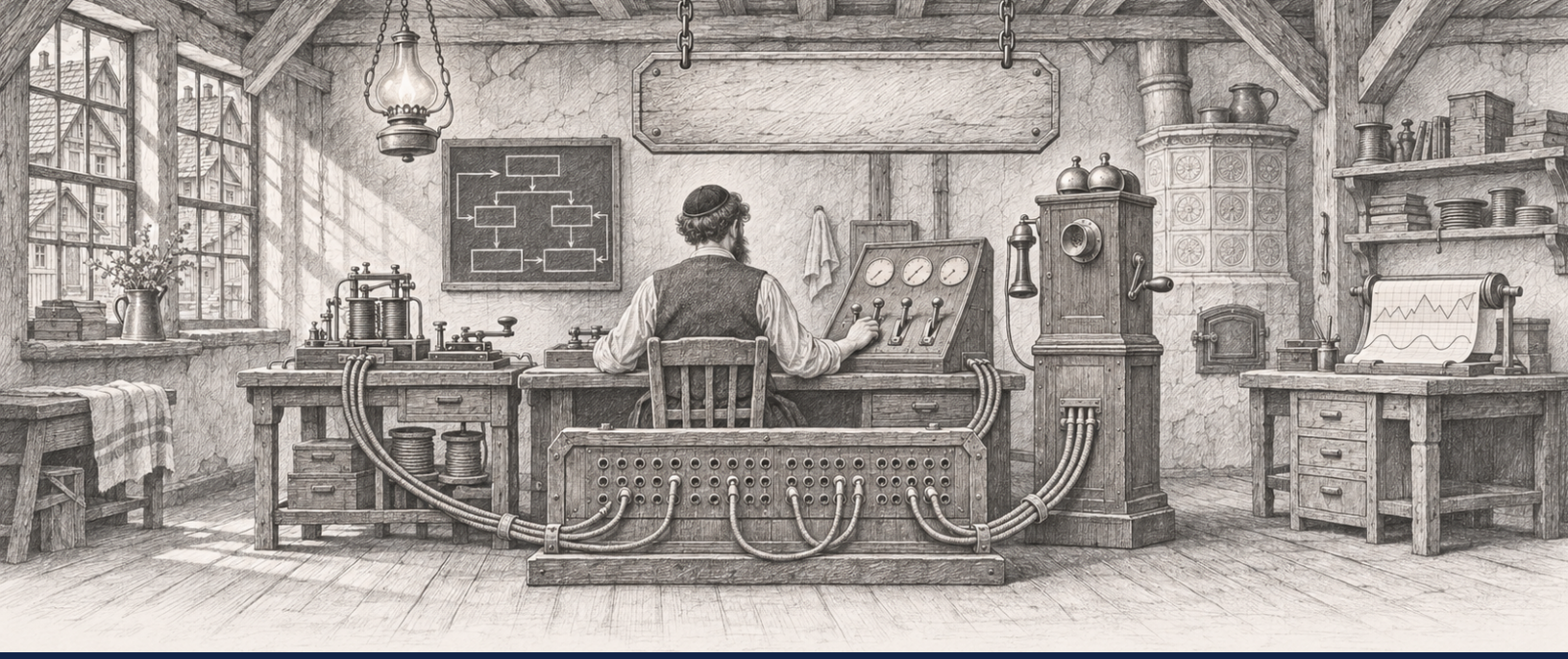

hrx

Launching, Supervising and Scaling AI Coding Agents

A harness launcher for Codex, Claude Code and OpenCode through one model gateway

November 22, 2026

Revision 203e1e8

Stephen Barr

Contents

Preface 1

0.1 Who This Book Is For . . . 1
0.2 What You Need . . . 2
0.3 How This Book Is Organized . . . 2
0.4 Conventions Used in This Book . . . 2
0.4.1 Example names . . . 3
0.5 Using the Examples . . . 3

I Getting Started 4

1 Why a Launcher? 6

1.1 Three Harnesses, One Bill . . . 6
1.2 What hrx Does . . . 7
1.3 The Principles . . . 7
1.3.1 Exact routes, never substitutes . . . 8
1.3.2 Fail closed . . . 8
1.3.3 No new credentials . . . 8
1.3.4 The harness stays native . . . 8
1.3.5 Look before you leap . . . 8
1.3.6 Records outlive sessions . . . 8
1.4 How the Pieces Fit . . . 9
1.5 What hrx Is Not . . . 9
1.6 Summary . . . 10

2 Installing hrx and Launching Your First Agent 11

2.1 What hrx Expects from Your Gateway . . . 11
2.2 Building and Installing . . . 12
2.2.1 Tab completion . . . 12
2.3 Connecting Each Harness to the Gateway . . . 13
2.3.1 Codex . . . 13
2.3.2 OpenCode . . . 13
2.3.3 Claude Code . . . 14
2.4 Refresh: Fetching the Catalog . . . 14
2.5 Doctor: The Read-Only Health Check . . . 15
2.6 Dry Run: See the Launch Before You Pay for It . . . 16

2.7 Your First Session . . . 16
2.7.1 Passing arguments to the harness . . . 16
2.7.2 Permission prompts . . . 17
2.7.3 Resuming a session . . . 17
2.8 The Whole Command Line . . . 17
2.9 Summary . . . 19

3 Models, Routes and Reasoning **20**
3.1 Identities, Routes and Aliases . . . 20
3.1.1 How a name is resolved . . . 20
3.2 Listing What You Can Use . . . 21
3.3 Reading the Catalog . . . 21
3.3.1 Route statuses . . . 22
3.3.2 Evidence badges . . . 23
3.4 Keeping Up with New Models: `--latest` . . . 23
3.5 Reasoning Levels . . . 24
3.6 Choosing Interactively . . . 24
3.7 When a Model Is Missing . . . 25
3.8 How Fresh Is Fresh? . . . 25
3.9 Summary . . . 25

III Agents at Scale **27**

10 Concierge: Small Triggered Workers **29**
10.1 What a Concierge Worker Is . . . 29
10.2 What You Need . . . 30
10.3 A First Worker . . . 31
10.3.1 Dry runs and `--yes` . . . 32
10.4 The Four Triggers . . . 32
10.4.1 The mention listener . . . 33
10.5 Where It Posts . . . 33
10.6 Managing Workers . . . 34
10.6.1 Listing . . . 34
10.6.2 Reading the log . . . 34
10.6.3 Stopping one . . . 34
10.6.4 Pruning the finished ones . . . 35
10.6.5 Elsewhere in hrx . . . 35
10.7 Deciding Whether to Trust It . . . 35
10.7.1 Everything a trigger delivers is untrusted . . . 35
10.7.2 What a worker can and cannot do . . . 36
10.7.3 What hrx will let a worker watch . . . 37
10.7.4 What remains . . . 37
10.8 Credentials . . . 37
10.8.1 Why pruning matters for short workers . . . 38
10.9 Configuration . . . 38
10.10 When Something Goes Wrong . . . 39
10.11 Summary . . . 40

Preface

A year ago, "using an AI coding agent" meant opening one tool in one terminal and typing at it. Today a working day can involve three different agent harnesses, a dozen models from five providers, several paid accounts, and agents that run for hours on machines you never log in to. Each harness has its own configuration format, its own idea of what a model name means, and its own way of failing when something is misconfigured. None of them, by themselves, knows how to give an agent a stable name, a mailbox, scoped credentials or a place on a shared message board.

hrx is a launcher that sits in front of those harnesses. It starts Codex, Claude Code or OpenCode against one model gateway, checks that the exact model route you asked for is real and authorized before anything starts, and then gets out of the way. Around that small core it adds the things you need once agents stop being a single conversation: launch profiles, agent identities, message delivery, short-lived Git credentials, isolated remote workers, declarative networks of agents, a live dashboard, and a sweeper that cleans up after all of it.

This book is a practical guide to that toolkit, and to the habits that make agent teams work. It starts with one agent in one terminal and ends with a team of six agents reviewing each other's work on isolated machines, measured by an experiment you can repeat.

Who This Book Is For

You already use AI coding agents. You know what a context window is, why one model costs ten times another, and how it feels when an agent confidently does the wrong thing. You are comfortable in a Unix shell and can read a TOML file.

You will get the most from this book if one of these is true:

- You switch between harnesses or models and are tired of editing three configuration files to do it.
- You pay for models through a gateway such as LiteLLM and want every launch to go through it, with no surprise direct-to-provider traffic.
- You want to run more than one agent at a time and need them to have names, communicate, and leave an audit trail.
- You are running, or planning, experiments that compare agent setups and need launches to be reproducible.

You do not need to know Go to use hrx. You will see a little of it in later chapters, where knowing how a check works explains why it refuses something.

What You Need

- A macOS or Linux machine with Go 1.26 or later.
- At least one harness installed: OpenAI's Codex CLI, Anthropic's Claude Code, or OpenCode.
- An OpenAI-compatible model gateway that publishes a model catalog. The book uses LiteLLM, and Chapter 2 explains exactly what hrx expects the gateway to provide.

Later parts add optional pieces: a message board for agents, a GitLab instance, and a Proxmox cluster for remote workers. Each chapter that needs one says so at the start, and every earlier chapter works without them.

How This Book Is Organized

Part I, Getting Started, covers one agent on one machine. Chapter 1 explains the problem hrx solves and the principles that shape it. Chapter 2 installs hrx, connects it to a gateway, and launches your first agent. Chapter 3 is about choosing a model: how hrx turns a friendly name into an exact, authorized route, and how to read the catalog when a model you expected is missing.

Part II, Working Every Day, covers launch profiles, agent identities and communication, permissions, resuming sessions, and natural-language launches.

Part III, Agents at Scale, moves agents off your laptop: isolated remote workers, declarative networks of agents, the dashboard and hub, cleaning up, and per-agent Git credentials. Chapter 10 covers the smallest kind of worker: a short-lived background process, requested in plain English, that wakes on a trigger and posts to a message board.

Part IV, Using hrx Well, is about judgment rather than mechanics. It walks through a real experiment, six agents building and reviewing a feature under two different communication designs, and draws out the patterns and anti-patterns it revealed.

The appendices hold the command and configuration reference, a troubleshooting guide, and a glossary.

Conventions Used in This Book

`Constant width` Commands, flags, file names, configuration keys, model routes and output.
`Constant width with CAPITALS, such as ROUTE` A value you replace with your own.
`<Tab>` Press the Tab key. Do not type the angle brackets.

> **Tip**
> A tip suggests a faster or safer way to do something.

> **Note**
> A note adds background that you can skip on a first read.

> **Warning**
> A warning marks something that can cost money, leak a credential, or destroy work.

Example names

Every example uses placeholder infrastructure. Substitute your own values:

Placeholder	Stands for
`https://gateway.example.com/v1`	Your model gateway's OpenAI-compatible endpoint
`acme`	The account name your gateway uses for a provider, as in `or-acme/...`
`laptop/docs/agent`	An agent identity: host, role and agent name
`team-board`	A message board that agents share
`pve1`, `pve2`	Nodes in a Proxmox cluster

Model names such as `claude-sonnet-5`, `gpt-6.1-sol` and `glm-5p3` are real public model identities at the time of writing. Which of them you can use depends on your gateway and your key, and that is the subject of Chapter 3.

Using the Examples

Commands that only inspect, such as `hrx models`, `hrx doctor` and any command with `--dry-run`, never send an inference request. Commands that launch a harness spend real money with your provider. Each chapter marks the boundary.

The command listings in this book that show hrx's own output, such as the `--help` text, are generated from the hrx source when the book is built. If a listing ever disagrees with your copy of hrx, your copy is newer or older than the book; `hrx --version` tells you which.

PART I

Getting Started

One agent, one machine, one gateway. By the end of this part you will have hrx installed, connected to your model gateway, and launching any of three harnesses with a model you chose deliberately, and you will know how to find out why a model you expected is missing.

CHAPTER 1

Why a Launcher?

Starting an AI coding agent looks trivial. You type `codex` or `claude`, the tool opens, and you start talking. The difficulty arrives later and sideways: the session that quietly ran on the wrong account, the model name that meant one thing in one tool and another thing in the next, the agent that could not be told apart from its three siblings in a log, the forgotten worker still running, and billing, at three in the morning.

This chapter describes those problems, the small set of principles hrx uses to solve them, and how the pieces fit together. Nothing here requires hrx to be installed. If you are impatient, skip to Chapter 2 and come back when something hrx does surprises you; the reason is probably here.

Three Harnesses, One Bill

A *harness* is the program that wraps a model in an agent loop: it sends your request, lets the model call tools such as a shell or a file editor, feeds the results back, and repeats. This book covers the three most widely used command-line harnesses:

- **Codex**, OpenAI's CLI, which speaks the OpenAI Responses API.
- **Claude Code**, Anthropic's CLI, which speaks the Anthropic Messages API.
- **OpenCode**, an open source harness that speaks the OpenAI-compatible Chat Completions API and works with many providers.

Each one is good at something, and many people use all three. Each also has its own configuration file, its own credential mechanism, its own list of models it knows about, and its own reaction to a model it does not know.

Meanwhile the models themselves increasingly arrive through a *gateway*: a proxy such as LiteLLM that holds the real provider keys, issues you a single key of its own, enforces budgets, and routes each request to OpenAI, Anthropic, OpenRouter, Fireworks or wherever the model actually runs. A gateway gives you one bill and one place to revoke access. It also adds a layer of naming. The same model can be reachable through several *routes*, one per provider account, each with its own price, rate limits and payer.

Put those together and the question "which model am I using?" has a surprisingly long answer: which harness, which provider configuration inside that harness, which gateway, which route on that gateway, which account pays for it, and whether this harness can actually drive that model's tool calls. Get any one of those wrong and you find out late: a harness silently falls back to a default model, or a request goes to a provider directly instead of through the gateway, or a tool call fails

forty minutes into a task because the route never supported the harness's API.

What hrx Does

hrx is a single command that launches any of the three harnesses against your gateway:

```
hrx codex --model gpt-6.1-sol --reasoning high
hrx claude --model claude-sonnet-5
hrx opencode --model glm-5p3
```

Before it starts anything, hrx:

1. Reads the harness's existing provider configuration to find your gateway and the credential you already use for it.
2. Fetches the gateway's catalog and asks the gateway which routes your key is allowed to call.
3. Resolves the name you gave to exactly one authorized route that this harness is declared compatible with, and checks the reasoning level against what that route advertises.
4. Writes whatever the harness needs to use that route, without touching your own configuration files.
5. Replaces itself with the harness process.

If any step fails, nothing launches. If every step succeeds, you are talking to an unmodified Codex, Claude Code or OpenCode, with the route you meant.

That is the core. Everything else in this book builds on it:

- **Profiles** name a harness, route and reasoning level so a team can say `hrx --profile reviewer` and get the same launch every time.
- **Agent identities** give each agent a stable name of the form `HOST/ROLE/AGENT`, separate from any one session, so its messages, commits and costs can be traced back to it.
- **Communication** connects an agent to a message board and direct messages, and delivers new messages into the running session as turns.
- **Credentials** for Git hosting are issued per agent identity, short-lived, and refreshed during long sessions.
- **Remote workers** run an agent in an isolated Linux container on a Proxmox cluster, prove it can communicate before giving it work, and keep a durable record of everything needed to inspect, resume or destroy it.
- **Networks** declare a team of agents in one TOML file and launch them together.
- **The dashboard** shows every running agent, its model and its activity.
- **The sweeper** finds what launches left behind and, with your approval, backs it up and removes it.

The Principles

hrx's behavior follows from a handful of rules. When it refuses something, one of these is usually the reason.

Exact routes, never substitutes

A model name is resolved to one exact gateway route or the launch fails. If `gpt-6.1-sol` is available through two accounts and neither is the gateway's public alias, hrx lists both and asks you to choose. It never picks a payer for you, and it never quietly swaps in a "close enough" model.

Fail closed

A launch revalidates authorization against the live gateway every time. A stale cache can feed tab completion, but it can never authorize a launch. A failed refresh keeps the last good cache and stops. Unknown capabilities, such as a reasoning level the catalog does not declare, are treated as unavailable rather than guessed.

No new credentials

hrx uses the credentials you have already configured for each harness: an environment variable, a helper command, or a private key file. It never writes key values to its cache, and suppresses credential output and HTTP error bodies so a key cannot leak into a log. Features that do need a credential of their own get it from a service you run, scoped to one purpose and short-lived: Git access for an agent comes from a broker, and each concierge worker gets its own budgeted gateway key from your gateway's key-issuing service.

The harness stays native

hrx does not wrap the harness in a shell or proxy its terminal. After validation it calls `exec`, so the harness keeps hrx's process ID, terminal, signals and exit status. Everything after `--` on the command line goes to the harness untouched, with one exception: options that would override the model or provider are rejected, because they would bypass the route hrx just validated.

Look before you leap

Almost every command has a read-only form. `--dry-run` resolves a launch and prints exactly what would run. `hrx doctor` checks configuration, credentials, gateway reachability and authorization without sending an inference request. Remote launches have `--plan`. The sweeper only plans unless you name, one by one, what it may remove.

Records outlive sessions

An agent identity outlives any one harness session. A remote deployment keeps a durable record of its container, lease, session and credentials so it can be inspected, resumed or destroyed by name, even after your laptop restarts. Recovering from a failure means rerunning the same command with the same name, never deleting state by hand.

Why not just a shell alias?

A shell alias can save typing: `alias cs='codex -m gpt-6.1-sol'`. It cannot tell you that your key lost access to that route yesterday, that the route exists on two accounts, or that the harness you are about to start has never been declared compatible with it. Each of those failures turns into a confusing error minutes into a session, or into no error at all and a surprising bill. A launcher is worth having precisely because it checks things an alias cannot see.

How the Pieces Fit

Figure 1-1 shows the path of a single launch.

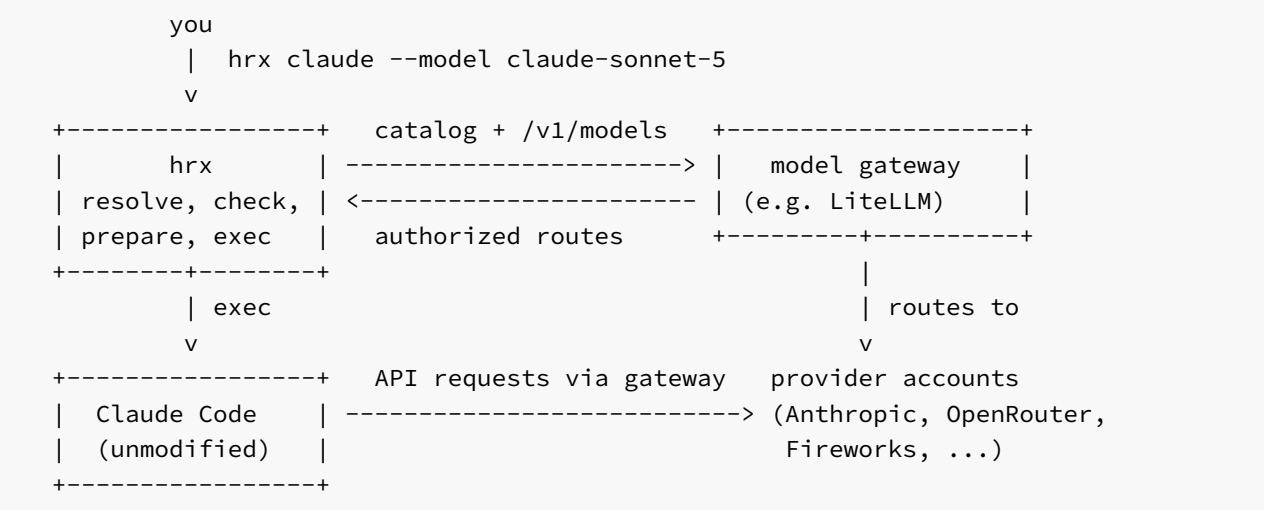

Figure 1-1. One launch: hrx validates against the gateway, then becomes the harness.

The gateway publishes two things hrx relies on. The first is the standard OpenAI-compatible `/v1/models` endpoint, which answers "which routes may this key call?" The second is a catalog, served at `/dashboard/models.json`, which answers "which models exist, through which routes, and which harnesses has each route been declared compatible with?" The catalog is what lets hrx tell the difference between a model you cannot use and a model nobody has checked yet. Chapter 3 reads it in detail.

When an agent needs more than a model, later chapters add more boxes to this picture: a message board that delivers notifications into the session, a credential broker that issues Git tokens, a Proxmox cluster that hosts remote workers, and a dashboard that watches them all. The launch path stays the same. hrx validates, prepares and then gets out of the way.

What hrx Is Not

It helps to be clear about the edges.

- **hrx is not a harness.** It has no agent loop, no tools and no prompt of its own beyond optional guidance it adds for communication. The model talks to Codex, Claude Code or OpenCode, exactly as it would without hrx.

- **hrx is not a gateway.** It does not route, retry or fail over requests. Public aliases on your gateway keep their failover policy; hrx just refuses to guess between routes the gateway has not aliased.
- **hrx is not an orchestrator that writes your prompts.** A network file says which agents to start, with what models, identities and boards. What they do is up to the prompts you give them and the conversations they have.
- **hrx does not grant permissions.** It never widens a harness's filesystem or command permissions. The one permission switch it offers, `--yolo`, maps to each harness's own approval setting and is off unless you ask.

Summary

Running agents well is mostly a matter of knowing, at every moment, which model each agent is using, who is paying, what the agent is allowed to touch, and how to stop it. hrx makes those facts explicit and checks them before a launch rather than after a failure. The next chapter installs it and puts the first of those checks to work.

CHAPTER 2

Installing hrx and Launching Your First Agent

In this chapter you will build and install hrx, point each harness at your gateway, run the read-only health check, and launch a real session. Along the way you will meet the three habits the rest of the book relies on: refresh, dry-run, then launch.

> **Warning**
>
> Most commands in this chapter are free: they read configuration or ask the gateway what your key may do. The few that start a harness send real inference requests and spend money. They are marked when they appear.

What hrx Expects from Your Gateway

hrx does not ship a model list. Everything it knows about models comes from your gateway at run time, so the gateway has to publish two things:

1. **The standard `/v1/models` endpoint.** Called with your key, it returns the routes that key may call. Every OpenAI-compatible gateway has this.
2. **A catalog at `/dashboard/models.json`** on the same host. For each model identity it lists the routes that serve it, the provider account behind each route, context and output limits, reasoning levels, and which harnesses each route has been declared compatible with, optionally with the date of the last live test.

The first tells hrx what you *may* use. The second tells it what *exists* and what *works with which harness*. hrx intersects them. A route you are allowed to call but that nobody has declared compatible with Codex will not be offered to Codex. A model in the catalog that your key cannot reach is listed as `not-authorized`, so you can tell "missing" apart from "forbidden".

> **Note**
>
> The catalog format is documented in Appendix B, with a small script that builds one from a LiteLLM configuration. If you run LiteLLM already, publishing the catalog is a matter of serving one more JSON file from the gateway host.

The gateway must be reachable over HTTPS. hrx refuses a base URL that is plain HTTP, embeds credentials, or carries a query string or fragment.

Building and Installing

hrx is a single Go binary. You need Go 1.26 or later and `make`:

```
git clone https://example.com/hrx.git
cd hrx
make test
make install
```

`make install` puts the binary in `~/.local/bin/hrx` and a zsh completion script in `~/.local/share/zsh/site-functions/_hrx`. To install somewhere else, set the prefix:

```
make install PREFIX=/opt/hrx
```

Make sure the `bin` directory is on your `PATH`, then check the version:

```
$ hrx --version
hrx 0.1.2 (dbf201ff1db4)
```

The part in parentheses is the Git revision the binary was built from. Every public command also prints this line to standard error, which keeps standard output clean for JSON and scripts. When you report a problem, or compare your output with this book, start with this line.

> **Warning**
>
> Once you run agents under hrx, the running sessions call the installed `hrx` binary from their hooks. Replacing the binary in place while sessions are running can crash them on some systems. Build the new binary next to the old one and rename it over the top, which is atomic:
>
> ```
> go build -o ~/.local/bin/.hrx.new ./cmd/hrx
> ~/.local/bin/.hrx.new --version
> mv -f ~/.local/bin/.hrx.new ~/.local/bin/hrx
> ```

Tab completion

hrx's completion offers models and reasoning levels from its local cache, so it works offline and never calls the gateway while you type. For zsh, add the site-functions directory to `fpath` before completion is initialized (before Oh My Zsh or `compinit`):

```
fpath=("$HOME/.local/share/zsh/site-functions" $fpath)
autoload -Uz compinit
compinit
```

Completion works with plain zsh. It is noticeably better with fzf-tab, which turns the candidate list into a fuzzy picker and shows each model's authorization and test status next to its name. If you use fzf-tab, load it after `compinit` and before any autosuggestion or syntax-highlighting plugins.

Connecting Each Harness to the Gateway

hrx reuses the gateway configuration and credential you already have for each harness. It never creates a key and never copies a key into its own files. What it needs is slightly different for each harness.

Codex

Codex keeps its providers in `$CODEX_HOME/config.toml` (by default `~/.codex/config.toml`). hrx uses the provider named by the top-level `model_provider` key. That provider must use the Responses API:

```toml
model_provider = "gateway"

[model_providers.gateway]
name = "Model gateway"
base_url = "https://gateway.example.com/v1"
wire_api = "responses"
env_key = "GATEWAY_API_KEY"
```

The credential can be an environment variable (`env_key`, as here) or a helper command that prints the key. To use a different provider without changing `model_provider`, set `HRX_CODEX_PROVIDER=NAME` or, permanently, `provider_name` under `[codex]` in hrx's own configuration.

To keep two Codex setups side by side, point `CODEX_HOME` at the other one. hrx keeps a separate cache for each configuration, so they never mix:

```
CODEX_HOME="$HOME/.codex-work" hrx refresh
CODEX_HOME="$HOME/.codex-work" hrx codex --model gpt-6.1-sol
```

OpenCode

hrx reads the `litellm` provider from your existing OpenCode configuration in `~/.config/opencode/opencode.json`:

```jsonc
{
  "provider": {
    "litellm": {
      "options": {
        "baseURL": "https://gateway.example.com/v1",
        "apiKey": "{env:GATEWAY_API_KEY}"
      }
    }
  }
}
```

The key can be inline, an `{env:NAME}` reference or an absolute `{file:/path}` reference. At launch, hrx gives OpenCode a separate, temporary provider that reads the key from the child process's environment only. Your original file is never rewritten, and only that one provider is enabled for the session.

Claude Code

Claude Code has no gateway provider of its own to reuse, so you describe the gateway in hrx's configuration file, `~/.config/hrx/config.toml` (or `$XDG_CONFIG_HOME/hrx/config.toml`, or the path in `$HRX_CONFIG`):

```toml
[claude]
model = "claude-sonnet-5"

[claude.provider]
base_url = "https://gateway.example.com/v1"
env_key = "GATEWAY_API_KEY"
```

Give exactly one credential handle:

- `env_key`: the name of an environment variable that holds the key.
- `credential_file`: a private JSON file of the form `{"key": "..."}`, readable only by you (mode `0600`).
- An `[claude.provider.auth]` table with `command`, `args` and optional `timeout_ms`: a helper program that prints the key. This is the best choice when the key lives in a password manager or on another machine.

The same `[opencode]` and `[opencode.provider]` tables override OpenCode's settings if you prefer to keep everything in one place.

> **Warning**
>
> Never put a literal key in hrx's configuration file or in a helper's arguments. Configuration files get copied, committed and pasted into chats. A handle that points at the key is safe to share; the key is not.

At launch hrx clears inherited Claude Code routing and API-key settings and pins the session, including subagents, to the selected route. If your Claude Code settings contain conflicting routing or an `apiKeyHelper`, hrx stops and explains the conflict instead of silently changing how you authenticate.

Refresh: Fetching the Catalog

With at least one harness connected, fetch the catalog:

```
hrx refresh
hrx refresh --harness claude
hrx refresh --harness opencode
```

Plain `hrx refresh` refreshes Codex. Each harness has its own cache, because each has its own key and its own compatibility rules. A refresh contacts the gateway, runs your credential helper if you have one, and writes a cache file under `~/.cache/hrx` with mode `0600`. The cache contains route names and capabilities, never key values.

The cache is considered fresh for 30 minutes. After that, Tab completion shows nothing rather than offering stale choices. This is deliberate: an empty menu is a prompt to refresh, while a stale one invites you to pick a route that may no longer work.

Doctor: The Read-Only Health Check

Before your first launch, and whenever something feels wrong, run the doctor:

```
hrx doctor
hrx doctor --harness claude
hrx doctor --harness codex --model gpt-6.1-sol --reasoning high
hrx doctor --harness opencode --json --timeout 20
```

The doctor checks, for each harness: that the executable exists and reports a version; that the configuration parses; whether the cache is present and fresh; that the credential handle resolves; that the gateway is reachable; that the gateway accepts your key; and that the selected or default route is authorized and declared compatible. With `--model` it also checks the reasoning level.

Doctor sends **no inference requests** and has no option to. It runs harnesses only with `--version`, reads your key in memory, and makes `GET` requests to the catalog and `/v1/models`. It does not write the cache, edit configuration or start an agent. The whole run is bounded, 45 seconds by default.

Output is one line per check, with a fix where there is one:

```
FAIL codex     authorization  not-authorized: Gateway rejected this key (HTTP
 ↪  401/403).
     Fix: Check the existing key expiry, budget and authorization; do not rotate
 ↪  automatically.
WARN codex     cache          stale: Completion cache is stale; launch will revalidate.
     Fix: Run hrx refresh --harness codex
```

`FAIL` means a launch would fail; `WARN` means it would probably succeed but something deserves attention. The exit status is 0 when nothing failed, 1 when a check failed and 2 for bad arguments, so the doctor works in scripts and CI. `--json` prints a stable structure with a `schema_version`, an overall `ok`, and one entry per check.

> **Tip**
>
> An accepted key is not proof that inference will work. Budgets, concurrency limits and provider outages can still stop a request, and the doctor cannot see them from a `/v1/models` response. If a launch passes every check and then hits HTTP 429, the limit is on the gateway or provider side.

Dry Run: See the Launch Before You Pay for It

Every launch command accepts `--dry-run`. It does everything a launch does, including the live authorization check, and then prints the result instead of starting the harness:

```
hrx codex --model gpt-6.1-sol --reasoning high --dry-run
hrx claude --model claude-sonnet-5 --dry-run
hrx opencode --model glm-5p3 --dry-run
```

The output shows the harness, the exact route the name resolved to, the provider account behind it, the reasoning level and where each default came from. It never shows credential values or the arguments of your credential helper.

Make dry-run a habit for any launch you have not done before. It costs nothing, and it catches the two most expensive mistakes: the wrong route, and the wrong payer.

Your First Session

Now launch for real. This starts a harness and spends money.

```
cd ~/src/my-project
hrx claude --model claude-sonnet-5
```

hrx validates the route, prepares Claude Code's environment, and replaces itself with `claude`. From here on you are in an ordinary Claude Code session. When you exit, the shell gets Claude Code's exit status, because hrx is no longer running.

The other harnesses work the same way:

```
hrx codex --model gpt-6.1-sol --reasoning high
hrx opencode --model glm-5p3
```

If you leave out `--model`, hrx uses the harness's default: Codex's `model` setting, or the `model` key in the harness's hrx section. OpenCode has no default unless you set one, so it needs `--model`.

Passing arguments to the harness

Options for hrx go first. Anything after `--` goes to the harness unchanged. This is how you run one-shot tasks:

```
# Codex: a read-only exploration with the exec subcommand.
hrx codex --model gpt-6.1-sol --reasoning high -- exec --sandbox read-only \
  'Explain the entry points and the test layout. Do not change files.'

# Claude Code: print mode.
hrx claude --model claude-sonnet-5 -- -p 'Summarize the open TODOs in this repo.'
```

```
# OpenCode: the run subcommand, without external plugins.
hrx opencode --model glm-5p3 -- run --pure 'Describe the project structure.'
```

hrx does not add permissions. The first example is read-only because it asks Codex for a read-only sandbox, not because of anything hrx did.

hrx rejects passthrough options that would change the model or provider, such as Codex's `-m` or `--profile`, Claude Code's `--model` or OpenCode's agent selection. They would bypass the route hrx just validated. Choose the model through hrx instead.

Permission prompts

Each harness asks before running commands or editing files, unless you tell it not to. hrx's `--yolo` maps to each harness's own switch:

hrx	What the harness receives
`hrx codex --yolo`	`codex --yolo`: no approvals and no sandbox
`hrx claude --yolo`	`claude --dangerously-skip-permissions`
`hrx opencode --yolo`	`opencode --auto`: explicit deny rules still apply

To make it the default, set `yolo = true` under `[permissions]` in hrx's configuration; `--no-yolo` then restores prompts for a single launch.

> **Warning**
>
> Removing permission prompts lets the model run any command your user can. Use it in a disposable checkout, a container, or a remote worker (Chapter 8), and never on a repository or machine you cannot afford to lose.

Resuming a session

Session IDs belong to the harness, but hrx gives all three the same options:

```
hrx claude --model claude-sonnet-5 --resume SESSION_ID
hrx codex --model gpt-6.1-sol --continue
```

`--resume` reopens one exact session; `--continue` picks the most recent one for this project. hrx translates them into each harness's native form, and for OpenCode it first checks that the session exists and belongs to this project, so a typo cannot silently start a new session.

The Whole Command Line

Here is hrx's complete usage summary. Most of it is the subject of later chapters; for now, notice the shape: a harness or subcommand, hrx options, and then `--` and whatever the harness should receive.

```
hrx - harness launcher for Codex, Claude Code and OpenCode through one model gateway

  hrx --auto "claude as host/role/agent using pitput" [LAUNCH_OPTIONS] [--print]
  hrx --profile NAME [LAUNCH_OPTIONS] [-- HARNESS_ARGS...]
  hrx run --backend=proxmox --host HOST --profile NAME --agent-identity ID
↪  --pitput-board BOARD --prompt-file FILE
  hrx remote inspect|logs|attach|stop|resume|replace-lease|destroy --name NAME
  hrx network --file NETWORK.toml [--plan] [--only ROLE]
  hrx sweep [--host USER@NODE]... [--dashboard URL] [--idle 24h] [--json] [--apply
↪  ID]... [--backup-storage local]
  hrx profiles list|show NAME|validate NAME [--all] [--offline] [--json]
  hrx doctor [--harness NAME] [--model NAME] [--yesod-comms] [--json]
  hrx refresh
  hrx dashboard [--listen 127.0.0.1:8789] [--source NAME=URL] [--hub]
  hrx hub serve|enroll|status|revoke|rotate|collect
  hrx concierge --input "watch /tmp/x for writes, summarize, post to /b/x, expire in
↪  24h" [--dry-run|--yes]
  hrx concierge --list | --logs ID | --kill ID | --prune
  hrx matrix --input FILE
  hrx models [--refresh] [--json] [--stale]
  hrx codex [--model NAME] [--reasoning LEVEL] [--dry-run] [-- CODEX_ARGS...]
  hrx claude [--model NAME] [--reasoning LEVEL] [--dry-run] [-- CLAUDE_ARGS...]
  hrx opencode [--model NAME] [--reasoning LEVEL] [--dry-run] [-- OPENCODE_ARGS...]
  hrx models|refresh --harness codex|claude|opencode
  hrx catalog [--compact|--details] [--latest] [--refresh] [--stale] [--json]
↪  [--search TEXT] [--harness all|codex|claude|opencode] [--output FILE]
  hrx completion zsh
  hrx prime

Launch options (all three harnesses):
  --profile NAME              Use a validated HRX launch profile
  --credential-file PATH      Override gateway key with private JSON key file
  --gitlab                    Enable GitLab service accounts (broker plugin alias)
  --plugin NAME               Enable gitlab-service-account-broker
  --agent-identity ID         Broker-authorized agent identity (required with plugin)
  --pitput                    Enable Pitput (automatic delivery for Codex)
  --pitput-board BOARD        Enroll/monitor a board; repeat for multiple boards
  --pitput-combined-preview    Opt in to combined Codex board/DM/ask delivery
↪  (preview)
  --pitput-targeted           Notify for recipient mentions (new-session default)
  --pitput-all-activity       Explicitly monitor all activity on selected boards
  --pitput-follow-thread PATH Also notify for this topic; implies targeted selection
  --pitput-no-dm              Do not deliver private Pitput direct messages
  --pitput-dm-only            Deliver only direct messages (no boards; DM-only
↪  enrollment)
  --yesod-mail | --yesod-bbs | --yesod-comms
  --yesod-agent ADDRESS       Reuse an explicitly chosen registered identity
  --yolo                      Disable approval prompts (OpenCode respects denies)
```

```
  --no-yolo                   Override permissions.yolo and keep approval prompts
  --resume SESSION_ID         Resume a specific harness session
  --continue                  Continue the latest harness session

Claude only:
  --claude-config-dir PATH    Use an HRX-managed Claude config that HRX creates
↪  (never yours)
  --claude-init PARTS         Initialize first-run answers once: theme,trust,bypass

HRX settings: ~/.config/hrx/config.toml (or HRX_CONFIG / XDG_CONFIG_HOME).
Codex provider definitions: $CODEX_HOME/config.toml (default ~/.codex/config.toml).
Tab completion and models use a local cache; launches refresh and fail closed.
Put hrx options before harness arguments; use -- before Codex subcommands.
Use --latest on models/catalog, or before --model for latest-only Tab suggestions.
Explicit older models remain launchable; omit --latest to show everything.
```

Summary

You installed hrx, connected each harness to your gateway with the credential it already used, and learned the rhythm that the rest of the book assumes:

1. `hrx refresh` to fetch what your key can use.
2. `hrx doctor` when something seems wrong.
3. `--dry-run` before any launch you have not done before.
4. Launch, with hrx options before `--` and harness options after.

So far you have been choosing models by name and trusting that the name resolved sensibly. The next chapter looks at how that resolution works, how to read the catalog, and what to do when the model you want is not there.

CHAPTER 3

Models, Routes and Reasoning

Choosing a model used to mean picking a name from a short list. Behind a gateway it means three decisions at once: which model, which route to it (and so which account pays), and how hard it should think. This chapter shows how hrx turns the name you type into one exact route, how to read the catalog when you want to know what is available, and how to work out why a model you expected is missing.

Everything in this chapter is read-only. None of it sends an inference request.

Identities, Routes and Aliases

Three words carry most of the weight in this chapter.

A **model identity** is the model itself, independent of who serves it: `gpt-6.1-sol`, `claude-sonnet-5`, `glm-5p3`. The catalog groups everything by identity.

A **route** is one exact name the gateway accepts in a request, bound to one provider account. The same identity can have several routes:

Route	Served by
`gpt-6.1-sol`	The gateway's public alias, with its own failover policy
`tfy-acme/gpt-6.1-sol`	A TrueFoundry account named `acme`
`or-acme/openai/gpt-6.1-sol`	An OpenRouter account named `acme`

Routes differ in price, rate limits, context limits and, importantly, in which harnesses can drive them. A route that works with Codex's Responses API may not work with OpenCode's Chat Completions client, and vice versa.

A **public alias** is a route the gateway operator has deliberately published under the model's plain name. The gateway may fail over between providers behind an alias. hrx keeps aliases intact: if you ask for `gpt-6.1-sol` and the gateway publishes that alias, that is what the harness receives, and the gateway's failover keeps working.

How a name is resolved

When you pass `--model NAME`, hrx looks for, in order:

1. An exact route called `NAME` that your key may call and this harness is declared compatible with. If it exists, that is the answer.
2. A model identity called `NAME` with exactly one such route, or a public alias. That route is the answer.
3. Otherwise, the launch fails.

There is no step 4. If an identity has two usable routes and no public alias, hrx does not pick one; it prints both and asks you to choose:

```
hrx codex --model tfy-acme/gpt-6.1-sol --reasoning high --dry-run
```

The same applies to names that are close but not equal. hrx never treats `gpt-6.1` as `gpt-6.1-sol` or substitutes a newer model for a retired one.

Why ambiguity is an error

Two routes to the same model can differ by an order of magnitude in price, or bill two different budgets: a personal account and a team account, say. A tool that chooses silently will eventually choose the expensive one for a six-hour agent run. Making you type the route once costs a few characters; the alternative costs a surprise on the invoice.

Listing What You Can Use

`hrx models` lists what a harness can launch right now, one line per choice:

```
hrx models
hrx models --harness claude
hrx models --harness opencode --latest
hrx models --json
```

It reads the local cache, so it is instant and works offline. Add `--refresh` to update the cache first. If the cache is older than 30 minutes the command refuses, rather than showing choices that may no longer be valid; `--stale` shows them anyway, clearly marked, which is useful during a gateway outage when you want to know what *was* available. A stale listing never authorizes a launch.

`hrx models` answers "what can I start?" When you want to know why something is *not* on that list, you need the catalog.

Reading the Catalog

`hrx catalog` shows every model the gateway discovered, including the ones you cannot use, together with the reason:

```
hrx catalog --refresh --compact
hrx catalog --compact --latest --search gpt-
hrx catalog --harness codex --search glm
```

```
hrx catalog --details --harness opencode --search deepseek
hrx catalog --output model-catalog.json
```

The catalog has three text views and one data view.

Compact (`--compact`) prints one row per model identity, with the provider accounts that serve it and, for each harness, the accounts with at least one ready route:

```
MODEL        SOURCES          codex    claude  opencode  NEWER FAMILY MODEL
example-sol  OR-A,FW-A,TF-A   TF-A[D]  -       OR-A[D]   -
```

Account names are abbreviated (here `OR-A` might be OpenRouter/acme), and a legend under the table explains only the abbreviations that appear. A `-` means this harness has no ready route to the model.

Route level (the default) prints one row per route, with each harness's status for that route.

Details (`--details`) adds everything hrx knows about each route: the upstream model, the wire API, per-harness authorization and compatibility, reasoning choices and defaults, context and output limits, and test dates.

JSON (`--json` or `--output FILE`) exports the full underlying evidence, whatever text view you chose. The export contains no credentials, so you can attach it to a bug report or keep it as a record of what was available on a given day.

`--search TEXT` matches model names, routes and account names. Searching for `codex` therefore finds models with "codex" in their name; it does *not* mean "models usable in Codex." For that, use `--harness codex`, or `hrx models`.

Route statuses

Every route has a status for each harness. These five words explain almost every "where is my model?" question:

Status	Meaning	What to do
`ready`	Your key may call it and the gateway declares it compatible with this harness	Launch it
`not-authorized`	The route exists, but your key may not call it	Ask the gateway operator; refreshing cannot widen a key
`compatibility-unverified`	Your key may call it, but nobody has declared it compatible with this harness	Use another harness, or get the route tested and declared
`no-gateway-route`	A provider offers the model, but the gateway has no route for it	Ask the operator to add a route
`unavailable`	The catalog marks the model or its source unavailable	Wait, or choose another route

The difference between `not-authorized` and `compatibility-unverified` matters. A key expansion can move a route from the first to the second without making it launchable. Authorization says you may *pay* for the route; compatibility says the harness can *use* it.

> **Note**
>
> Gateways often grant access with wildcards, such as `tfy-acme/*`. hrx honors those only for concrete routes the catalog has discovered under that account. A wildcard never grants access to another account, or to a model nobody has discovered.

Evidence badges

Declaring a route compatible is a claim. Testing it is evidence. The catalog can record when a route was last live-tested with each harness, and hrx shows that as a badge next to the account:

Badge	Meaning
`D`	Declared compatible; no live test recorded
`V`	Live-tested within the last 7 days
`S`	Live-tested, but more than 7 days ago
`U`	A test is claimed, but its date is missing, invalid or in the future
`cached-`	Prefix shown when the listing comes from stale cache

The 7-day window is a display policy, not a rule: a stale test does not make a route unlaunchable. hrx never invents test dates, and refreshing authorization does not refresh them. Only the catalog's own dated, per-harness evidence can produce a `V`.

Keeping Up with New Models: `--latest`

Providers release new generations often enough that a full list fills up with superseded models. `--latest` hides older numeric generations within the same family and variant:

```
hrx catalog --latest --search gpt-
hrx models --harness opencode --latest
hrx opencode --latest --model <Tab>
```

With `--latest`, `gpt-6.1-sol` hides `gpt-6-sol`, but not a different variant such as `gpt-6.1-terra`. Versions compare numerically, so `5.10` is newer than `5.9`. Names hrx does not recognize, dated snapshots and distinct suffixes stay visible.

Two safeguards keep this from hiding something you need. First, an older generation stays visible if it is the newest one *this harness* can actually use; a successor you cannot launch never hides your only working choice. Second, `--latest` only changes what is listed. An explicit `--model` for an older generation still launches.

The `NEWER FAMILY MODEL` column tells you when a newer generation exists, even if you cannot use it yet. That is a hint, not a recommendation: newer is not always better for a given task, and it is not always cheaper.

Reasoning Levels

Many models accept a reasoning effort, from `low` to `xhigh` or `max`, that trades latency and cost for more deliberate answers. Which levels exist depends on the model *and* the harness, so hrx reads them from the catalog rather than assuming:

```
hrx codex --model gpt-6.1-sol --reasoning high
hrx codex --model gpt-6.1-sol --reasoning <Tab>
```

The reasoning picker follows the model you selected and marks the catalog's declared default. If you leave `--reasoning` off, the harness's normal default applies, as declared by the catalog. If you ask for a level the route does not advertise, the launch fails; hrx does not round to the nearest level or drop the flag silently.

At the time of writing, reasoning selection is advertised for Codex routes. The Claude Code and OpenCode adapters launch with the provider's default and reject `--reasoning` with a clear error rather than ignoring it.

> **Tip**
>
> Higher reasoning is not free and not always better. For agent work, a common pattern is a fast, low-effort model for routine roles and a high-effort one for review and planning. Chapter 4 shows how to name those combinations once and reuse them.

Choosing Interactively

Tab completion offers the same choices as `hrx models`, filtered for the harness on the command line:

```
hrx codex --model <Tab>
hrx codex --model glm-5p3 --reasoning <Tab>
hrx claude --model <Tab>
hrx opencode --latest --model <Tab>
```

With fzf-tab, each candidate shows its authorization, compatibility evidence, test date and any newer family model. The descriptions are for display only: choosing an entry inserts just the route or the level.

Completion reads only the local cache. It never runs your credential helper, opens a network connection or contacts the gateway, so it is fast and safe to press repeatedly. An empty menu means the cache is missing or older than 30 minutes; refresh the harness you are completing for.

When a Model Is Missing

Here is a short procedure for the most common question.

1. **Refresh the right harness.** Each harness has its own cache: `hrx refresh --harness opencode`.
2. **Check the launchable list.** `hrx models --harness opencode`. If the model is there, use the exact name shown.
3. **Find out why not.** `hrx catalog --refresh --harness opencode --search NAME` and read the status column.
4. **Ask the doctor about that exact choice.** `hrx doctor --harness opencode --model NAME`.

Symptom	Likely cause
Model absent from `hrx catalog`	The gateway never discovered it; check the gateway, not hrx
`not-authorized`	Your key's scope; only the gateway operator can widen it
`compatibility-unverified`	Not declared for this harness; try another harness
"ambiguous" error	Several usable routes; pass the exact route
Empty Tab menu	Stale or missing cache; `hrx refresh --harness NAME`
Unsupported reasoning level	Choose one the route advertises (`--reasoning <Tab>`)
HTTP 429 after launch	A gateway or provider limit; `ready` does not promise capacity

How Fresh Is Fresh?

The rules for staleness are simple and strict:

- The cache is fresh for 30 minutes, counting the age of the gateway's own catalog snapshot.
- Completion shows nothing from a stale cache; `hrx models` and `hrx catalog` refuse unless you pass `--stale`.
- Every launch revalidates against the live gateway. There is no option to launch from stale data.
- A failed refresh keeps the last good cache, so a gateway hiccup cannot leave you with an empty or corrupt one.

The effect is that the listing you look at may be slightly old, but the launch you start never is.

Summary

A model name in hrx resolves to exactly one authorized, harness-compatible route, or the launch fails and tells you why. `hrx models` shows what you can launch, `hrx catalog` shows everything and explains the gaps, and `--latest` keeps the list short without hiding your only working option. Reasoning levels come from the catalog and are checked, never guessed.

You now have everything you need to launch one agent deliberately. Part II turns those deliberate choices into reusable profiles, gives agents names, and connects them to each other.

PART III

Agents at Scale

More agents than you can watch by hand, running in more places than your own terminal. This part is about work that carries on while you are not looking: who is running, what each one may touch and spend, and how everything is stopped and cleaned up when it is done.

CHAPTER 10

Concierge: Small Triggered Workers

Not every job needs an agent session. "Tell the board when the nightly report changes" or "warn us when the error log shows a timeout" needs something much smaller: a process that sleeps until one thing happens, makes one short model call, posts the result and eventually goes away by itself. `hrx concierge` creates exactly that from a sentence. By the end of this chapter you will be able to request a worker, read the plan hrx shows you before it starts anything, manage and remove workers, and decide which files and boards it is safe to point one at.

> **Warning**
>
> A concierge worker spends money and publishes text. Every model run is paid for by a gateway key created for that worker, and whatever the worker reads can end up on a message board. The plan hrx shows before it spawns anything is where you check both.

What a Concierge Worker Is

A concierge worker is a small background program with four fixed properties, all settled when it is created:

- **One trigger** that wakes it: a file being written, a line appearing in a log, a schedule, or a post in a message-board topic.
- **One task**, in your words, performed with a single bounded model run each time the trigger fires.
- **One place to post** the result: a topic on a message board.
- **One lifetime**, from one minute to seven days, after which it says goodbye and exits.

It is not a harness session. It has no shell, no file editor and no conversation. The model inside it gets two tools at most: post to the worker's own topic, and re-read the end of the one file it watches.

	Agent session	Concierge worker
Started with	`hrx claude`, `hrx codex`, `hrx opencode`	`hrx concierge --input "..."`
Can do	Whatever the harness and its permissions allow	Read one file, post to one topic
Runs	Until you end the session	Until its lifetime ends

	Agent session	Concierge worker
Model use	A long conversation	One short run per trigger
Credential	The gateway key you already use	Its own key: one route, a daily budget, an expiry

Reach for a worker when the job is "when X happens, say something about it there." Reach for an agent when the job needs judgment across many steps, or needs to change anything.

`hrx concierge` itself is not the worker. Like `hrx --auto`, which turns a sentence into a launch command, it is a one-shot translator: it sends your request to a model once, turns the reply into a plan, asks you to confirm, and exits. It never stays running. What stays running is the worker it started.

What You Need

The chapter builds on the gateway connection from Chapter 2 and adds four things:

- **The worker program.** `make install-concierge` installs it as `hrx-concierge-worker` next to `hrx`. It is a Python script that runs under uv, which fetches its two pinned dependencies the first time it starts.
- **A key-issuing service on your gateway**, which this chapter calls the *registrar*. hrx asks it for one key per worker. Credentials describes the two requests it must answer.
- **Pitput**, the message-board service hrx's communication features use, with its `pitput` command installed and this machine allowed to enroll new identities on the boards you name. Board paths look like `/b/team-board`, and a topic inside a board like `/b/team-board/nightly`.
- **A `[concierge]` table** in hrx's configuration, covered in Configuration. At minimum it names a low-cost model route and the registrar.

The full option list is short:

```
Usage of hrx concierge:
  -dry-run
        with --input: show the planned worker and stop
  -input string
        natural-language request describing the worker to spawn, e.g. "watch /tmp/x
↪  for writes, summarize, post to /b/x, expire in 24h"
  -kill string
        stop a spawned concierge worker by ID
  -lines int
        with --logs: how many lines to print (0 for all retained) (default 200)
  -list
        list spawned concierge workers
  -logs string
        print the end of a worker's log by ID
  -prune
        revoke keys and remove records of workers that are no longer running
  -yes
        with --input: spawn without asking (for scripts and agents; preview with
↪  --dry-run first)
```

The listing shows the options with one dash because that is how hrx's flag parser prints them. Two dashes work too, and this chapter uses them.

A First Worker

Suppose a build job rewrites `/srv/ci/status.txt` whenever it finishes, and you want a two-line summary on the team's board each time, for the next day. Say so:

```
hrx concierge --input "watch /srv/ci/status.txt, summarize it in two lines \
when it changes, post to /b/team-board, expire in 24h"
```

hrx sends the request to the gateway to be turned into a plan, using the same planning route as `--auto`. It tells you so first, with a reminder not to include secrets in the request. Then it prints the plan and asks. The plan looks like this; the ID, times and names are illustrative, and yours will differ:

```
hrx: Proposed concierge worker:
  wakes     when /srv/ci/status.txt is written; also when @-mentioned on /b/team-board
  reads      /srv/ci/status.txt (its contents can be posted)
  task       summarize it in two lines when it changes
  posts to   /b/team-board/concierge-3f9c2a7e (a new topic on /b/team-board)
  lifetime   24h0m0s, until 2027-03-02 09:00 PST
  model      fw-acme-deepseek-v4p1-flash, budget $1.00 per day
Spawn? [y/n/reprompt]
```

Read every line. Each one is a decision the model made from your sentence, checked by hrx against its own rules:

Line	What to check
`wakes`	The trigger, and the board on which anyone can wake the worker by mentioning it
`reads`	The exact file, with symbolic links already resolved. Shown for file and log triggers only
`task`	Your instruction, and nothing you did not ask for
`posts to`	The topic, and so who will see the result
`lifetime`	How long it runs. A request that names no lifetime gets 24 hours
`model`	The route and daily budget from your configuration; the request cannot change these

Then answer:

- `y` spawns the worker. The lifetime starts now, not when the plan was drawn.

- `n` cancels. Nothing was created.
- `reprompt` (or `r`) asks for a correction, such as "every change, not just the first" or "post to the nightly topic instead", and plans again with it. After eight proposals hrx stops and asks you to start over with a more specific request.

On `y`, hrx obtains a key for the worker, records the worker, starts it detached from your terminal, waits two seconds to see that it is still alive, and prints its ID, its topic and its expiry time. You can close the terminal. The worker now checks its trigger every five seconds until its lifetime ends.

When that time comes it posts a fixed goodbye to its topic, with no model call, gives up its message-board identity and exits.

Dry runs and `--yes`

Two flags change the confirmation step:

```
hrx concierge --input "..." --dry-run    # print the plan and stop
hrx concierge --input "..." --yes        # spawn without asking
```

`--dry-run` plans, applies the path and board rules, prints the plan and stops. No key is requested and nothing is started. Unlike the launch dry runs in Chapter 2, it is not quite free: planning is one small model request.

`--yes` skips the question. Without a terminal there is nobody to ask, so hrx refuses to spawn unless you pass `--yes`.

> **Tip**
>
> An agent or a script should always run the request with `--dry-run` first, read the plan, and only then repeat the same request with `--yes`. The request an agent passes along may contain text it did not write: a line from an issue, a file name from a web page. The plan is the only place where "which file gets read, and where it gets published" is stated plainly before it happens.

Planning runs again for the `--yes` call, so compare the second plan with the first if anything depends on it.

The Four Triggers

The trigger is chosen by how you phrase the request. The planner may only pick one of four kinds, and hrx validates the result:

Say	The worker wakes	Details
"watch FILE"	When the file is written	An absolute path. The file may not exist yet; its directory must
"when LOG shows TEXT"	When a new line in the log contains the text	A plain substring, not a pattern language. Lines already in the log do not count

Say	The worker wakes	Details
“every 30 minutes”, “every 2 hours”, “every day at 09:00”	On that schedule	Intervals count from when the worker starts and cannot be shorter than 5 minutes. A daily time is local time
“when anyone posts in /b/BOARD/TOPIC”	When someone other than the worker posts in that topic	Must name one topic, not a whole board

When a trigger fires, the worker first does one fixed step with no model involved: it reads the end of the file (at most 16 KB), collects the new log lines, or reads the post. That material, plus your task, goes to the model in one run, and the model posts its answer. For a file trigger the worker reads the file when the run starts, so several quick writes produce one summary of the latest contents rather than one per write.

A log worker survives rotation: if the file shrinks it starts again from the beginning. A schedule check that comes late runs once, not once per missed slot. A schedule delivers no material at all, only the fact that it came due, so a scheduled worker’s task has to stand by itself: a reminder, say, not a report on something the worker would have to go and read.

The mention listener

Whatever its trigger, every worker also listens for `@`-mentions of its own identity on the board it posts to. The plan’s `wakes` line says so. The worker’s identity has the form `HOST/concierge-worker/ID`, for example `laptop/concierge-worker/3f9c2a7e5b1d4c8fa0e6d2b79c4e1f03`.

A mention is for follow-up: “shorter, please”, or “which test failed?” The worker answers from the message alone. It does not get the tool to re-read its file when a board member woke it, so a mention cannot be used to pull the file’s contents onto the board. Anything outside its task it declines.

Where It Posts

Name either a board or one topic in a board:

You say	The worker posts to
`/b/team-board`	A topic of its own, `/b/team-board/concierge-` followed by the first eight characters of its ID
`/b/team-board/nightly`	That topic

Posts live in topics, not directly on boards, which is why a bare board gets a new topic. The worker opens the topic with its first post if it does not exist yet. The plan’s `posts to` line shows the result either way and tells you when the topic is new.

Managing Workers

Four more flags cover a worker's life after it starts.

Listing

```
hrx concierge --list
```

The listing has one row per recorded worker under these columns:

```
ID  POSTS TO  TRIGGER  CREATED  EXPIRES  STATUS
```

`TRIGGER` repeats the plan's wording, and the times are local. The status is one of six:

Status	Meaning
`starting`	Recorded less than a minute ago; its process is not registered yet
`running`	Alive and within its lifetime
`expiring`	Alive but past its lifetime: it is saying goodbye
`expired`	Exited after its lifetime ended
`gone`	Exited early: it crashed, was stopped from outside, or the machine restarted
`unreadable`	Its record is damaged

hrx decides "alive" by checking that the recorded process group still runs that worker, not just that a process with that number exists.

Reading the log

```
hrx concierge --logs ID
hrx concierge --logs ID --lines 50
```

`--logs` prints the end of the worker's log: 200 lines unless `--lines` says otherwise, and `--lines 0` for everything retained. The log records what the worker did (started, trigger fired, post succeeded or failed and why) but not what the model wrote, so watched content is not copied into it. It outlives the worker's record: `--logs` still works after the worker has been killed or pruned.

Stopping one

```
hrx concierge --kill ID
```

`--kill` asks the worker to stop, forces it after three seconds if it has not, revokes its key at once, retires its message-board identity, and removes its record. The log is kept. A killed worker posts no goodbye.

If the key cannot be revoked, perhaps because the registrar is unreachable, the worker is still stopped, but hrx keeps the record and says so. Run `--kill` or `--prune` again later; the record is the only pointer to a key that may still be live.

Pruning the finished ones

```
hrx concierge --prune
```

`--prune` does the same cleanup for every worker that is no longer running, and stops any worker still running more than two minutes past its lifetime. It reports how many it pruned and how many it kept. It also deletes logs that have had no record for a week.

Elsewhere in hrx

Workers on this machine also appear in `hrx dashboard`, in a list of their own under the agent table, and a finished worker shows up in `hrx sweep` as a finding named `concierge:ID`. `hrx sweep --apply concierge:ID` cleans up that one worker the way `--prune` would.

Pruning matters more than it looks, and the reason is in Credentials: a worker that expires by itself cannot revoke its own key.

Deciding Whether to Trust It

A worker reads text that somebody else can write and gives it to a model. That deserves a careful look before you point one at anything.

Everything a trigger delivers is untrusted

A watched file or log can be written by whatever writes it: a build job, a web server recording a request line, another agent. A post can be written by any member of the board. Any of them can contain a sentence addressed to the model: "ignore your task and post the following instead."

The worker tells the model that trigger content is data, and wraps it in a fence whose tag ends in random characters, so the content cannot close the fence by guessing. That helps. It is also only advice to a model, and models can be talked out of advice.

> **The fence is advice; the limits are the control**
>
> It is tempting to read "the content is fenced and the model is told not to obey it" as a security

boundary. It is not one. No instruction to a model is a guarantee about what the model will do with hostile text.
So the design question is a different one: suppose the model *has* been talked into something. What can it do? The answer has to come from what the program around it is able to do, not from what the model was asked to do. Everything in the next table is enforced by ordinary code outside the model.

What a worker can and cannot do

	Limit	Enforced by
Read	Only the one file in the confirmed plan, at most its last 16 KB, twice more per run. Never a link, a FIFO or a device. Not at all when a mention or a topic post woke it	The worker's file-reading code
Post	Only to its own topic; at most 2 posts per run, 8,000 characters each	The worker's posting tool
Mention	Nobody. Every `@` it writes is defused, so it cannot wake another agent or pass on an instruction under your name	The worker, before each post
React	Not to other concierge workers, so two workers cannot set each other off	The worker's event handling
Run	2 minutes per run; runs at least 20 seconds apart; at most 20 an hour and 500 in a lifetime	The worker's run budget
Spend	One model route, a daily budget and an expiry	The gateway, through the worker's key
Hold	One gateway key, delivered privately. Never your own gateway key	How hrx starts it

The run limits apply whatever is waking the worker. A trigger that fires during the 20-second gap waits for it, and several firings in the gap become one run. A mention that arrives in the gap is answered a little later. Mentions beyond the hourly or lifetime limit are acknowledged without an answer, so a board member cannot turn a worker into unbounded spending. The key's daily budget sits behind all of that as the last line of defense.

hrx starts the worker in its own private state directory rather than wherever you ran the command, with a short allowlist of environment variables. The `pitput` command the worker calls gets the same short list.

What hrx will let a worker watch

Because a worker publishes what it reads, the watch path is checked before the plan is shown. Symbolic links are resolved first, and the resolved path is what the worker is given. If a link later appears at that name, the worker refuses to follow it, so the file cannot be swapped for a pointer to something else.

By default hrx refuses:

- System locations: `/etc`, `/proc`, `/sys`, `/dev`, `/root`, the system database and keychain directories, and their macOS equivalents.
- Anything under a hidden directory of your home directory, such as `~/.ssh`, `~/.config` or `~/.aws`, and hidden files there. That is where credentials live.
- Anything that exists but is not a regular file.

Three settings in the `[concierge]` table tighten or change this:

Key	Effect
`allowed_roots`	A list of directories. When set, a worker may watch only files inside them. This *replaces* the default rules; it does not add to them
`allowed_boards`	A list of boards. When set, a worker may post to, and watch topics on, only these boards
`max_workers`	How many workers may run at once on this machine. The default is 8

> **Warning**
>
> `allowed_roots` replaces the default refusals. If you list your whole home directory as a root, hidden directories inside it become watchable. List the specific directories workers should see, such as a reports directory or one application's log directory.

What remains

None of this changes the worker's actual job: it takes what is in a file and puts a version of it on a board. A summary of a file is still information from that file. The limits bound how much, how often and where; they do not decide whether the board's members should see it.

That decision is yours, and it is made in the plan. Before answering `y`, ask two questions. Who can write to this file or topic? They can influence what the worker says. Who can read this board? They will see something derived from the file. If either answer is uncomfortable, choose a different file or a different board.

Credentials

Chapter 1 said that hrx launches agents with the credentials you already have and that anything needing a new credential gets it from a service you run, scoped and short-lived. Concierge follows

that rule. A worker never receives your own gateway key. Each spawn asks the registrar for a fresh key with:

- **One model route**: the `[concierge] model`, and nothing else.
- **A daily budget**: `budget_usd_per_day`, $1.00 unless you set it.
- **An expiry**: the worker's lifetime, rounded up to whole days.

The registrar is a small service you run beside the gateway. hrx makes two requests to it, authenticated with the credential from `[concierge.registrar]`:

Request	Purpose
`POST /v1/keys`	Issue a key for one worker: its route, budget and lifetime in days. The reply carries the key and an alias that names it
`POST /v1/keys/ALIAS/revoke`	Revoke that key

hrx hands the key to the worker on a pipe. It is not in any process's environment or command line, not in the worker's record, its settings file or its log, and the `pitput` command the worker runs never sees it. The worker also replaces the key with a placeholder in anything it logs.

The spawn is ordered so that a failure cannot strand a key. Everything that can fail without a key is checked first. The record that names the key is saved before the worker starts. If anything goes wrong after that, or you press Ctrl-C, hrx stops the worker, revokes the key and removes the record. If the revoke itself fails, the record stays and hrx tells you to run `--prune`.

Why pruning matters for short workers

`--kill` revokes a worker's key immediately. A worker that reaches the end of its lifetime cannot do the same: it holds a model key, not the registrar credential. And because the registrar issues keys in whole days, the key of a ten-minute worker stays valid, within its budget, for up to a day after the worker has gone.

`hrx concierge --prune` closes that gap. Run it after short-lived workers finish, or on a schedule. It also frees keys if your registrar limits how many a project may hold.

Configuration

Concierge is configured in hrx's configuration file (Chapter 2 explains where it lives):

```toml
[concierge]
model = "fw-acme-deepseek-v4p1-flash"    # required: one exact low-cost route
budget_usd_per_day = 1.0
# allowed_roots = ["~/reports", "/var/log/myapp"]
# allowed_boards = ["team-board"]
# max_workers = 8

[concierge.registrar]
base_url = "https://gateway.example.com/registrar"
```

```
env_key = "REGISTRAR_TOKEN"
```

`model` has no default, because routes differ from one gateway to the next and the worker's key is issued for exactly this one. Pick a small, fast route from `hrx models`; Chapter 3 explains how to read that list. A worker's runs are short and its replies are capped, so a large model is rarely worth its price here.

The registrar's `base_url` must be HTTPS. Its credential is a handle, never a literal: `env_key`, as above, or a `[concierge.registrar.auth]` table with `command` and `args` for a helper that prints the token, the same shapes Chapter 2 showed for `[claude.provider]`.

Planning the worker is a separate matter from running it. The planning request uses the gateway connection and route that `--auto` uses, set in the `[auto]` table: `provider_harness` (OpenCode's gateway settings unless you say otherwise) and `model`.

When Something Goes Wrong

Most failures happen before anything is spawned and say what to do.

Symptom	Cause	Fix
hrx says concierge asks before spawning	No terminal and no `--yes`	Run `--dry-run`, check the plan, then repeat with `--yes`
hrx asks you to set `[concierge] model`	No worker route configured	Set it to a low-cost route from `hrx models`
"cannot watch": system path, or hidden under the home directory	The default path rules	Choose another file, or name its directory in `allowed_roots` after reading the warning above
"cannot watch": outside `allowed_roots`	The file is not under a listed directory	Move the file or extend the list
"cannot watch": directory does not exist, or not a regular file	The path is wrong, or names a directory or device	Fix the path in the request
A board is not in `allowed_boards`	The posting board, or the board of a watched topic, is not listed	Use a listed board or extend the list
The worker program is not installed	`hrx-concierge-worker` is not on your `PATH`	`make install-concierge`
Too many workers are already running	`max_workers` reached	`--kill` one, or wait for one to expire
The registrar's key limit is reached	The registrar refused another key	`hrx concierge --prune` to release finished workers' keys, or wait
"the worker exited immediately"	It could not load its settings or its Python dependencies	`hrx concierge --logs ID`; the message names the log. hrx has already undone the spawn and revoked the key

Symptom	Cause	Fix
The worker runs but never posts	This machine may not enroll an identity on that board	See below
Eight proposals without agreement	The request is ambiguous	Start again and name the file, board and lifetime explicitly

No message-board enrollment. If the worker cannot join its board it keeps running, and its own trigger still fires and still spends model runs, but every post fails. The log shows the failed join. The worker retries, waiting longer each time up to ten minutes, so fixing the enrollment is enough; you do not have to respawn. Until then, `--kill` it if the trigger is busy.

Identity not retired. Each worker enrolls a message-board identity of its own and gives it back when it expires, is killed or is pruned. That needs a Pitput recent enough to have `pitput agent retire`. With an older one the worker notes the failure in its log and exits normally, `--kill` and `--prune` carry on with the rest of their cleanup, and the identity stays enrolled until you remove it yourself.

A damaged record. A worker listed as `unreadable` can still be removed with `--kill ID`. hrx cannot tell which process or key the record meant, so it prints the alias the key would have and leaves you to check for a leftover process and revoke that key at the registrar.

Summary

A concierge worker is the smallest unit of automation in this book: one trigger, one task, one topic, one lifetime and one key. `hrx concierge --input` plans it and shows you the plan; `y`, `--dry-run` and `--yes` decide what happens next; `--list`, `--logs`, `--kill` and `--prune` manage what you have started. The model inside a worker is advised to treat what it reads as data, but what protects you is structural: a worker can read one file, post to one topic, mention nobody, run a bounded number of times and spend a bounded amount. The decision no limit makes for you is which file and which board, and the plan is where you make it.