COMPLETE DEVELOPER PACKAGE

BUILD YOUR FIRST AI AGENTS

Five practical agents, one reusable architecture, and the engineering that keeps them honest

Mukesh Singh

First Edition

Manuscript 0.1.0 · Bundle 1.0.0-dev

Contents

Copyright and licence **2**

0.1 No warranty . . . 3
0.2 Output produced by models . . . 3
0.3 Support and updates . . . 3

How to use this book **4**

0.4 Three reading paths . . . 4
0.5 What you need . . . 4
0.6 How code is shown . . . 5
0.7 Callouts . . . 5
0.8 When you get unstuck . . . 6

1 Start Here: What an Agent Actually Is **7**

1.1 Learning objectives . . . 7
1.2 Why this matters . . . 7
1.3 What you will build . . . 8
1.4 Architecture . . . 10
1.5 Prerequisites . . . 10
1.6 Step-by-step build . . . 11
1.7 Code walkthrough . . . 12
1.8 Run it . . . 13
1.9 Tests . . . 13
1.10 Failure modes . . . 15
1.11 Security review . . . 16
1.12 Cost notes . . . 16
1.13 Exercises . . . 17
1.14 Recap . . . 17
1.15 Chapter checklist . . . 18

1.16 Further reading 18

2 Models, Tools, and the Execution Loop 19

2.1 Learning objectives 19
2.2 Why this matters 20
2.3 What you will build 20
2.4 Architecture 21
2.5 Prerequisites 21
2.6 Step-by-step build 23
2.7 Code walkthrough 23
2.7.1 The completion contract 23
2.7.2 A provider you can read 24
2.7.3 The loop 25
2.7.4 Who decides whether a tool runs 27
2.8 Run it 28
2.9 Tests 28
2.10 Failure modes 29
2.11 Security review 31
2.12 Cost notes 31
2.13 Exercises 32
2.14 Recap 32
2.15 Chapter checklist 33
2.16 Further reading 33

Free sample. This file contains the front matter and the first 2 of 11 chapters of *Build Your First AI Agents*. The full edition adds the remaining chapters, four appendices, the 25-prompt library, and the reference tables.

Copyright and licence

Build Your First AI Agents — Five Practical Agents, One Reusable Architecture

First Edition · Manuscript `0.1.0` · Code bundle `1.0.0-dev`

© 2026 Mukesh Singh. All rights reserved.

The prose and the code are licensed separately, because a reader needs different rights over a book than over software:

Part	Licence	What that lets you do
Book text and figures	read, quote with attribution, and teach from your own copy; no redistribution of the file	read, quote with attribution, teach from it
Code bundle (`agentkit/`, `projects/`, `examples/`)	personal and internal commercial use; modify and build on it; no resale or redistribution of the bundle	run it, modify it, use it inside your own product
Prompt library (`prompts/`)	the code licence	copy the prompts into your own agents
Third-party assets	see `LICENSES.md` in the code bundle	whatever each licence allows

The full terms are in `LICENSE-BOOK.md` and `LICENSE-CODE.md` in the code bundle, and those files govern if this summary and they ever disagree.

Reselling, redistributing, or republishing the bundle or the book file is not permitted. If you need a team licence, or you want to use the material in a paid course, contact the author through the storefront listing you bought this book from.

0.1 No warranty

The code and text are provided as they are, without warranty of any kind. The agents you build with them can read web pages, call paid APIs, and take actions in systems you connect them to. You are responsible for the credentials you supply, the data they touch, the limits you set, and the outputs you publish.

Nothing in this book is legal, security-certification, or professional advice. Where a technique has a known limitation, the chapter states it; where a number needs a date and a source, the chapter gives one. Neither is a substitute for your own review before you ship.

0.2 Output produced by models

Several chapters ask a language model to write text, and the projects treat that text as untrusted input. Model output can be wrong, can be plausible without being true, and can reproduce text it was trained on. Every project in this book is built so a human reads the result before it leaves the machine. Keep that property when you adapt the code.

0.3 Support and updates

Item	Where
Errata and corrections	the storefront listing for this edition; corrections ship in the next download
Code bundle	the download link on the same storefront listing
Questions	the author contact on that listing
Version stamp	printed on the title page of every export

Updates to the code bundle are free within the first edition. The version stamp on the title page tells you which manuscript and which bundle you are holding; quote both when you report a problem.

How to use this book

This book is a build log with the reasoning left in. Every chapter that shows code shows code that runs, and every number in a chapter has a source recorded in the repository: version, date, and how it was measured.

0.4 Three reading paths

You want	Read	Skip
To understand agents before building one	Chapters 1–3, then one project	The other four projects
One working agent this week	Chapter 2, then the project you need (5–9)	Chapters 3–4, until you need the detail
To put agents into production	Chapters 3, 4, 10, 11	Nothing

Chapters 1–4 build the shared runtime, `AgentKit`. The five projects in Chapters 5–9 each depend on it, and none depends on another project. You can start at any project chapter once Chapter 4 is behind you.

0.5 What you need

Requirement	Check it with
Python 3.12 or newer	`python3 --version`
`uv` (the package manager used throughout)	`uv --version`
A terminal, git, and about 2 GB of disk	`git --version`

Requirement	Check it with
An API key	**Not needed** for Chapters 1–4, or for any test in the bundle

The offline path is deliberate. The runtime, the tools, and the tests run against a scripted model, so a missing key or a dead network cannot stop you from finishing a chapter. Chapters 5 and 10 show the adapter that talks to a hosted model, with prices dated and sources recorded.

0.6 How code is shown

Listings are generated from the tested source tree at build time: a marker in the manuscript names a file and a line range, and the build copies those lines in. If the code changes, the next build breaks or updates with it. You will never read a listing that disagrees with the repository.

Listings are trimmed to the interesting 10–25 lines. The full file ships in the bundle, and the marker in the assembled Markdown names the file and range, so you can always find the rest.

Blocks labelled `console` are captured output from a real run, not typed by hand. Absolute paths and credentials are removed before anything is pasted in.

0.7 Callouts

Six kinds appear, each with a fixed meaning:

Why it matters — the consequence of a design decision. **Pitfall** — a specific mistake and the symptom it produces. **Security** — a threat and the code-level control that answers it. **Cost** — tokens, calls, money, or latency, labelled as estimate or measurement. **Try it** — a two-to-five minute change you can run as written. **Field note** — something observed while building, not theory.

0.8 When you get unstuck

1. Run `make verify` in the bundle. It checks that every listing resolves, every figure is registered, and no forbidden claim or secret slipped in.
2. Run `make test`. A failing test names the behaviour it protects, so the failure message is the diagnosis.
3. Read the chapter's *Failure modes* table. It lists the symptom, the cause, and the fix for the errors readers hit most.
4. If you are still stuck, send the chapter number, the command you ran, and the version stamp from the title page to the author contact on the storefront listing you bought this book from.

Exercises come in three tiers in every chapter: *modify* (change something that works), *break* (make a control fail and watch what happens), and *extend* (add a capability). Answers and sketches ship with the bundle.

1 Start Here: What an Agent Actually Is

1.1 Learning objectives

By the end of this chapter you can:

- separate a **model call**, a **tool call**, and an **agent loop**, and say which of the three a piece of code implements;
- state the loop back in one sentence, without notes;
- name five situations in which an agent is the wrong solution;
- describe what an agent loop costs in time and money before you run one;
- read a listing in this book and find the file it came from.

1.2 Why this matters

The failure this chapter prevents is cheap to create and expensive to operate. Somebody wraps a chat completion in a retry, the retry calls a search tool, the search returns nothing useful, and the program calls search again — because nothing in the code counts attempts. It looks like an agent. It has no ceiling. The first time it meets a question it cannot answer, it spends your token budget proving that.

Two ideas keep that from happening, and both are decided before the first prompt is written. First, an agent is a *loop with limits*, not a prompt with personality. Second, every action the loop can take is an action somebody registered in code and somebody permitted in configuration. Once those two are true, the interesting engineering moves to the boundaries — what the tool may touch, what the model may return, what happens when either fails — and that is where the rest of this book lives.

PITFALL

Symptom: the bill grows faster than the usage. A run that "sometimes takes a while" usually has no iteration cap, no per-call timeout, or both. The symptom appears in the invoice before it appears in the logs. Chapter 2 builds the four limits, and Chapter 4 makes them configuration instead of constants.

1.3 What you will build

One file, and it has no agent in it. `examples/00-first-call.py` makes a single completion against a scripted provider and prints four things: the text, the reason the model stopped, how many tool requests came back, and the token count. It is the baseline the rest of Part I adds to, and it runs offline with no key.

The artefact you should have at the end of the chapter is that output:

```
$ uv run python examples/00-first-call.py
text: An agent is a loop, not a prompt.
stop_reason: final
tool_calls: 0
tokens: 15
```

Four lines. By Chapter 4 the same program prints twenty, and each extra line is a decision recorded by the runtime: which tool ran, why a call was refused, how much of the budget remains.

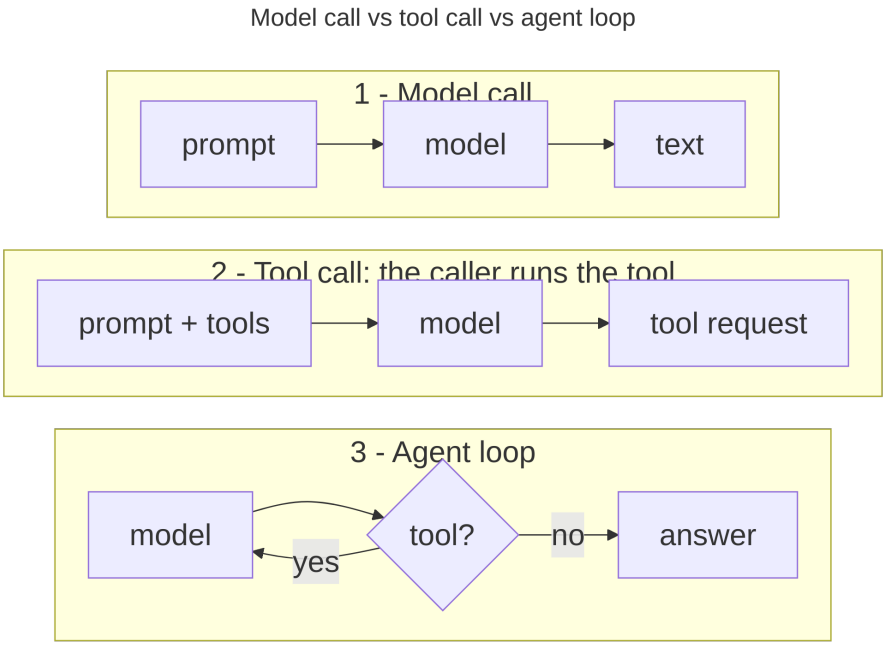


Figure 1.1: Three levels: a model call returns text; a tool call adds one external effect; a loop feeds results back until the model answers without a tool.

1.4 Architecture

Figure 1.1 (F01) puts the three things people call "an agent" side by side. The left panel is a model call: a prompt goes in, text comes out, nothing else happens, and nothing is remembered. The middle panel is a tool call: the same exchange, except the model's reply names a function and arguments, and the caller runs that code. The right panel is a loop: after the tool result goes back into the context, the model is asked again, and the cycle repeats until the model stops asking for tools or a limit stops it.

The distinction matters because each level adds a failure mode the previous one did not have. A model call can be wrong. A tool call can be wrong *and* have already written a file. A loop can be wrong, have written the file, and be on its fourth attempt to do it again. The controls you need grow with the panel you are standing in, which is why Figure 1.1 is the first figure in the book and Figure 2.1 is the second.

1.5 Prerequisites

Requirement	Why	Check
Python 3.12 or newer	`StrEnum`, `asyncio.timeout`, and modern typing are used throughout	`python3 --version`
`uv`	Creates the environment and runs every command in the book	`uv --version`
A terminal and git	The bundle is a repository, not a zip of snippets	`git --version`
No API key	Chapters 1–4 and every test run against a scripted provider	—

The exact interpreter, package manager, and typesetting versions used to verify this manuscript are recorded in the bundle (`plan.md` §7) and in the claims register as entry CLM-0003. If your output differs from a listing, compare your `python3 --version` first.

Install once, from the repository root:

```
$ uv sync
Resolved 17 packages in 1ms
Checked 15 packages in 0.28ms
```

`uv sync` creates `.venv` and installs the two workspace packages, `agentkit` and `research-agent`, in editable mode. Every later command runs inside that environment, so there is no second install step anywhere in the book.

1.6 Step-by-step build

Each step ends with something you can run.

1. **Run the shipped example.** From the repository root, `uv run python examples/00-first-call.py`. You should see the four lines shown above. Nothing here needs a key, a network, or a paid account.
2. **Change what the model says.** Edit the string inside `Step.final(...)` and run the same command again. Line 1 changes and the other three do not: the token counts come from the scripted step's default `Usage`, not from the text you typed. That is the first hint that a provider returns a *record*, not a string.
3. **Ask the provider what it was sent.** `MockProvider` keeps every request it received in `.requests`, and every tool menu in `.tool_menus`, so `tests/unit/test_mock_provider.py` asserts on both. Run that file and read the assertions: a request list of length one with an empty menu is the shape a single call produces, and Chapter 2 turns that empty menu into the point.
4. **Run the tests that guard the example.**

```
$ uv run pytest -q tests/unit/test_examples.py
.....                                                                    [100%]
5 passed in 0.87s
```

They exist so the file you are reading cannot drift away from the file that runs.

1.7 Code walkthrough

The listing is the entire program apart from the module docstring and the `if __name__` guard.

```python
from __future__ import annotations

import asyncio

from agentkit import Message, MockProvider, Step


async def main() -> None:
    model = MockProvider([Step.final("An agent is a loop, not a prompt.")])
    reply = await model.complete([Message.user("What is an agent?")])
    print(f"text: {reply.text}")
    print(f"stop_reason: {reply.stop_reason}")
    print(f"tool_calls: {len(reply.tool_calls)}")
```

Four decisions in seven lines, and each one is a decision Chapter 2 depends on:

`async def` on a protocol method, called with `await`. A hosted model call is network I/O that can take seconds. Making the contract asynchronous at the start means a provider that streams, retries, or cancels is not a rewrite later, and it means a limit implemented with `asyncio.wait_for` can actually interrupt a call.

A `Completion` record rather than a string. `Completion` carries the model name, the text, any tool requests, the reported token usage, and a `stop_reason`. The reason a program needs all five is that four of them change what you do next: text with no tool requests ends a loop, a `stop_reason` of `length` means the answer is truncated and must not be treated as complete, and usage is the only input to a cost report that is not a guess.

`stop_reason` is a closed set. It is one of `final`, `tool_calls`, `length`, `error`. A provider adapter maps a vendor's finish reason onto one of those four values or refuses the response. That mapping is where most provider-specific bugs live, and it is why the runtime can stay small: it never parses vendor vocabulary.

No loop. The example calls `complete` once and prints the result. If you wanted an agent, you would wrap this in `while`, append the reply to the messages, and ask again. That is three lines of code and the beginning of four new failure modes, which is the subject of the next chapter.

1.8 Run it

```
$ uv run python examples/00-first-call.py
text: An agent is a loop, not a prompt.
stop_reason: final
tool_calls: 0
tokens: 15
```

Read the output as a contract, not a demo. `text` is what a user would see. `stop_reason: final` says the model stopped because it was finished, not because it hit a length limit. `tool_calls: 0` says the reply requested no side effect, so nothing else can have happened. `tokens: 15` is the provider's report — ten input tokens for the question, five output tokens for the answer, as scripted by the mock. A hosted provider would report its own numbers here; the shape of the line would not change, which is the point of the record.

1.9 Tests

Five tests in `tests/unit/test_examples.py`. Two assert properties of the directory (an example exists, every example runs); three assert the output shape this chapter promises.

```python


def test_first_call_prints_the_scripted_answer():
    """Chapter 1 promises this shape; if it drifts, the chapter is wrong."""
    result = run_example(FIRST_CALL)
    first_line = result.stdout.splitlines()[0]
    assert first_line == "text: An agent is a loop, not a prompt."
```

What each failure would mean:

- **The example exits non-zero.** A reader following the chapter gets a traceback on the first command. The test reports the stderr, so the failure message is the diagnosis.
- **The first line differs.** The chapter and the code have drifted apart. Either the scripted sentence changed and the chapter was not rebuilt, or the chapter was edited and nobody re-ran the example. Both are caught at build time, before a copy is sold.
- **`tool_calls: 0` becomes non-zero.** Something started advertising or requesting tools in a chapter that promised none. That is a regression in the story as much as in the code: Chapter 1 has to be the chapter where nothing happens.

The tests run in a subprocess with the repository on `PYTHONPATH`, so they pass for a reader who cloned the bundle and installed only pytest. That detail is deliberate: a test suite that needs the book's own build environment to pass is not a test suite a reader can use. ## When not to build an agent

The most useful skill in this book is deciding *not* to build the loop. Six criteria, each of which has shown up in a real project review:

1. **The steps are known in advance.** If you can draw the flowchart before the first prompt — fetch, parse, validate, write — write that. A fixed pipeline is faster, cheaper, and testable with plain functions. Autonomy buys nothing when the path is already decided.
2. **The answer has to be exact.** Tax, payroll, invoice totals, dosage, contractual dates: any domain where "close" is "wrong" and a human must sign off on every number belongs in deterministic code with a model as an optional drafting aid, not in a loop.
3. **The decision rule is small and known.** Five labels with clear rules are a `match` statement, and a `match` statement does not hallucinate. Reach for a model when the rule is too large to write down, not when it is too small.
4. **The action is irreversible and no human can be in the loop.** Publishing, refunding, deleting, emailing a customer: if nobody can approve the action, the correct design is to remove the capability, not to add a stronger prompt.
5. **The latency budget is tighter than a model round-trip.** Users forgive a slow search page; they do not forgive an autocomplete that takes four seconds. A loop multiplies that latency by the number of turns.

6. **You cannot judge the output.** With no golden examples and no reviewer, you cannot tell a good run from a plausible one, which makes every improvement unmeasurable. Build the evaluation set first; Chapter 11 shows what it looks like.

An agent earns its cost when the path genuinely varies per input, when a tool is needed to reach facts the model was not trained on, and when a wrong action is either reversible or gated by a person. Those three conditions are what the next ten chapters build against.

1.10 Failure modes

Symptom	Cause	Fix
`ModuleNotFoundError: No module named 'agentkit'`	the command ran outside the repository root, or `uv sync` never ran	`cd` to the root, run `uv sync`, and use `uv run` for every command
Line 1 differs from the book	you edited the scripted sentence, or you are reading a different edition	compare the version stamp on the title page, then re-run the file unmodified
the mock script ran out	something asked for a second turn	each `complete()` consumes one `Step`; a second call needs a second step
`tokens: 0`	the provider reported no usage — zeros are a legal `Usage`	record cost as unknown for that call; never read zero as "free"
The output appears twice	the script ran under a wrapper that also prints, such as `pytest -s` or a notebook cell	run the module directly with `uv run python`

The exception named in that table is `ProviderScriptExhausted`, raised when a script has no turns left. Read its docstring: a mock that repeats its last turn forever would turn a test bug into a hang.

1.11 Security review

This chapter's design has exactly one security property, and it is worth naming precisely: **the program has no execution path.** There is no tool registry, no filesystem call, no network socket. The only sink is `print`. That is why the first chapter can be read on a locked-down machine, and why the threat model here is not about the model's behaviour but about yours: the moment you copy model text into a shell, a SQL client, or a file, you have added an execution path that no limit in this book can guard.

The second control is honesty about what a scripted provider proves. A mock that returns a sentence the test author typed cannot fail, so a green test proves the wiring, not the model. Chapter 11 turns that distinction into evaluation: stored inputs, recorded outputs, and a score that a prompt change has to beat.

The third is arithmetic: `tokens: 15` is a report, not an invoice. Cost enters the book in Chapter 10 as tokens × dated price, with the price source and the retrieval date recorded next to the number, so the estimate can be re-checked after a provider changes its rates.

1.12 Cost notes

One model call. The scripted provider reports ten input tokens and five output tokens, and those numbers are the mock's defaults, not a measurement of anything you would pay for. The example therefore costs nothing to run, on purpose: the first program in a book about bounded loops should not require a credit card.

The cost that matters appears one chapter later, and it is multiplicative. A run that takes N model turns makes N model calls, and each call carries the whole conversation so far, so input tokens grow faster than turns. The iteration cap you will write in Chapter 2 is therefore a spend ceiling as much as a liveness guard. Worked prices, with a date and a source, arrive in Chapter 10, with a dated entry in the claims register.

1.13 Exercises

Modify. Change the scripted sentence to an empty string and re-run. Predict the output before you look: the text line is empty, `stop_reason` is still `final`, and the token count does not change. Ask yourself which of the four lines a caller would wrongly trust.

Break. Give the provider two `Steps`, call `complete()` once, then call it again after the script is exhausted. You get `ProviderScriptExhausted`. Read the class docstring in `agentkit/src/agentkit/providers/mock.py`: a script that runs out is a test bug, and the error says so instead of looping.

Extend. Print `reply.model` and `reply.usage.input_tokens` as well. Then write the assertion you would add to `tests/unit/test_examples.py` to keep those two values stable, and decide whether the assertion protects the reader or locks in the mock.

1.14 Recap

- A model call returns text; a tool call adds an external effect; an agent loop adds control flow the model does not see.
- An agent is a loop **with limits**, and the limits are written in code before the prompt is written.
- A provider returns a record — text, tool requests, usage, stop reason — not a string, and the runtime's job starts with that record.
- The example in this chapter has no execution path, which is the only reason it is safe to run on any machine.
- Every listing in this book comes from a file that a test executes.

1.15 Chapter checklist

☐ `uv sync` completed and `uv run python examples/00-first-call.py` prints four lines.
☐ I can name the three levels in Figure 1.1 from memory.
☐ I can state the loop in one sentence.
☐ I can list five situations where an agent is the wrong solution.
☐ `uv run pytest -q tests/unit/test_examples.py` passes.
☐ I know where the code for every listing lives in the bundle.

1.16 Further reading

- Yao, Shunyu, et al. **ReAct: Synergizing Reasoning and Acting in Language Models.** arXiv:2210.03629, submitted 6 October 2022; version 3, 10 March 2023. https://arxiv.org/abs/2210.03629 — the paper that made the reason-act-observe loop a named pattern. Read Section 1 and the HotpotQA experiments for the loop, not for the prompting details.
- Anthropic. **Building effective AI agents.** 19 December 2024. https://www.anthropic.com/engineering/building-effective-agents — the workflow-versus-agent distinction, and the argument for starting with the smallest thing that works. Figure 1.1 is that distinction drawn three ways.
- OpenAI. **Function calling.** https://platform.openai.com/docs/guides/function-calling — the vendor reference for the tool-call shape Chapter 2 normalises into `ToolCall`. Read it for one thing in particular: how much provider-specific vocabulary an adapter has to map away.

2 Models, Tools, and the Execution Loop

2.1 Learning objectives

By the end of this chapter you can:

- implement a completion contract that a vendor adapter and a scripted provider both satisfy;
- write a loop that lets a model ask for one tool, run it, and answer;
- name the four ways that loop fails in production and the limit that stops each one;
- read a run's event log and say what happened, in order, without the code;
- explain why the scripted provider is a first-class implementation rather than a test shortcut.

2.2 Why this matters

Chapter 1's program has no failure mode worth the name: one call, one print, no state. The loop you write in this chapter has four, and all four are silent until they are expensive.

The model asks for a tool that returned nothing useful, so it asks again. The tool hangs on a socket that never closes. The conversation grows until the prompt no longer fits. The provider rate-limits you and a retry storm doubles the bill while you read the log. None of these throw an error a user would report, because the program is still running. That is the defining property of a runaway agent: it looks busy.

The answer in this chapter is not vigilance. It is four numbers, checked at four places in the loop, each of which raises a typed error instead of returning a half-finished answer.

WHY IT MATTERS

> **A limit that returns a partial result is not a limit.** If your loop breaks out and returns whatever text it has, every caller downstream treats a truncated run as a success. The runtime in this chapter raises `AgentRunError` instead, so a run either finishes or fails loudly. Chapter 4 turns the four numbers into configuration a deployment can change.

2.3 What you will build

`examples/offline_run.py`: a runtime, a scripted provider, one registered tool, and a read-only permission policy. The model decides to search, the runtime runs the search, the result goes back into the conversation, and the model writes the final answer. No network, no key, no cost.

The artefact is the run report at the bottom of the file's output — an answer, a summary line, and six events. You will use that shape for the rest of the book: every project prints events, every chapter reads them, and Chapter 10 ships them as structured logs.

2.4 Architecture

Two figures, one loop.

Figure 2.1 (F02) is the lifecycle. A run starts by checking its budget, then asks the model. If the reply contains no tool requests, the run is over and the result carries the answer plus the events that produced it. If it does contain tool requests, each one passes three gates — is the name registered, may this run use it, must a human approve it — before anything executes, and the result of the gate (a value, a refusal, or a timeout) goes back into the conversation as a message the model can act on.

Figure 2.2 (F03) is the seam that makes the loop testable. Project code talks to a protocol; two implementations satisfy it; one of them is a script. The runtime never imports a vendor SDK, which is why the same test file exercises the loop and the safety paths with no network.

The two figures answer different questions. Figure 2.1 says *what the loop does*; Figure 2.2 says *what the loop is allowed to depend on*. Keep them separate, and the runtime stays small enough to read in one sitting.

2.5 Prerequisites

Requirement	Why	Check
Chapter 1 completed	The provider record and `stop_reason` are assumed	`uv run python examples/00-first-call.py`
`uv sync` run at least once	`agentkit` is installed editable	`uv run python -c "import agentkit"`
About ten minutes	The loop is 40 lines; the tests take under a second	—
No API key	Everything in this chapter runs against the scripted provider	—

Run the import check before continuing: exit status zero means the package is

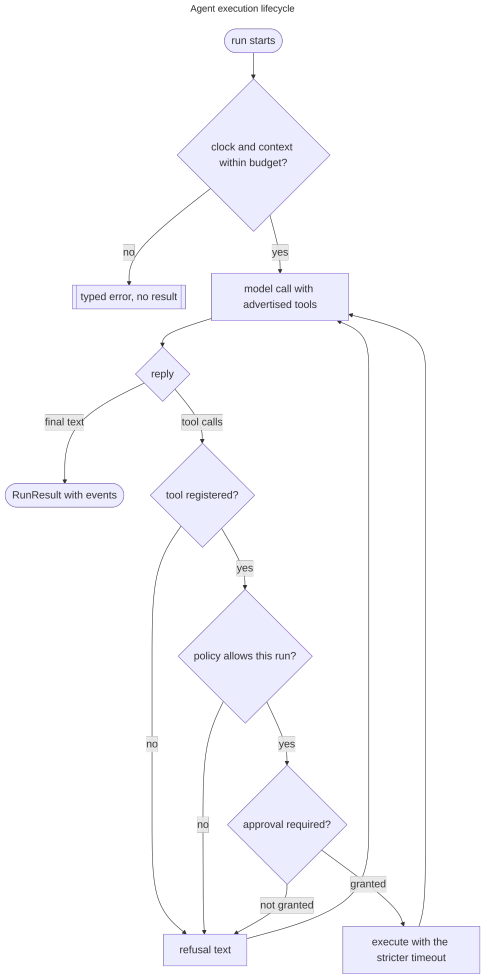


Figure 2.1: The lifecycle: budget checks, one model call, then either a result or a gate every requested tool must pass.

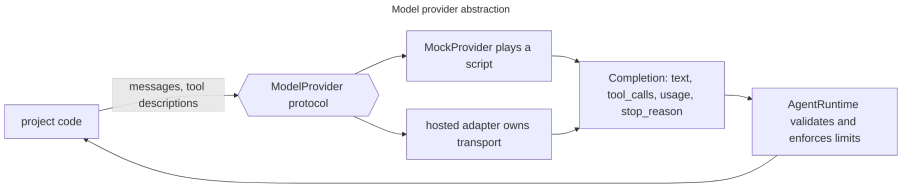


Figure 2.2: One protocol, two implementations. The runtime never imports a vendor SDK.

installed. Add `print(agentkit.__version__)` if you want the version number; this build is `0.1.0`.

2.6 Step-by-step build

1. **Run the finished loop.** `uv run python examples/offline_run.py` prints the answer, a one-line summary, and six events. Read the events before you read the code; they are the chapter in miniature.
2. **Find the two model calls in the log.** `[1]` is the turn that asked for the search, `[2]` is the turn that answered. The runtime did not decide either thing — the script did, and the runtime only enforced the bounds.
3. **Ask for a tool that does not exist.** In the example, change `Step.call("search", q="agents")` to `Step.call("delete_database")` and run it again. The run still finishes; the refusal is a message in the conversation, not a crash, and the log gains a `tool_unknown` event. Put the line back afterwards.
4. **Read the limits in the test suite.**

   ```
   $ uv run pytest -q tests/security/test_runtime_limits.py
   .................                                                        [100%]
   17 passed in 0.14s
   ```

   Those tests are the specification of this chapter; the prose below explains the code they pin down.

2.7 Code walkthrough

2.7.1 The completion contract

A provider has one job: take messages and a tool menu, return a `Completion`. Everything a project needs to know about a model reply is in that record.

```
@runtime_checkable
class ModelProvider(Protocol):
    """Completion interface. Async because real calls are I/O-bound.
```

```
    `tools` carries only *descriptions* the model may ask for; the runtime is
    what enforces which of them may actually run (spec §5.2 "do not trust the
    model to enforce permissions").
    """

    name: str
    model: str

    async def complete(
        self,
        messages: Sequence[Message],
        *,
        tools: Sequence[dict[str, Any]] = (),
        response_schema: type[BaseModel] | None = None,
    ) -> Completion:
```

Two things are worth pausing on. `tools` carries *descriptions*, not callables: the model can ask, and the runtime is what decides whether asking leads to running. And the method is `async` even though the mock never awaits anything, because a contract that is synchronous cannot be cancelled, and the wall-clock limit in this chapter depends on cancellation.

The module also ships `validate_structured()`, which parses a reply into a Pydantic model and raises rather than filling in a missing field. Chapter 4 uses it; the rule it enforces is stated there.

2.7.2 A provider you can read

`MockProvider` is not a stub. It is the implementation that lets the suite prove the loop without a network call, and it is scripted in exactly the vocabulary the loop speaks.

```
@dataclass(frozen=True)
class Step:
    """One scripted model turn.

    `tool_calls` holds provider-neutral payloads without ids; the provider
    assigns deterministic ids when it hands the step out.
    """

    text: str = ""
```

```
    tool_calls: tuple[dict[str, Any], ...] = ()
    error: ProviderError | None = None
    usage: Usage = field(default_factory=lambda: Usage(input_tokens=10,
output_tokens=5))

    @classmethod
    def final(cls, text: str) -> Step:
        """A plain answer. Ends the loop."""
        return cls(text=text)

    @classmethod
    def call(cls, tool: str, **arguments: Any) -> Step:
        """Ask for one tool call."""
        return cls(tool_calls=({"name": tool, "arguments": arguments},))
```

`Step.call()` builds one tool request with arguments; `Step.final()` ends a run; `Step.fail()` raises a provider error on that turn so the retry path can be tested. A script that runs dry raises `ProviderScriptExhausted` instead of repeating the last turn, because a test that asks for more turns than it wrote is a bug in the test, and a mock that loops forever would hide it.

2.7.3 The loop

The head of the loop is where every limit is checked and nothing has happened yet.

```
        for iteration in range(1, self.limits.iterations + 1):
            self._check_clock(deadline, iteration, tool_calls)
            self._check_context(conversation, iteration)
            completion = await self._ask(conversation, menu, deadline,
iteration, events)
            usage = _add_usage(usage, completion.usage)
            self._check_response(completion, iteration, usage)
            events.append(
```

`self._clock()` is injected, which is how the wall-clock test stops a run without sleeping. `_check_context` sums character counts over the conversation before the call, not after, because the point is to avoid paying for a prompt that is already too large. `_ask` owns retries and the per-call timeout; the loop body stays readable because of it.

The two exits are deliberately asymmetric.

```
            if not completion.requests_tools:
                events.append(
                    AgentEvent(EventKind.RUN_FINISHED, iteration, detail="final
answer")
                )
                return RunResult(
                    answer=completion.text,
                    messages=tuple(conversation),
                    events=tuple(events),
                    iterations=iteration,
                    tool_calls=tool_calls,
                    usage=usage,
                )

            # Recorded even with no text: an OpenAI-compatible server rejects a
            # `tool` message that does not follow the turn that requested it.
            conversation.append(
                Message.assistant_turn(completion.text, completion.tool_calls)
            )

            for call in completion.tool_calls:
                if tool_calls >= self.limits.tool_calls:
                    raise ToolCallLimitExceeded(
                        f"the model requested more than {self.limits.tool_calls}
tool call(s); "
                        f"stopped at iteration {iteration} before running
{call.name!r}"
                    )
```

The final-answer path returns a `RunResult` with the answer, the full conversation, the events, and the usage. The tool path checks the tool-call budget *before* incrementing it, appends the tool result to the conversation, and goes round again. If the loop runs off the end of `range()`, control reaches a `raise` — never a partial return. That asymmetry is the single most important design decision in the file: a caller can trust `RunResult` to mean "finished", and can trust an exception to mean "not finished".

The assistant turn is recorded even when it carried no words, and it keeps the tool calls it asked for. That is not bookkeeping for its own sake: a provider that speaks the OpenAI-compatible wire format rejects a `tool` message that does not follow the assistant turn which requested it, so a conversation that drops the re-

quest cannot be replayed. A log you cannot re-send is not an audit trail.

2.7.4 Who decides whether a tool runs

The model names a tool. The runtime decides whether that name leads to code, and whether this run may use it.

```
        try:
            tool = self.registry.get(call.name)
        except UnknownToolError:
            events.append(
                AgentEvent(
                    EventKind.TOOL_UNKNOWN, iteration, tool=call.name,
detail="not registered"
                )
            )
            known = ", ".join(item.name for item in self.registry) or "none"
            return f"error: unknown tool {call.name!r}; available tools: {known}"

        decision = self.policy.evaluate(tool)
        if not decision.allowed:
            events.append(
                AgentEvent(EventKind.TOOL_DENIED, iteration, tool=call.name,
detail=decision.reason)
            )
            return f"error: permission denied: {decision.reason}; the tool was
not run"

        if decision.requires_approval and not await self._approved(
            call, decision, iteration, events
```

Every refusal is returned as text, prefixed `error:`, and lands in the conversation as a tool result. The loop continues; the capability does not. That is what lets a model recover — try another tool, ask a question — instead of retrying into a wall until the iteration cap fires.

The runtime also advertises only the tools this run may use (`advertised_tools`, lines 133–143). Omitting a capability the policy would refuse is not the security control — the registry and policy are — it is the clearer contract, and it keeps the prompt from inviting a request that cannot succeed.

2.8 Run it

```
$ uv run python examples/offline_run.py
answer: Two notes found: agents are loops, and tools need limits.
tools: ('search',) | iterations: 2 | tool calls: 1 | refusals: 0
events:
  [1] model_request: attempt 1
  [1] model_response: stop_reason=tool_calls
  [1] tool_succeeded search: 1 item(s)
  [2] model_request: attempt 1
  [2] model_response: stop_reason=final
  [2] run_finished: final answer
```

Six events, and each one is a decision point rather than a log line somebody added for debugging. `model_request` with `attempt 1` says no retry happened. `model_response: stop_reason=tool_calls` says the reply contained a tool request, so the run had to continue. `tool_succeeded search: 1 item(s)` says the registry resolved the name, the policy allowed it, and the call returned. The second `model_response` says `final`, so the run ended by choice rather than by limit.

The summary line is the part to copy into your own projects. A run that reports its iterations, tool calls, and refusals can be triaged from a log without reproducing it.

2.9 Tests

`tests/security/test_runtime_limits.py` is the specification this chapter is written against. Seventeen tests, no network, and a `FakeClock` so the wall-clock limit is proven without waiting.

```
def test_the_iteration_limit_stops_a_model_that_never_stops():
    script = [Step.call("echo", text="again")] * 6
    runtime, provider = build(
        script, [echo_tool(Recorder())], limits=Limits(iterations=2)
    )

    with pytest.raises(IterationLimitExceeded, match="still working after 2
iteration"):
```

```
        asyncio.run(runtime.run([Message.user("loop forever")]))

    assert len(provider.requests) == 2, "the third model call never happened"
```

Read the assertions, not the setup. `pytest.raises(..., match="still working after 2 iteration")` asserts the *message* carries the budget and the count, because that message is what a reader of a failed run has to work with. Then `len(provider.requests) == 2` asserts the third model call never happened — the limit stopped the run before it spent another call, which is the property that matters when the provider bills per token.

What each group catches:

- **A model that never stops.** `iterations=2` against an endless script raises `IterationLimitExceeded`; the provider records exactly two requests.
- **A model that only calls tools.** `tool_calls=2` raises `ToolCallLimitExceeded`, and the recorder asserts exactly two executions ran.
- **A slow run.** A clock that advances six seconds per call against a fifteen-second budget raises `WallClockExceeded` before the second turn, so the failure is attributed to the budget rather than to the provider.
- **A hanging tool.** A tool that sleeps forever is cancelled at its timeout, the run continues with an `error:` result, and `TOOL_TIMEOUT` appears in the log. The next test asserts that a tool declaring a 60-second timeout cannot exceed the run's 1-second one.
- **A limit that returns a result.** The suite's result-object test catches the subtle bug: a caller reading `.answer` on a run that died.

The suite also asserts that every limit error is declared permanent (`retryable is False`). That single flag is what stops an orchestrator from retrying a run that will fail the same way on every attempt.

2.10 Failure modes

Symptom	Cause	Fix
the iteration limit fires on a run that was nearly finished	the model keeps requesting tools; the tool results are not useful enough for it to stop	read the event log for repeated `tool_failed` or `tool_denied`, fix the tool contract, and only then raise the cap
The run hangs and no limit fires	a tool or provider awaited something that never returns, and the declared timeout is long	check `tool_timeout_seconds`; the runtime uses the stricter of tool and run limits, so a lying tool cannot buy time
`WallClockExceeded` before any tool call	provider latency plus retry backoff consumed the budget	count `model_retry` events; lower `retries`, or raise the run budget with a recorded reason
`ContextTooLarge` at turn three	tool results are large and every turn re-sends them	cap what a tool returns, return identifiers rather than blobs, and lower the retrieved-document count
The same failure repeats every turn	a permanent provider error is being retried	check `ProviderError.retryable`; a rejected key or a bad request must not be retried, and the runtime already refuses to

2.11 Security review

Four controls in this chapter's design, each with a line of code behind it:

1. **The model cannot name its way into code.** `self.registry.get(call.name)` raises `UnknownToolError` for anything unregistered, and the loop returns a refusal. A prompt injection that invents a tool name reaches nothing.
2. **Permission is a property of the run, not of the prompt.** `PermissionPolicy` is deny-by-default: a run grants categories explicitly, and a tool whose category is not granted never executes. The model's text cannot widen the grant, because the grant is not in the context.
3. **Approval fails closed.** The default approver is `deny_all_approvals`, and the runtime treats anything other than a literal `True` — including an approver that raises — as a refusal. An agent library that approved by default would make `requires_approval=True` a decoration.
4. **Time cannot be renegotiated by the callee.** `min(tool.timeout_seconds, limits.tool_timeout_seconds)` means a third-party tool cannot extend its own ceiling.

Two things this chapter deliberately does *not* do, both taken up later: it does not redact secrets from logs (Chapter 4), and it does not record tool arguments in events. The second is a feature — arguments are untrusted model text that may contain anything a user typed — but it means the event log alone cannot reconstruct a full request. Chapter 10 adds the trace that can, with redaction.

2.12 Cost notes

Every iteration is one model call, and each call re-sends the whole conversation, so input tokens grow with every turn while output tokens stay roughly flat. Three ceilings bound the damage, all from `Limits`:

Ceiling	Default	What it bounds
`iterations`	6	model calls per run
`tool_calls`	12	executions per run
`retries`	2	attempts per model call (three attempts, then raise)

Worst case with those defaults is 18 model requests per run: six turns, each retried twice, if a provider fails transiently every time. That is the number to put in a capacity plan, not "usually two or three". Refusing a tool saves the tool's cost but not the model call that requested it, which is why a refusal loop still burns budget — and why the iteration cap matters more than the tool cap.

No prices appear here. Chapter 10 works through tokens × dated price with the provider's published rate and retrieval date recorded in the claims register.

2.13 Exercises

Modify. Build a runtime with `Limits(iterations=1)` around a two-step script that asks for a tool. You get `IterationLimitExceeded` with the budget in the message. Confirm from `provider.requests` that only one model call happened.

Break. Register a tool that sleeps for thirty seconds and give the run `tool_timeout_seconds=1`. Watch for the `tool_timeout` event and confirm the run continues and finishes. Then change the tool's own `timeout_seconds` to 60 and repeat: the outcome must not change, because the stricter limit wins.

Extend. Register a second tool and script one `Step` that requests both in a single turn. Check `tool_calls` in the summary, and check the order of `tool_succeeded` events. Then decide whether the loop should stop at the first failure or run the rest — and find the line that decides it.

2.14 Recap

- A provider returns a `Completion` record; the mock is a real implementation of the same contract, which is why the suite needs no network.
- The loop has exactly two exits: a finished `RunResult`, or a typed exception. Nothing in between is a result.
- The runtime, not the model, decides what runs: registry lookup, then policy, then approval, then a timeout the tool cannot raise.
- Four limits cover the four ways a loop runs away: iterations, tool calls, wall clock, context size — plus a token ceiling for providers that report usage.
- The event log is the run's audit trail: ordered, argument-free, and stable enough for a test to assert on.

2.15 Chapter checklist

☐ `uv run python examples/offline_run.py` prints six events and one tool.
☐ I can explain why the mock provider ships in the package.
☐ I can name the four gates between "model asked" and "code ran".
☐ The runtime-limits suite passes (`tests/security/test_runtime_limits.py`).
☐ I know which limit is raised when a model never stops, and where.
☐ I can name what the event log deliberately omits, and why.

2.16 Further reading

- Python documentation. **Coroutines and tasks** (asyncio). https://docs.python.org/3/library/asyncio-task.html — read the *Timeouts* and *Task cancellation* sections. The wall-clock limit is a deadline plus `asyncio.wait_for`, and knowing what cancellation cannot interrupt is the difference between a limit that works and one that raises after the call was already billed.
- Yao, Shunyu, et al. **ReAct: Synergizing Reasoning and Acting in Language Models.** arXiv:2210.03629, 2022; version 3, 2023. https://arxiv.org/abs/2210.03629 — this chapter implements a strict subset of ReAct: act, observe, reason. The paper shows what the extra structure buys; the limits here are what it costs to run safely.
- Anthropic. **Building effective AI agents.** 19 December 2024. https://www.anthropic.com/engineering/building-effective-agents — read Appendix 2 on writing tool descriptions. An ambiguous tool schema produces exactly the retry loop the iteration cap exists to stop.