zechner 2025). There is no built-in plan mode, no to-do system, no sub-agents, no MCP (Model Context Protocol) layer, no permission popups. Anything extra is an opt-in extension.

1.2 Why it exists: a world of slop

The motivation is captured in the title of the author's talk, *"Building pi in a World of Slop"* (Zechner 2026a), and stated plainly in his write-up: existing agents had grown bloated: *"Claude Code has turned into a spaceship with 80% of functionality I have no use for"* (Zechner 2025). Zechner wanted four things those tools made hard:

1. Control over exactly what the model sees in its context.
2. Observability: a clean, inspectable record of every interaction.
3. Portability across providers, including self-hosted and local models.
4. Minimalism: a predictable tool you can reason about end to end.

The bet behind the minimalism is interesting in its own right: today's frontier models have been so heavily trained on coding-agent behaviour that they already "know" what `read`, `write`, `edit`, and `bash` are for. Spelling everything out in a giant system prompt buys little and costs context. Strip it back, and a capable model fills the gap (Zechner 2025).

1.3 Why this matters for *local* models

That portability is exactly what this book exploits. Because pi's model layer (`pi-ai`) speaks to any OpenAI-compatible endpoint, and because LM Studio and the MLX server expose such an endpoint for models running on your own Mac, you can swap Claude for a local model by editing one configuration file (Chapter 4). The agent loop, the tools, and the workflow stay identical; only the model's address changes from a cloud URL (uniform resource locator) to a local one, whether LM Studio's `http://localhost:1234` or the MLX server's `:8080` (Chapter 2).

Running locally has real consequences. Your prompts and source never leave the machine; there is no per-token bill; you can work offline. The trade-off is that an open model on your laptop is generally less capable than a frontier cloud model, and you pay in memory and latency instead of dollars, themes we make concrete in Chapter 2 and Chapter 8.

The broader research context is rich: autonomous agents that resolve real software-engineering tasks end to end (Jimenez et al. 2024; Zhang et al. 2024), surveys that map the large-language-model (LLM) landscape for code and for agents (Hou et

al. 2024; Wang et al. 2024), the tool-use and reason-then-act mechanisms that make an agent loop possible at all (Schick et al. 2023; Yao et al. 2023), and the code-generation evaluations used to benchmark these systems (Chen et al. 2021), all collected in the References.

1.4 How pi, the model server, and the model fit together

It helps to separate three things that are easy to conflate. pi is the *agent*: the program in your terminal that runs the loop, holds the conversation and the four tools, and decides what to do with the model's replies. It contains no model of its own. A *model server* (LM Studio or the MLX server) is a long-running process that loads a model's weights into memory and answers inference requests over the Hypertext Transfer Protocol (HTTP). The *model*, or LLM, is the set of trained weights the server loads, for example `qwen3-coder-30b-a3b-instruct` or `gpt-oss-20b`. One server can hold several models and serves whichever a request names.

The seam between pi and the server is a single stable contract: the OpenAI Chat Completions HTTP API (application programming interface). pi's model layer (`pi-ai`) sends a POST to `…/v1/chat/completions` carrying the conversation and the tool definitions; the server runs the named model and returns the assistant's reply, which is either text or a request to call one of pi's tools. Because that contract holds whether the endpoint is LM Studio's `http://localhost:1234/v1`, the MLX server's `:8080/v1`, or a cloud provider, pi neither knows nor cares which model produced an answer. Changing the model means editing a name in `models.json`; changing the backend means editing a URL. Nothing in the agent loop moves. Not every model or server implements the tool-calling part of this protocol (Chapter 8), but the request and response shape is the same regardless.

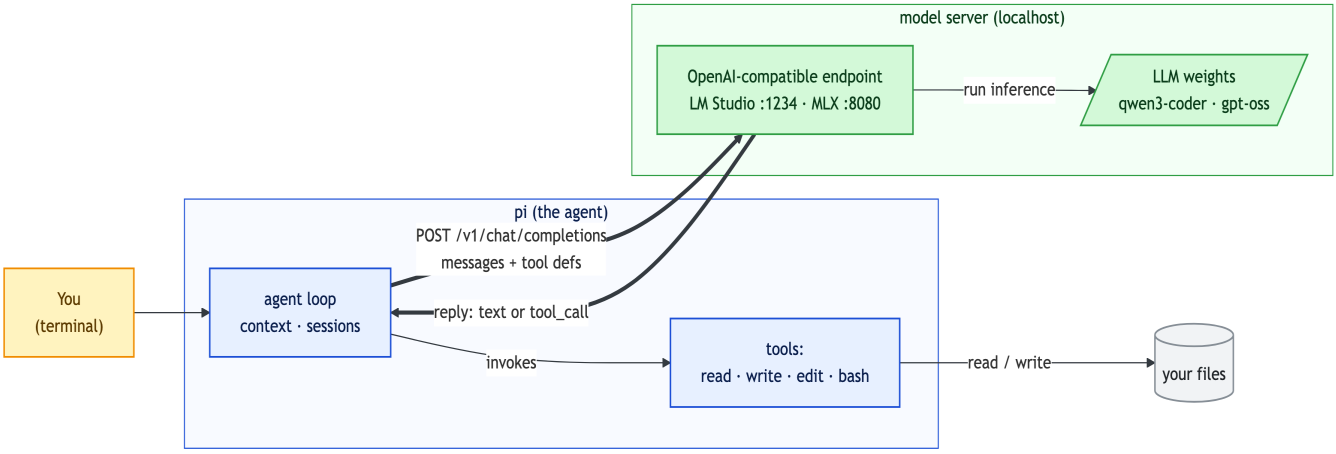


Figure 1.1: The three layers and one request/response cycle. pi (the agent) talks to a local model server over the OpenAI Chat Completions API; the server runs the chosen LLM and returns either text or a tool call, which pi executes against your files before looping.

A concrete exchange makes the cycle tangible. Suppose `models.json` points pi at LM Studio, which is serving a Qwen coder model:

```
{
  "providers": {
    "lmstudio": {
      "baseUrl": "http://localhost:1234/v1",
      "api": "openai-completions",
      "apiKey": "lmstudio",
      "models": [{ "id": "qwen3-coder-30b-a3b-instruct" }]
    }
  }
}
```

When you ask pi to "create hello.txt containing Hi", pi turns the request into a POST to `http://localhost:1234/v1/chat/completion` whose JSON (JavaScript Object Notation) body carries the conversation and the tool definitions (abbreviated here):

```
{
  "model": "qwen3-coder-30b-a3b-instruct",
  "messages": [
    {"role": "system", "content": "You are pi, a coding agent…"},
    {"role": "user",   "content": "create hello.txt containing Hi"}
  ],
  "tools": [
    {"type": "function", "function": {"name": "write", "description": "Write a file"}},
    {"type": "function", "function": {"name": "bash",  "description": "Run a shell command"}}
  ]
}
```

LM Studio runs the model, which replies not with prose but with a request to use a tool:

```
{
  "choices": [{
    "finish_reason": "tool_calls",
    "message": {
      "role": "assistant",
      "tool_calls": [{
        "type": "function",
        "function": {
          "name": "write",
          "arguments": "{\"path\": \"hello.txt\", \"content\": \"Hi\"}"
        }
      }]
    }
  }]
}
```

pi, not the model, executes that `write` call on your filesystem, appends the outcome to the conversation, and POSTs again. The model now sees that the file exists and replies with plain text such as "Created hello.txt.", which carries no tool call, so pi ends the turn. Those two steps are the whole agent loop: the server and the model only ever *propose* actions, while pi is what actually touches your machine.

1.5 The mental model

Hold these five ideas and the rest of the book follows:

1. Harness, not magic. pi is a loop: send the model the conversation and the tool definitions; the model replies with text or a tool call; pi runs the tool and feeds the result back; repeat.
2. Four tools. Everything the agent does is `read`, `write`, `edit`, or `bash` (Chapter 5).
3. Context files. `AGENTS.md` adds project rules; `SYSTEM.md` replaces the system prompt (Chapter 6).
4. Sessions. Every run is saved as a branchable tree you can resume and export (Chapter 6).
5. Pluggable models. The model is just a configured endpoint, local or cloud (Chapter 4).

1.6 Where pi runs

pi can run as an interactive terminal user interface (UI), as a one-shot non-interactive command (`-p`, great for scripts), or as a headless remote-procedure-call (RPC) process that another program drives over stdin/stdout, the closest thing to a "pi server" (Chapter 7).

Exercises

1. In your own words, list one advantage and one disadvantage of pi's four-tools-only design compared with a feature-rich agent.
2. Watch the first ten minutes of *"Building pi in a World of Slop"* and note one design decision you agree with and one you would question.
3. Before reading on, predict which parts of a cloud agent's workflow would change when you switch to a local model, then check your prediction against Chapter 8.
4. Using Figure 1.1, trace the two-step `hello.txt` exchange and mark the single arrow where pi, not the model, changes a file on disk. Which layer would you swap to run the same task on `gpt-oss-20b` instead of `qwen3-coder-30b-a3b-instruct`?

Solutions

1. An advantage is predictability and a small, inspectable surface: with only `read`, `write`, `edit`, and `bash` plus a sub-thousand-token system prompt, you can reason about exactly what the model sees and does, and you spend almost no context on tool descriptions. A disadvantage is that anything beyond those four tools (plan mode, a to-do system, sub-agents, MCP integrations, permission popups) is not built in and must be added as an opt-in extension, so a feature-rich agent gives you more out of the box.
2. Open-ended, so several answers are valid. A representative answer: I agree with the bet that frontier models already "know" what `read`, `write`, `edit`, and `bash` are for, so a giant system prompt buys little; this keeps context lean. I would question whether dropping permission popups and a built-in plan mode is safe for beginners, since `bash` can run arbitrary commands and a confirmation step would reduce risk.
3. Prediction check: very little of the workflow changes. The agent loop, the four tools, sessions, and context files stay identical; only the model's address moves from a cloud URL to a local one in the configuration. What does change in practice is capability, memory and latency cost instead of a per-token bill, and the chance that a given local model or server does not fully implement tool-calling, the failure modes covered in Chapter 8.
4. Step one: `POST /v1/chat/completions` carries the conversation and tool defs to the server; the server runs the model and returns a `write` tool_call. pi (not the model) executes that call, which is the `TOOLS --> FS ("your files")` arrow in Figure 1.1, the only point where a file on disk actually changes. Step two: pi POSTs again, the model now sees the file exists and replies with plain text and no tool call, so pi ends the turn. To run the same task on `gpt-oss-20b`, you swap only the model: change the model name in `models.json`. The server (LM Studio) and the agent stay the same.

2 Installing a local backend

> **Learning objective**
>
> You can stand up a local model server on an Apple-Silicon Mac in two ways (the MLX-LM server and LM Studio), explain why unified memory makes Apple Silicon a good fit, and you know which backend can actually drive pi's tool-using agent loop and which is better kept for plain text generation.

pi does not care *what* serves the model, only that it speaks the OpenAI Chat Completions protocol over the Hypertext Transfer Protocol (HTTP) (Chapter 4). On Apple Silicon two MLX-based local backends expose exactly that endpoint, and this book uses both:

Backend	Endpoint	Models from	Best job in this book
`mlx_lm.server`	`http://localhost:8080/v1`	Hugging Face (MLX)	generation-only workloads
LM Studio	`http://localhost:1234/v1`	Hugging Face (MLX/GGUF)	pi's agent loop (tool-calling)

The split in the last column is the single most important fact in this chapter, and Section 2.2 explains why: the bare MLX server serves text well but does not return *structured* tool calls, so pi's agent loop needs LM Studio.

2.1 Why Apple Silicon

Apple-Silicon Macs are unusually good at local inference because the central processing unit (CPU) and the graphics processing unit (GPU) share one pool of unified memory. There is no separate "VRAM" (video RAM) to copy weights into: the backend offloads the model to the GPU (via Metal) and the model's memory simply comes out of system random-access memory (RAM). The practical rule of thumb is therefore blunt:

> A model needs roughly its on-disk size in free RAM to run comfortably.

On the 128 GB machine used for this book that means even 40+ GB models load entirely on the GPU. On an 8–16 GB Mac you should stay with 3B–8B models.

Table 2.1 lists the models worth running on a 128 GB machine such as the Apple M5 Max as of July 2026.[1] Each entry is a ready-converted MLX repository under `mlx-community/…` on Hugging Face, and the disk column is the actual download size, which by the rule of thumb above is also the memory the loaded model occupies.

Table 2.1: Recommended local models for a 128 GB Apple-Silicon Mac (Apple M5 Max), July 2026.

Model (`mlx-community/…`)	Parameters	Quantization	Disk	Best for
`Qwen3.5-122B-A10B-4bit`	122 B MoE, 10 B active	4-bit	70 GB	strongest general model that fits; multimodal
`gpt-oss-120b-MXFP4-Q8`	117 B MoE, 5.1 B active	MXFP4	63 GB	deep reasoning with tool use

[1] Recommendations cross-checked against community guides for M5-Max-class machines (https://apxml.com/posts/best-local-llms-apple-silicon-mac, https://insiderllm.com/guides/best-local-llms-mac-2026/); parameter counts and download sizes verified against the Hugging Face API on 2026-07-16.

Model (`mlx-community/…`)	Parameters	Quantization	Disk	Best for
`Qwen3-Coder-Next-4bit`	80 B MoE, 3 B active	4-bit	45 GB	agentic coding; pi's tool loop; 262 K context
`Qwen3-Coder-30B-A3B-Instruct-4bit`	30 B MoE, 3 B active	4-bit	17 GB	this book's primary example model
`Devstral-Small-2-24B-Instruct-2512-4bit`	24 B dense	4-bit	15 GB	lightweight agentic coding
`gpt-oss-20b-MXFP4-Q8`	21 B MoE, 3.6 B active	MXFP4	12 GB	this book's reasoning alternative

The frontier open-weight releases of 2026 remain out of reach even here: Kimi-K2.5 needs 658 GB and GLM-5.2 roughly 400 GB in 4-bit, an order of magnitude beyond a laptop. The largest three models in Table 2.1 are the practical ceiling, and together with the two coder models they cover every task in this book: the coder models drive pi's agent loop, the general models handle plain generation.

Two advances make this practical at all. Quantization compresses model weights to 4–8 bits with little quality loss, which is the reason a multi-billion-parameter model fits on a laptop (Zhu et al. 2024; Lin et al. 2025). And research on inference under tight memory budgets (including Apple's own "LLM in a Flash" (Alizadeh et al. 2024)) pushes the envelope further, while reproducible benchmarks now quantify the performance–energy trade-offs of Apple Silicon for machine learning (Sierra-Herrera et al. 2026). Apple's MLX framework (Hannun et al. 2023) is a native array framework optimized for exactly this unified-memory hardware, and it is what both backends in this chapter build on.

2.2 MLX-LM — running Hugging Face models

MLX-LM (Apple Machine Learning Research 2026) is the language-model toolkit built on Apple's MLX framework (Hannun et al. 2023). It pulls models directly from the Hugging Face hub (typically the `mlx-community/*` repositories, which are already converted and quantized for MLX) and is often faster than a GGUF (GPT-Generated Unified Format) runtime on M-series hardware. Install it into the book's uv environment (see the README) or any virtualenv:

```
uv add mlx-lm      # or: pip install mlx-lm
```

2.2.1 Generate and chat directly

The quickest way to use a model needs no server at all. `mlx_lm.generate` runs a single prompt and exits; `mlx_lm.chat` opens an interactive read–eval–print loop (REPL). Both download the model from Hugging Face on first use:

```
# one-shot generation
mlx_lm.generate --model mlx-community/Mistral-Small-24B-Instruct-2501-4bit \
  --prompt "Erkläre MoE-Modelle"

# interactive chat
mlx_lm.chat --model mlx-community/Qwen3-30B-A3B-4bit
```

On the test machine, a one-shot run against a smaller coder model produced real output and reported its own throughput:

```
$ uv run mlx_lm.generate --model mlx-community/Qwen2.5-Coder-7B-Instruct-4bit \
    --prompt "Erkläre MoE-Modelle in einem Satz." --max-tokens 70
==========
MoE-Modelle sind KI-Modelle, die mehrere Untermodelle enthalten und bei der
Vorhersage jeweils das Ergebnis eines geeigneten Untermodells verwenden.
```

```
==========
Prompt: 42 tokens, 213.866 tokens-per-sec
Generation: 47 tokens, 108.851 tokens-per-sec
Peak memory: 4.440 GB
```

Download economics — Hugging Face

Public `mlx-community/*` repositories download anonymously over the fast Hugging Face content delivery network (CDN), with no token needed. A Hugging Face token is required only for *gated* repos (the official `meta-llama/*`, `google/gemma-*`, and some `mistralai/*`).

2.2.2 Serve an OpenAI-compatible endpoint

To let pi (or any OpenAI client) reach an MLX model, start the bundled server:

```
mlx_lm.server --model mlx-community/Qwen2.5-Coder-7B-Instruct-4bit --port 8080
```

It serves the OpenAI API (application programming interface) at `http://localhost:8080/v1` (default host and port are `127.0.0.1:8080`). Confirm it:

```
curl -s http://localhost:8080/v1/models
# {"object": "list", "data": [{"id": "mlx-community/Qwen2.5-Coder-7B-Instruct-4bit", ...}]}
```

mlx_lm.server does not emit structured tool calls

Plain chat and JSON (JavaScript Object Notation) generation through `mlx_lm.server` work well, which makes it a fine backend for generation-only workloads (spotregen's `local` driver, for instance, talks to exactly such an endpoint; see Regeneration drivers). But as of mlx-lm 0.31.3 the server does not return OpenAI-style `tool_calls`: when the model decides to call a tool, the server hands back the attempt as ordinary assistant *text* (a fenced JSON block) with `tool_calls: null`. A real pi run against it shows this clearly: asked to create a file, the model emitted

```
{
  "name": "bash",
  "arguments": { "command": "echo hi > marker.txt" }
}
```

as text, and `marker.txt` was never created. Because pi (like any tool-calling client) needs *structured* tool calls to dispatch a tool, pi's agent loop cannot run through `mlx_lm.server`. Use it for generation, and use LM Studio (next) to drive pi.

2.3 LM Studio — MLX with tool-calling for pi

LM Studio (LM Studio 2026) runs the same `mlx-community` models but, unlike the bare MLX server, its OpenAI endpoint *parses the model's tool attempt into structured `tool_calls`*, so pi's agent loop works. Install the app and its command-line interface (CLI), `lms`:

```
brew install --cask lm-studio
lms server start                 # OpenAI API on http://localhost:1234/v1
```

Fetch a tool-capable MLX model from Hugging Face and load it:

```
lms get "https://huggingface.co/mlx-community/Qwen3-Coder-30B-A3B-Instruct-4bit"
lms load qwen3-coder-30b-a3b-instruct
lms ps                          # shows the loaded model, its context and device
```

Use the full Hugging Face URL with `lms get`

The bare `author/repo` form fails for arbitrary community repos: LM Studio lowercases it and searches its own catalogue, reporting `artifact … does not exist`. Pass the full `https://huggingface.co/…` URL (uniform resource locator) instead. If `lms` rejects you with `Invalid passkey for lms CLI client`, a stale copy in `~/.lmstudio/bin` is shadowing the app; re-bootstrap from the installed bundle (Chapter 8).

The 8192 context trap

`lms ps` also reports the loaded model's `CONTEXT`. LM Studio loads models with an 8192-token context window by default, regardless of what the model technically supports. For short agent tasks this is fine, but long sessions will silently truncate. Reload the model with a larger context window for real agent work. We return to this in Chapter 8.

With the model loaded, pi drives it through the `lmstudio` provider (Chapter 4). The agent loop fires correctly: asked to create a file, pi executed the `bash` tool and the file appeared on disk:

```
$ pi -p --offline --provider lmstudio --model qwen3-coder-30b-a3b-instruct \
    "Use the bash tool to run exactly: echo hi > marker.txt - then say done."
Done. The command `echo hi > marker.txt` has been executed successfully, creating a
file named `marker.txt` with the content "hi".
$ cat marker.txt
hi
```

This is the same uncheatable, on-disk proof used in Chapter 5, and it is the contrast that matters: the identical prompt left no file behind through `mlx_lm.server`.

Exercises

1. Run `mlx_lm.generate` with a short prompt against a small `mlx-community` model and read off the reported prompt and generation throughput (tokens per second) and peak memory. How does the peak memory compare with the model's download size?
2. Start `mlx_lm.server` and send it the `curl …/v1/models` request, then a `/v1/chat/completions` request with a `tools` payload. Confirm the model's tool attempt comes back as text with `tool_calls: null`.
3. Load a model in LM Studio and check its `CONTEXT` with `lms ps`. LM Studio defaults to 8192; raise it (reload with a larger context) and re-check.

Solutions

1. `mlx_lm.generate` prints the generated text between `==========` markers, followed by three self-reported metrics: prompt tokens-per-sec, generation tokens-per-sec, and peak memory. Peak memory lands close to (slightly above) the model's on-disk size, because the quantized weights are loaded into unified memory in full, plus a small overhead for the key–value (KV) cache and activations. This is the rule of thumb from this chapter made visible: a model needs roughly its on-disk size in free RAM.

2. Expected observation: `curl …/v1/models` returns a JSON list with the loaded model's id under `data[].id`. The `/v1/chat/completions` request with a `tools` payload comes back with the model's tool attempt as ordinary assistant text (a fenced JSON block naming the tool and arguments) while `tool_calls` is `null`. As of mlx-lm 0.31.3 the bare server does not emit structured tool calls, so no tool is dispatched and any file the attempt describes is not created. This is why pi's agent loop needs LM Studio rather than `mlx_lm.server`.

```
curl -s http://localhost:8080/v1/models
curl -s http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "mlx-community/Qwen2.5-Coder-7B-Instruct-4bit",
    "messages": [{"role": "user", "content": "create marker.txt with hi"}],
    "tools": [{"type": "function", "function": {"name": "bash", "description": "Run a shell comma
  }'
```

3. Expected outcome: `lms ps` first reports `CONTEXT 8192` (LM Studio's default). After reloading the model with a larger context window and running `lms ps` again, the reported context reflects the new, higher value. The lesson: the default window is smaller than the model can support, so long sessions silently truncate unless you raise it deliberately. The larger window costs more memory.

3 Installing & configuring pi

> **Learning objective**
>
> You can install the pi command-line interface (CLI) from npm, verify it runs, and find your way around its help, configuration directory, and bundled documentation.

3.1 Prerequisites

pi is a Node.js CLI, so you need a recent Node and npm:

```
node --version   # e.g. v26.x  (Node 20+ is fine)
npm  --version
```

If Node is missing, `brew install node`.

3.2 Install

```
npm install -g --ignore-scripts @earendil-works/pi-coding-agent
```

This installs the CLI as the command `pi`. Verify:

```
pi --version     # e.g. 1.4.0
pi --help        # full flag reference
```

> **Why `--ignore-scripts`?**
>
> `--ignore-scripts` skips any package lifecycle scripts during install. It is a sensible default hardening step for any globally installed CLI (pi does not require post-install scripts to function), and it is the install line pi's own documentation shows.

A note on naming history: pi was originally published under `@mariozechner/…` in the `badlogic/pi-mono` repository; it now lives at `earendil-works/pi` (Zechner 2026b) and publishes the `@earendil-works/pi-coding-agent` package. If you find older blog posts referencing the `@mariozechner` package, they describe the same tool.

3.3 What you just installed

The package is a small monorepo of four parts; as a user you mostly care about the last one.

Package	Role
`pi-ai`	Unified API (application programming interface) over Anthropic / OpenAI / Google / any OpenAI-compatible endpoint
`pi-agent-core`	The agent loop: tool calls, state, streaming
`pi-tui`	The terminal user interface (UI)
`pi-coding-agent`	The `pi` CLI you run

3.4 The configuration directory

pi keeps global configuration under `~/.pi/agent/`:

```
ls ~/.pi/agent/
# auth.json      ← credentials for cloud providers (not needed for local backends)
# models.json    ← custom providers/models (we create this in the next chapter)
```

You will create `~/.pi/agent/models.json` in Chapter 4. For local-only use you do not need `auth.json` or any API key.

3.5 The bundled documentation

A real advantage of pi is that its full docs ship inside the package: handy offline and always matched to your installed version. Find them with:

```
ls "$(npm root -g)/@earendil-works/pi-coding-agent/docs"
# models.md  custom-provider.md  providers.md  usage.md  rpc.md  sessions.md  …
```

`docs/models.md` and `docs/custom-provider.md` are the authoritative reference for the configuration in the next chapter.

3.6 Keeping pi up to date

```
pi update self      # update the pi CLI itself
pi update --all     # update pi and any installed extensions
```

Exercises

1. Run `pi --help` and find the flags for non-interactive mode, choosing a provider, and listing models. You will use all three in Chapter 5.
2. Open `docs/usage.md` from the bundled docs and skim the "Non-interactive" section.
3. Inspect `~/.pi/agent/` before and after first run: which files appear?

Solutions

1. Non-interactive mode is `-p`/`--print` (process the prompt and exit); the provider is chosen with `--provider` (e.g. `--provider lmstudio`); models are listed with `--list-models`. These are the three flags combined throughout Chapter 5, for example `pi -p --offline --provider lmstudio --model 'qwen3-coder-30b-a3b-instruct' "..."`.
2. The "Non-interactive" section of `docs/usage.md` documents the same one-shot usage shown here: pass

`-p`/`--print` with a prompt string so pi runs the request and exits, which is the mode used for scripting and quick trials. (Exact wording depends on your installed version, since the docs ship matched to the package.)

3. Before any run, `~/.pi/agent/` is typically empty or absent. After installing and using pi you will see credential and configuration files appear there, notably `auth.json` (cloud credentials, not needed for local backends) and, once you create it in Chapter 4, `models.json` for custom providers and models. For local-only use no `auth.json` or API key is required.

4 Pointing pi at a local model

> **Learning objective**
>
> You can register an LM Studio model with pi through `~/.pi/agent/models.json`, list it, and understand the compatibility flags that local OpenAI-compatible servers need.

4.1 The idea

pi already knows how to speak the OpenAI Chat Completions protocol, and LM Studio *serves* that protocol at `http://localhost:1234/v1`. So "use a local model instead of Claude" is really just: define a provider in `models.json` that points pi at that local URL (uniform resource locator).

4.2 Minimal configuration

Create `~/.pi/agent/models.json`:

```
{
  "providers": {
    "lmstudio": {
      "baseUrl": "http://localhost:1234/v1",
      "api": "openai-completions",
      "apiKey": "lmstudio",
      "compat": { "supportsDeveloperRole": false, "supportsReasoningEffort": false },
      "models": [
        { "id": "qwen3-coder-30b-a3b-instruct", "reasoning": false },
        { "id": "openai/gpt-oss-20b",           "reasoning": true  }
      ]
    }
  }
}
```

Field by field:

- `baseUrl`: LM Studio's OpenAI-compatible endpoint. Always `/v1`.
- `api: "openai-completions"`: the protocol pi speaks to this provider.
- `apiKey: "lmstudio"`: *required* by pi, but LM Studio ignores its value. Any non-empty string works.
- `compat`: local servers do not implement every OpenAI extension. For reasoning models such as `gpt-oss`, set `supportsDeveloperRole: false` (pi then sends the system prompt as a `system` message rather than the newer `developer` role) and `supportsReasoningEffort: false`. These can sit at the provider level (all models) or per model.
- `models[].id`: must match the identifier the server advertises exactly (`lms ps`, or `curl http://localhost:1234/v1/models`).
- `reasoning`: `true` for models that "think" (e.g. `gpt-oss`); `false` otherwise.

For local models that is genuinely all you need: `name`, `contextWindow`, `maxTokens`, and `cost` are optional and default sensibly (cost defaults to zero, which is correct for a model running on your own hardware). A fuller entry looks like:

```
{
  "id": "openai/gpt-oss-20b",
  "name": "GPT-OSS 20B (LM Studio, local)",
  "reasoning": true,
  "input": ["text"],
  "contextWindow": 131072,
  "maxTokens": 32000,
  "cost": { "input": 0, "output": 0, "cacheRead": 0, "cacheWrite": 0 }
}
```

! contextWindow is pi's bookkeeping, not LM Studio's limit

The `contextWindow` you declare here only tells pi when to compact the conversation. The *actual* served context is the window the model was loaded with in LM Studio (default 8192; see Section 2.3). If you declare 131072 but leave the model loaded at 8192, long sessions still truncate. Reload the model with a larger context to match.

4.3 Adding the MLX provider

The same `providers` map holds one block per backend (Chapter 2). Add `mlx` next to `lmstudio`; both speak `openai-completions` and want the same `compat` flags. `models.json` is re-read whenever you open `/model`, so no restart is needed.

```
{
  "providers": {
    "lmstudio": { "...": "as above" },
    "mlx": {
      "baseUrl": "http://localhost:8080/v1",
      "api": "openai-completions",
      "apiKey": "mlx",
      "compat": { "supportsDeveloperRole": false, "supportsReasoningEffort": false },
      "models": [
        { "id": "mlx-community/Qwen2.5-Coder-7B-Instruct-4bit", "reasoning": false }
      ]
    }
  }
}
```

Each `id` must match what the server advertises: for `mlx` it is the Hugging Face repo (`curl http://localhost:8080/v1/models`); for `lmstudio` it is the loaded model's identifier (`lms ps`). Launch against either provider:

```
pi --provider lmstudio --model 'qwen3-coder-30b-a3b-instruct'   # MLX, tool-calling
pi --offline --list-models mlx                                  # what mlx serves
```

! The `mlx` provider is for generation, not pi's tools

`mlx_lm.server` does not return structured `tool_calls` (Section 2.2), so a run with `--provider mlx` will not execute pi's tools, so keep it for plain generation. For pi's agent loop, use the `lmstudio` provider. See Chapter 8.

4.4 Confirm pi sees the model

`models.json` is re-read every time you open `/model`, so no restart is needed. List your provider's models:

```
pi --offline --list-models lmstudio
```

```
provider  model                          context  max-out  thinking  images
lmstudio  qwen3-coder-30b-a3b-instruct   131.1K   32K      no        no
lmstudio  openai/gpt-oss-20b             131.1K   32K      yes       no
```

`--offline` skips pi's startup network calls, appropriate when you are only using local models.

4.5 Choosing and switching models

- At launch: `--provider lmstudio --model 'qwen3-coder-30b-a3b-instruct'`.
- Mid-session in the terminal user interface (TUI): type `/model` (or `Ctrl+P` to cycle a shortlist).
- The `--model` flag accepts patterns and `provider/id` form, e.g. `--model lmstudio/qwen3-coder-30b-a3b-instruct`.

Which local model should I pick?

Favor models with genuine tool-calling support (the `qwen3-coder`, `gpt-oss`, and `llama3.3` families all work). Pick the size your random-access memory (RAM) and patience allow: a 13 GB model is snappy; a 40+ GB model is capable but slower to load. Gemma-family models currently lack tool-calling templates and will not drive pi's agent loop. See Chapter 8.

Exercises

1. Add a third model you have downloaded (`lms ls`) to `models.json` and confirm it appears in `pi --list-models lmstudio`.
2. Remove the `compat` block, re-run a `gpt-oss` prompt, and observe what breaks. Then put it back.
3. Set a deliberately tiny `contextWindow` (e.g. 2048) and watch when pi decides to compact during a longer task.

Solutions

1. Add a new entry to the `lmstudio` provider's `models` array whose `id` matches the identifier from `lms ls` exactly, then list the provider again (no restart needed, since `models.json` is re-read whenever you open `/model`):

   ```
   { "id": "qwen2.5-coder-7b-instruct", "reasoning": false }
   ```

   ```
   pi --offline --list-models lmstudio
   ```

   The new model appears as a row in the table alongside the existing ones.

2. Removing `compat` drops `supportsDeveloperRole: false` and `supportsReasoningEffort: false`. pi then sends the system prompt with the newer `developer` role and may pass a reasoning-effort parameter, which local OpenAI-compatible endpoints such as LM Studio's do not implement. The result is errors or degraded behaviour for the `gpt-oss` reasoning model. Restoring the `compat` block fixes it.

3. With `contextWindow` set to 2048, pi treats the conversation budget as tiny and decides to compact (summarize earlier turns) very early in a longer task, since `contextWindow` is pi's own bookkeeping for when to compact, not the served limit. Note the served context is still governed by the window the model was loaded with in LM Studio, so a too-small value there causes truncation independently of this setting.

5 Your first agentic session — the four tools

> **Learning objective**
>
> You can run pi both interactively and non-interactively against your local model, you understand the four built-in tools, and you can prove for yourself that the agent really executes them.

5.1 The whole agent in four tools

pi gives the model exactly four built-in tools:

Tool	What it does
`read`	Read a file's contents
`write`	Create or overwrite a file
`edit`	Make a surgical change to a file
`bash`	Run a shell command

That is the entire surface. There is no separate "search", "todo", "plan", or "browser" tool. The model composes everything from these four (you can `bash` your way to `grep`, `git`, `python3`, and so on). Chapter 1 explains the philosophy; here we just use it.

5.2 Two ways to run

Interactive (terminal user interface, TUI):

```
pi --provider lmstudio --model 'qwen3-coder-30b-a3b-instruct'
```

You get a chat-style terminal UI; type a request and watch pi read, write, edit, and run commands. Exit with `Ctrl+C`.

Non-interactive (one-shot), ideal for scripting and for trying things out:

```
pi -p --offline --provider lmstudio --model 'qwen3-coder-30b-a3b-instruct' \
   "Create fib.py that prints the first 10 Fibonacci numbers, then run it with python3."
```

`-p`/`--print` processes the prompt and exits. `--offline` keeps everything local.

5.3 A verified first task

Running the command above in an empty directory produced `fib.py` on disk:

```
a, b = 0, 1
for _ in range(10):
    print(a)
    a, b = b, a + b
```

…and the correct output `0 1 1 2 3 5 8 13 21 34`. The file appearing on disk is proof that the `write` tool fired, not just that the model described a file.

5.4 Proving the agent really runs commands

A healthy scepticism: how do we know `bash` *executed*, rather than the model *pretending* to show output? Ask for something the model cannot possibly know, a fact about *your* machine, and have it written to disk:

```
pi -p --offline --provider lmstudio --model 'qwen3-coder-30b-a3b-instruct' \
   "Use the bash tool to run exactly:  uname -m > proof.txt && sysctl -n hw.model >> proof.txt && cat p
```

On the test machine `proof.txt` then contained:

```
arm64
Mac16,5
```

Those are this host's central-processing-unit (CPU) architecture and hardware model, values the model has no way to fabricate. Their presence on disk is conclusive evidence that the local model drove pi's `bash` tool for real. It is the same uncheatable proof pattern that verified LM Studio's tool-calling in Section 2.3, and any other tool-capable model you register in Chapter 4 can be checked the same way (bigger models just take longer to load first; see Chapter 8).

5.5 A safety word before you go further

⚠ pi has no permission popups

By design, pi does not prompt before each tool call: built-in tools read, write, edit, and run shell commands **with your user's full permissions**. There is no sandbox. That is what makes one-shot mode so smooth, but it also means you should run pi in directories you trust and treat untrusted repository content as you would any code you are about to execute. Chapter 6 shows how to constrain behaviour with `AGENTS.md`, and Chapter 8 links pi's own security notes.

5.6 Handy input syntax

- `@path/to/file` injects a file's contents into your prompt.
- `!command` runs a shell command and includes its output in your prompt.
- Piped stdin is merged into the prompt: `cat README.md | pi -p "Summarize this"`.

💡 Exercises

1. Re-run the Fibonacci task interactively (`pi --provider lmstudio --model qwen3-coder-30b-a3b-instruct`) and watch each tool call scroll by. Which of the four tools does it use?
2. Restrict pi to a single tool with `--tools read` and ask it to *change* a file. What happens, and why?
3. Use `@` to feed pi one of this book's `.qmd` files and ask it to summarize the chapter.

ℹ Solutions

1. The task typically uses `write` to create `fib.py` and `bash` to run it with `python3`; if the model revises the file it may also fire `edit`. The conclusive sign is `fib.py` appearing on disk with the correct output `0 1 1 2 3 5 8 13 21 34`, proving the `write` and `bash` tools actually fired rather than the model merely describing them.
2. With `--tools read` the model has only the `read` tool available, so it cannot create or modify the file: there

is no `write` or `edit` tool in its surface to call. It will typically read the file and report that it cannot change it (or describe the change in prose only), and no file on disk is altered. This is the tool surface acting as a guard rail, useful given that pi otherwise runs tools with your full permissions and no sandbox.

3. Prefix the path with `@` to inject the file's contents into your prompt, for example:

```
pi -p --offline --provider lmstudio --model 'qwen3-coder-30b-a3b-instruct' \
   "Summarize this chapter: @05_first_session.qmd"
```

pi reads the named `.qmd` file's text into the prompt and the model returns a summary. Any chapter file works; the summary quality depends on the local model you chose.

6 Context files & sessions

> **Learning objective**
>
> You can steer pi's behaviour with `AGENTS.md` and `SYSTEM.md`, and you can save, resume, branch, and export sessions.

6.1 Telling pi how to work: `AGENTS.md`

Instead of repeating instructions every run, you put project instructions in an `AGENTS.md` file. pi discovers these by starting at `~/.pi/agent/` (global) and then walking up from the current directory, concatenating every match it finds. For compatibility it also reads `CLAUDE.md`.

A small project `AGENTS.md`:

```
# Project rules for pi

- This is a Python project managed with `uv`. Use `uv run` to run code.
- Always run `ruff check .` after editing Python files.
- Never touch files under `data/` - they are large and read-only.
```

Because matches are concatenated up the tree, you can keep broad rules in a global `~/.pi/agent/AGENTS.md` and project-specific rules in each repo.

6.2 Replacing the system prompt: `SYSTEM.md`

`AGENTS.md` *adds* guidance; `SYSTEM.md` *replaces* pi's built-in system prompt entirely. Place it at `~/.pi/agent/SYSTEM.md` (global) or `.pi/SYSTEM.md` (project). To *append* to the default prompt instead of replacing it, use `APPEND_SYSTEM.md`. Most beginners should leave the system prompt alone and use `AGENTS.md`.

> **Project trust in non-interactive mode**
>
> pi treats project-local files (`.pi/…`, project `AGENTS.md`) as something to *trust* before loading. Non-interactive modes (`-p`, `--mode json`, `--mode rpc`) do not show a trust prompt; they follow your global `defaultProjectTrust` setting. Pass `--approve`/`-a` to trust project-local files for one run, or `--no-approve`/`-na` to ignore them.

6.3 Sessions: save, continue, resume, branch

pi automatically saves each session as a tree (you can branch and go back). The flags you will use most:

```
pi -c                 # continue the most recent session
pi -r                 # pick a session to resume
pi --fork <id>        # branch a new session from an existing one
pi --no-session       # ephemeral: don't save this run
pi --name "refactor"  # give a session a readable name
```

Sessions can be exported for sharing or documentation:

```
pi --export session.html   # write an HTML transcript and exit
```

This pairs well with the agent-facing exercises in this book: run a task, then `--export` the transcript as a record of exactly what the model did.

6.4 Extending pi (briefly)

pi keeps the core tiny on purpose; everything else is an extension (a TypeScript module) or a skill (a reusable prompt/tool bundle), discovered from `~/.pi/agent/` or the project's `.pi/`. You do not need any of this to use a local model, but it is where features like permission gates, the Model Context Protocol (MCP), or sub-agents live if you want them later. Manage them with `pi install`, `pi list`, and `pi remove`.

Exercises

1. Add an `AGENTS.md` to a scratch project telling pi to always create a `CHANGELOG.md` entry after edits, then give it a small task and check whether it complied.
2. Run two related tasks with `-c` (continue) so the second builds on the first, then `--export` the session and read the transcript.
3. Use `--fork` to try two different approaches from the same starting point.

Solutions

1. Create `AGENTS.md` in the scratch project with a rule such as: "After any edit, add a dated entry to `CHANGELOG.md` describing the change." Because pi walks up from the current directory and concatenates every `AGENTS.md` (and `CLAUDE.md`) it finds, this project file is picked up automatically. Give pi a small edit task and inspect whether `CHANGELOG.md` gained a matching entry. Compliance is guidance, not a hard guarantee, so the point of the check is to see how reliably the model follows the standing instruction.
2. Run the first task normally, then run the second with `pi -c` to continue the most recent session so its context carries over and the second task can build on the first. Finish with `pi --export session.html` to write an HTML (Hypertext Markup Language) transcript, then open it to read exactly what the model did across both turns. The transcript doubles as a shareable record of the run.
3. Start from a saved session and run `pi --fork <id>` to branch a new session from that point, then pursue approach A. Fork the same `<id>` again for approach B. Both branches share the identical starting context but evolve independently, so you can compare the two strategies without one polluting the other. (You can also fork the current session interactively; forking from an explicit id is the scriptable form.)

7 Headless “server” mode (RPC)

> **Learning objective**
>
> You understand pi’s non-interactive output modes (`text`, `json`, and `rpc`), and you can decide which to use for scripting versus for embedding pi as a programmatic backend.

7.1 “PI Server” demystified

Newcomers often ask how to run pi “as a server”. pi is not a long-running network daemon you point a browser at; its headless story is the `--mode` flag, which changes how a single `pi` process talks to whatever is driving it:

Mode	Output	Use it for
`text` (default)	Human-readable text	Interactive use and one-shot `-p` prompts
`json`	Structured JSON events	Scripting that needs to parse what happened
`rpc`	JSON-over-stdin/stdout protocol	Embedding pi as a backend in other programs

So a “pi server”, in the sense of *something a program can drive*, is `pi --mode rpc` (remote procedure call): your program writes JSON (JavaScript Object Notation) commands to pi’s stdin and reads JSON events from its stdout. This is how you would wire a local-model pi into an editor plugin or a service, all still running against LM Studio with no cloud involved.

7.2 The crucial gotcha

> **json and rpc modes read from stdin**
>
> Both `--mode json` and `--mode rpc` expect to be driven over stdin. If you run, say, `pi -p --mode json "do something"` from a script with no stdin attached, pi reaches end-of-input immediately, emits nothing, and exits. That is not a model or network failure; it is the protocol working as designed. For one-shot prompts you want text mode (`-p`, the default); reserve `json`/`rpc` for programs that keep a pipe open and speak the protocol. This trips up almost everyone once; see Chapter 8.

7.3 A mental model for driving rpc

A controlling program typically:

1. Spawns `pi --mode rpc` (optionally with `--provider lmstudio --model …`).
2. Keeps the child’s stdin and stdout pipes open for the process lifetime.
3. Writes JSON request lines to stdin (prompts, tool permissions, model switches).
4. Reads JSON event lines from stdout (text deltas, tool calls, completion).

The exact message schema is documented in the package’s `docs/rpc.md` and `docs/json.md` (`ls "$(npm root -g)/@earendil-works/pi-coding-agent/docs"`), and is the most reliable reference because it matches your installed version. For most users building automation, the simplest robust option is still `pi -p` text mode wrapped in a shell script; reach for `rpc` only when you are building an interactive integration.

7.4 When to use which

- A nightly script that asks the local model to summarize a repo: `pi -p` (text).
- A pipeline step that must branch on what tools ran → `pi -p --mode json`, parsing the event stream.
- An editor plugin / desktop app embedding pi: `pi --mode rpc`, holding the pipes open.

Exercises

1. Open `docs/rpc.md` and `docs/json.md` from the bundled docs and identify the event types pi emits when it makes a tool call.
2. Reproduce the gotcha: run `pi -p --mode json --offline --provider lmstudio --model qwen3-coder-30b-a3b-instruct "hi"` with no stdin and confirm you get no output. Then run the same prompt in plain `-p` text mode.
3. Sketch (in pseudocode) a wrapper that spawns `pi --mode rpc`, sends one prompt, and prints assistant text events.

Solutions

1. Locate the bundled docs with `ls "$(npm root -g)/@earendil-works/pi-coding-agent/docs"` and open `rpc.md` and `json.md`. The exact names match your installed version, which is why these files (rather than memory) are the reliable reference. You are looking for the events emitted around a tool call: a tool-call event carrying the tool name and its arguments, followed later by a tool-result/completion event. Text output arrives as a stream of text-delta events, and the run ends with a completion event. Note the precise field names from your local copy, since they are version-specific.

2. With no stdin attached, `pi -p --mode json --offline --provider lmstudio --model qwen3-coder-30b-a3b-instruct "hi"` produces no output and exits: json mode reads commands from stdin, hits end-of-file (EOF) immediately, and stops. This is the protocol working as designed, not a model or network failure. Running the same prompt in plain text mode, `pi -p --offline --provider lmstudio --model qwen3-coder-30b-a3b-instruct "hi"`, returns a normal answer, confirming that the silence was the mode and not the setup.

3. A minimal wrapper, in pseudocode:

```
proc = spawn("pi", ["--mode", "rpc", "--provider", "lmstudio", "--model", "qwen3-
coder-30b-a3b-instruct"])
keep proc.stdin and proc.stdout open

write JSON request line to proc.stdin: { type: "prompt", text: "Summarize this repo" }

for each line read from proc.stdout:
    event = parse_json(line)
    if event is an assistant text delta:
        print event.text
    if event signals completion:
        break

close proc.stdin
```

   The key points are: hold both pipes open for the process lifetime, write newline-delimited JSON requests, and read newline-delimited JSON events, printing only the assistant text deltas and stopping on the completion event. Use the field names from your local `docs/rpc.md` for the real implementation.

8 Troubleshooting: successes & failures

> **Learning objective**
>
> You can recognise and fix the handful of issues that actually bite when running pi against a local MLX or LM Studio backend, drawn from a real setup session.

This chapter is a field guide: each item is something that either worked surprisingly well or went wrong during the verified setup behind this book.

8.1 What worked well

- Install is trivial and fast. `npm i -g --ignore-scripts @earendil-works/pi-coding-agent` pulled ~140 packages in a few seconds; `pi --version` reported `0.79.9`.
- Local tool-calling genuinely works. `qwen3-coder-30b-a3b-instruct` drove pi's `write` and `bash` tools through LM Studio, confirmed by on-disk side-effects the model could not fake (Chapter 5, Section 2.3).
- No permission popups means clean scripting. One-shot `-p` runs complete unattended.
- Metal acceleration is automatic. Both backends build on MLX, which offloads the model to the graphics processing unit (GPU) out of the box; `lms ps` shows the loaded model and its device.

8.2 Failures and fixes

8.2.1 `--mode json` / `--mode rpc` produces no output

These modes read commands from stdin. Invoked with a bare prompt argument and no stdin attached, pi sees end-of-file (EOF) and exits silently. Use plain `-p` text mode for one-shot prompts; keep `json`/`rpc` for programs that hold a pipe open (Chapter 7).

8.2.2 Long sessions get truncated / the model "forgets"

LM Studio loads models with an 8192-token context window by default, no matter what you put in pi's `models.json` `contextWindow` (which is only pi's own compaction threshold). Check the `CONTEXT` column in `lms ps`. Fix by reloading the model with a larger context window, either in the app's model-load dialog or from the command-line interface (CLI), e.g.:

```
lms load qwen3-coder-30b-a3b-instruct --context-length 32768
```

Bigger context costs more memory.

8.2.3 A reasoning model errors or behaves oddly

Reasoning models such as `gpt-oss` need compatibility flags on the provider so pi does not send OpenAI-only features the local server rejects:

```
"compat": { "supportsDeveloperRole": false, "supportsReasoningEffort": false }
```

(pi then uses the `system` role instead of `developer` and omits `reasoning_effort`.)

8.2.4 The first request is very slow

Cold-loading a model into memory dominates first-call latency: a 40+ GB model can take the better part of a minute before any output, then is fast while it stays resident. Pick a model size that fits your latency budget; load the model ahead of time with `lms load <model>` if you script it.

8.2.5 The model ignores the tools entirely

Not every model is trained (or templated) for tool-calling. Stick to families with real tool support: `qwen3-coder`, `gpt-oss`, `llama3.3`. Gemma-family models currently lack tool-calling templates and will not drive pi's agent loop, however good they are at chat.

8.2.6 `mlx_lm.server` runs but pi's tools never fire

The bare MLX server (`mlx_lm.server`, Section 2.2) serves text correctly but, as of mlx-lm 0.31.3, does not emit structured `tool_calls`: the model's tool attempt comes back as plain assistant text (a fenced JSON block, JavaScript Object Notation) with `tool_calls: null`, so pi has nothing to dispatch and no file is created. This is the server, not the prompt or the model. Use `mlx_lm.server` only for generation-only workloads; drive pi's agent loop through LM Studio (`lmstudio` provider), whose endpoint parses tool calls.

8.2.7 LM Studio: `Invalid passkey`, or `lms get` cannot find a model

Two first-run snags with the `lms` CLI. A stale `~/.lmstudio/bin/lms` left by an older app build rejects the CLI with `Invalid passkey for lms CLI client`; re-bootstrap from the freshly installed bundle:

```
"/Applications/LM Studio.app/Contents/Resources/app/.webpack/lms" bootstrap
```

And `lms get author/repo` fails with `artifact … does not exist` for arbitrary community models, because LM Studio lowercases the name and searches its own catalogue. Pass the full Hugging Face URL (uniform resource locator) instead: `lms get "https://huggingface.co/mlx-community/…"`.

8.2.8 Picking a model for your RAM

A model needs roughly its on-disk size in free random-access memory (RAM) (Chapter 2). On 8–16 GB Macs use 3B–8B models; the 128 GB machine in this book ran 40+ GB models comfortably, but a 235B model (140+ GB on disk) exceeds even that and should be avoided there.

8.3 A note on security

pi is a local agent with no sandbox: its tools act with your user's permissions. Project "trust" only governs whether project-local *settings* are loaded; it does not restrict what the model can ask the tools to do once you are working in a directory. Treat untrusted repositories accordingly. pi's bundled `docs/security.md` is the authoritative statement.

Exercises

1. Stop the LM Studio server (`lms server stop`), run a one-shot pi prompt against the `lmstudio` provider, and read the error pi reports. Restart the server and confirm the same prompt succeeds.
2. Compare `lms ps` `CONTEXT` before and after reloading a model with a larger `--context-length`.
3. Try a Gemma model in `models.json` and confirm it cannot complete a `bash` task; then switch back to `qwen3-coder-30b-a3b-instruct`.

Solutions

1. With the server stopped, pi cannot reach `http://localhost:1234/v1` and reports a connection error (connection refused) instead of hanging: the failure is immediate and names the endpoint, which distinguishes "no server" from "server up but model misbehaving". After `lms server start` (and loading the model), the same one-shot prompt returns a normal answer.

2. Inspect the `CONTEXT` column of `lms ps` while a model is loaded: by default it shows 8192, because LM Studio loads models with an 8192-token window regardless of pi's `models.json` `contextWindow` (which only sets pi's own compaction threshold). Reload with a larger window and `lms ps` now reports the raised value. The larger window costs more memory.

```
lms ps                                                  # CONTEXT shows 8192
lms load qwen3-coder-30b-a3b-instruct --context-length 32768
lms ps                                                  # CONTEXT now shows 32768
```

3. Point `models.json` at a Gemma model and ask pi to do a `bash` task (for example, create a file). The model will not drive the agent loop: Gemma-family models currently lack tool-calling templates, so the tools never fire and no file is created, however fluent the chat reply looks. Switch the model back to `qwen3-coder-30b-a3b-instruct` (a family with real tool support, alongside `gpt-oss` and `llama3.3`) and the same `bash` task completes with the expected on-disk side-effect.

References

Alizadeh, Keivan, Seyed Iman Mirzadeh, Dmitry Belenko, et al. 2024. "LLM in a Flash: Efficient Large Language Model Inference with Limited Memory." *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. https://doi.org/10.18653/v1/2024.acl-long.678.

Apple Machine Learning Research. 2026. "MLX-LM: Run and Fine-Tune Language Models with MLX." Software, GitHub: ml-explore/mlx-lm. https://github.com/ml-explore/mlx-lm.

Chen, Mark, Jerry Tworek, Heewoo Jun, et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv Preprint arXiv:2107.03374*. https://arxiv.org/abs/2107.03374.

Hannun, Awni, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2023. *MLX: Efficient and Flexible Machine Learning on Apple Silicon*. Software framework, GitHub: ml-explore/mlx. https://github.com/ml-explore/mlx.

Hou, Xinyi, Yanjie Zhao, Yue Liu, et al. 2024. "Large Language Models for Software Engineering: A Systematic Literature Review." *ACM Transactions on Software Engineering and Methodology* 33 (8). https://doi.org/10.1145/3695988.

Jimenez, Carlos E., John Yang, Alexander Wettig, et al. 2024. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2310.06770.

Lin, Ji, Jiaming Tang, Haotian Tang, Shang Yang, Guangxuan Xiao, and Song Han. 2025. "AWQ: Activation-Aware Weight Quantization for on-Device LLM Compression and Acceleration." *GetMobile: Mobile Computing and Communications* 28 (4). https://doi.org/10.1145/3714983.3714987.

LM Studio. 2026. "LM Studio: Discover, Download, and Run Local LLMs." Software, lmstudio.ai. https://lmstudio.ai.

Schick, Timo, Jane Dwivedi-Yu, Roberto Dessì, et al. 2023. "Toolformer: Language Models Can Teach Themselves to Use Tools." *Advances in Neural Information Processing Systems (NeurIPS)* 36. https://doi.org/10.52202/075280-2997.

Sierra-Herrera, Oscar H., Mario Eduardo González Niño, Edwin Francis Cárdenas Correa, Jersson X. Leon-Medina, and Francesc Pozo. 2026. "A Reproducible Benchmarking Methodology for Machine Learning Hardware: Performance–Energy Trade-Offs from GPUs to Apple Silicon." *Algorithms* 19 (5): 363. https://doi.org/10.3390/a19050363.

Wang, Lei, Chen Ma, Xueyang Feng, et al. 2024. "A Survey on Large Language Model Based Autonomous Agents." *Frontiers of Computer Science* 18 (6). https://doi.org/10.1007/s11704-024-40231-1.

Yao, Shunyu, Jeffrey Zhao, Dian Yu, et al. 2023. "ReAct: Synergizing Reasoning and Acting in Language Models." *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2210.03629.

Zechner, Mario. 2025. "What i Learned Building an Opinionated and Minimal Coding Agent." November. https://mariozechner.at/posts/2025-11-30-pi-coding-agent/.

Zechner, Mario. 2026a. *Building Pi in a World of Slop*. Conference talk, YouTube. https://www.youtube.com/watch?v=RjfbvDXpFls.

Zechner, Mario. 2026b. *Pi: A Minimal, Extensible Terminal Coding Agent*. GitHub repository, earendil-works/pi. https://github.com/earendil-works/pi.

Zhang, Yuntong, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. "AutoCodeRover: Autonomous Program Improvement." *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3650212.3680384.

Zhu, Xunyu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2024. "A Survey on Model Compression for Large Language Models." *Transactions of the Association for Computational Linguistics* 12. https://doi.org/10.1162/tacl_a_00704.