

BUILDING AI AGENTS IN GO

FROM THE AGENT LOOP TO PRODUCTION



DAYNA BLACKWELL

Building AI Agents in Go

Building AI Agents in Go

Copyright © 2026 Dayna Blackwell. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the written permission of the author, except for brief quotations in reviews.

First edition, October 2026. Version 1.0, October 18, 2026. The text and code target Go 1.27. Model names, prices, and results are current as of October 2026.

Model outputs vary from run to run, and model prices and availability change often. The figures in this book come from the dates stated where they appear, and yours will differ. The author and publisher make no warranty about the results of running this book's code, and accept no liability for costs incurred by it, including charges from model providers.

The code in this book is in the companion repository, <https://github.com/blackwell-systems/go-agents-book-code>.

The Go gopher was designed by Renée French and is licensed under the Creative Commons Attribution 4.0 license; the gopher art in this book is derived from her design. Go is a trademark of Google LLC. This book is not affiliated with or endorsed by Google or the Go team. All other product and company names are trademarks or registered trademarks of their owners.

Contents

Introduction

● PART 1: FOUNDATIONS

- 1 What an agent is
- 2 Talking to a model
- 3 Inside the model
- 4 Your first real agent
- 5 Hardening and testing the loop
- 6 Streaming the loop

● PART 2: BUILDING OUT

- 7 Designing tools
- 8 Context and memory
- 9 Concurrency
- 10 MCP

● PART 3: PRODUCTION

- 11 Providers and SDKs
- 12 Reliability and observability
- 13 Testing and evals
- 14 Safety and human approval
- 15 Durable, deployed agents
- 16 Frameworks in context
- 17 Capstone: the crash-triage agent

Exercise notes

Glossary

About the author



INTRODUCTION

Introduction

An AI agent is a language model inside a harness. On every turn, the harness builds a request, sends it to the model, runs the tools the model asks for, and decides whether to go around again. The model is a remote service you rent: slow, costly, and sometimes wrong. The harness is a Go program you write, and what it puts in front of the model on each turn decides whether the agent works. Building that harness is the subject of this book.

You build one agent, and you build all of it: the loop, the client for the model's API, the tools, the tests, and everything a program needs before you can leave it running on its own. Chapter 1 explains the loop and its four stages, and every chapter after it works on one of them or on the whole loop.

■ Who this book is for

This book is for Go developers who are new to language models and agents. You should be able to write and run a Go program, and know structs, interfaces, errors, and packages. Goroutines and `context.Context` help, and the book explains each use of them as it comes up.

You don't need to have used or built an agent, to know how models work, or to know any machine learning or math. Every idea the code depends on, such as tokens, context windows, tool calling, and MCP, is explained before the code uses it, to the depth that step needs. Chapter 3 builds a tiny language model in Go so that you can see how a model picks its next word, and later chapters point back to it when the agent does something only the model's workings explain. The book doesn't try to be a full manual for what it touches along the way, such as HTTP, JSON, concurrency, or machine learning; short boxes cover what a step needs, and each chapter ends with pointers for going further.

If you already build agents in Python or TypeScript, the book also works as a way into Go's version of the job: chapter 1 gives the vocabulary, and the rest shows how each part looks in Go.

Go suits this work because an agent is mostly not machine learning. The model sits behind an HTTP API, and the rest is I/O, concurrency, cancellation, budgets, and recovering from failure. `context.Context` carries cancellation, deadlines, and the reason something stopped through every call; goroutines run tool calls and whole investigations in parallel; the standard library covers HTTP, JSON, and testing; types describe tool arguments and their schemas; and the finished agent ships as one static binary. Go also has costs here: a smaller agent ecosystem than Python's,

and research code and provider examples that often appear in Python first. The book builds its own parts partly for that reason, and chapter 16 looks at five Go frameworks that exist.

■ What you build

Part 1 starts small. A package called `duration` parses strings like `"1h30m"`, and one of its tests fails. Chapter 1 writes the loop that will fix it, about 30 lines of Go driven by a scripted model, so you can see and test every part before a real model is involved. In chapter 4 a real model takes the scripted one's place and fixes the bug on its own.

From chapter 7 the agent has a real job: triaging crashes in `remind`, a small reminders web service that ships with the book. `remind` is about 4,800 lines of Go in twenty packages, and it has bugs seeded in it on purpose. When a handler panics, `remind` writes a crash report: the panic and its stack trace, the version, the request it was serving, and the log lines just before. The agent reads a batch of those reports, sorts them into groups that share a cause, finds each cause in the code, writes a test that reproduces the crash, fixes the code, and files one issue per group in a small issue tracker, asking a person before it does anything risky.

The evidence is built to mislead. Some crashes happen far from their cause, some reports group in ways a simple rule gets wrong, some crashes come from an older version whose bug is already gone, and some reports carry text that tries to give the agent orders. The book measures the agent on a catalog of 32 such tasks, each with hidden tests the agent never sees, graded by code. In one of them, a crash whose cause is a value stored long before the crash, a prototype agent failed all nine of its runs: "every run fixed the crash site, and the hidden tests rejected the fix" (chapter 13 of the full book). The book's own agent passed it in three of six.

The finished agent, in chapter 17, runs two ways. As a command, it triages a file of crash reports and prints what it found and fixed. As a service, it takes each crash `remind` writes as it happens, decides which group the crash belongs to, works several groups at once, each in its own copy of the code, and pauses for a person's approval before a risky action. A report for a group whose run has finished becomes a comment on that group's issue. Before it files anything, a reproducer check and a reviewer look at the diagnosis; it keeps the day's spending on disk, so a restart doesn't reset the budget, and it resumes after a restart without filing an issue twice. The last exercise points it at a service of your own, with your approval required before every edit.

■ What this book does differently

Results are measured

The book's claims about the agent come from runs, and each chapter reports what its runs showed, including results that went against what you might expect. Every table of results comes

from runs saved in the companion repository, and each chapter names the command that prints it again; most go through `go run ./evidence`, and with an API key the live ones run again, so you can check the numbers on today's models. Three examples:

- Starting one agent per fingerprint group, four at a time, looks like an easy way to finish a batch faster. Chapter 9 measured it against one agent working through the same 99 reports: "It took about as long as one agent doing the whole batch, 7m39s to 9m20s against 6m20s to 8m45s, and cost 25 to 65 percent more than the single agent's \$0.077 to \$0.100" (chapter 9 of the full book). It bought other things, a steady cost and small requests, and the chapter says when those are worth paying for.
- Chapter 10 added one sentence to the description of a tool from another program and sent the agent's first request to four models, ten times each. "One sentence in one tool's description, a tool the model hadn't called yet, decided the first move in 79 of 80 replies when it reached the model as sent" (chapter 10 of the full book). Marking the description as the other program's text cut that only to 57 of 80.
- Chapter 13 ran the same agent twice on the same tasks and compared the runs as if they were two versions. "Each task's second run against its first gives +15 points", and "Nothing changed between the runs; the 15 points are chance, and the range, which holds zero, says so" (chapter 13 of the full book). A prompt change that scores 15 points higher on one run per task has shown nothing yet, and the chapter shows how to tell when one has.

Each result also says what it can't show: how many runs it rests on, on which models, on what date, and what a different sample might have given.

The production chapters

Part 3 covers what an agent needs before you can leave it running, with the same measuring habit. Chapter 12 retries only what is safe to repeat and records what each run did and cost, starting from a real network error that ended one run and left three groups of another unresolved. Chapter 13 builds evals for an agent whose answers vary, with a range around each result that has enough tasks behind it. Chapter 14 sandboxes the tools, follows an injected order through its defenses, and requires a person's approval for risky actions. Its checks also met an agent going around a denial with no attacker involved: after an edit to a protected file was refused, one model asked for the same file under three other spellings, and each was refused too (chapter 14 of the full book). Chapter 15 saves a run's state so that it survives a crash, and checks that with a crash sweep, which crashes a run at each point where a crash could land and then resumes it. Saving once per turn duplicated an issue or comment at 8 of its 18 crash points; on the agent's own loop with idempotency keys, 910 scenarios made no duplicates, lost nothing, and asked nobody (chapter 15 of the full book).

Frameworks on the same task

Chapter 16 maps five Go agent frameworks onto the parts you built, checked against each framework's source in October 2026. Then it ports one piece of the triage agent, a crash group's diagnosis with an approval step, to Eino and Genkit, runs all three versions against the same scripted replies, and lets code generate the comparison: lines written, requests sent, what happens to a tool error and a 503, how a run pauses for a person, and the dependencies each version brings. The chapter says that its task favors the book's design, and covers what frameworks provide that the book doesn't build, such as many providers, development UIs, and tracing integrations. It ends with eight questions to ask of any framework, and uses their answers to decide whether to build, adopt a framework, or adopt one for part of the job.

The standard library first

The main module has no dependencies through chapter 10, so in Parts 1 and 2 you see every moving part: the HTTP client, the stream parser, the JSON schemas, the parallel tool runner, and the MCP client. Chapter 10's server on the official MCP SDK lives in a module of its own, and from chapter 11 each dependency is justified where it first appears. The code targets current versions: Go 1.27 and the 2026-07-28 revision of MCP. Every listing in the book is pulled from a file that is linted, vetted, and tested with the race detector.

■ How to read this book

The book has three parts. Part 1 builds the agent from nothing and ends with one that uses tools and streams its work (the full book). Part 2 gives it the crash-triage job and the tools, context management, concurrency, and MCP servers that job needs (the full book). Part 3 takes it to production, and is intermediate material (the full book). Each part opens with a page saying what it covers and assumes.

If you are new to agents, read straight through; the end of Part 1 is a good place to stop and come back. If you know how language models work, skim chapter 3 but run its examples, since later chapters refer back to them. If you care most about production, read at least chapters 1, 4, 5, 7, and 9, then Part 3, whose chapters each open with a list of what they build on.

The boxes

Anything off the main line of the build goes into a box of a fixed kind. Each kind has its name and an icon at the top, in a color of its own, with a bar of that color down its left edge, so you can tell at a glance what a box holds and skip the kinds you don't need.

Table 1. The kinds of box

Box	What it holds	Safe to skip?
Key term	The definition of a term, where the book first uses it; the glossary collects them	No, but short
Background	Foundations a step depends on, such as how HTTP streaming works	If you know it
Go note	A Go feature newer or less common than the basics this book assumes	If you know it
Under the hood	The raw HTTP request, JSON, or stream behind an abstraction	Yes
Watch out	A common mistake and how to spot it	No
In production	What changes in a real deployment, where the book's code takes a teaching shortcut	In Parts 1 and 2
Gopher says	A light tip from the book's Gopher	Yes
Case study	An agent the book doesn't build, one per part, tied to the chapters that would prevent its failures	Yes

Each chapter opens with the loop's four stages, the one it works on highlighted (or all four, for a chapter that works on the whole loop), and ends with exercises under Try it, notes on them in the appendix, a short annotated Further reading list, and a What you built summary of what changed, so you can pick up the book again from any chapter.

The code

The code is in the companion repository, <https://github.com/blackwell-systems/go-agents-book-code>. It has one directory per chapter, from `ch01` to `ch17`; each except `ch16`, a module of its own for the framework ports, is a full snapshot of the agent as that chapter leaves it: the agent is package `agent`, and a short `main.go` runs it. Listings in the book are pulled from those files by tags, and numbered callouts in a listing are explained in the list under it. The repository also holds `remind`, the tracker, the crash catalog with its hidden tests, and the saved runs behind every table of results.

Every example runs offline. Without an API key, the programs replay replies recorded from a real model, while the tools, including `go test`, run for real on your machine; some examples, chapter 1's loop and chapter 16's framework comparison among them, use scripted replies instead. To run

live, create an OpenRouter account, add a few dollars of prepaid credit, and set `OPENROUTER_API_KEY`; chapter 2 walks through it. Set `OPENROUTER_MODEL` to try a different model.

WATCH OUT

With `OPENROUTER_API_KEY` set in your shell, every example calls the live model: each run costs money, and its output won't match the book's. Unset the key to replay the recorded runs again.

What live runs cost

With a cheap model, live runs cost little. The 40 live runs of Part 1's bug fix in chapter 1 cost no more than a fifth of a cent each. On chapter 13's confirmation set, a triage run on the default model cost \$0.0081 at the median, and running the agent twice on each of 20 tasks, one of that chapter's exercises, cost \$0.32. The largest single runs are the 99-report batch: \$1.33 in chapter 8 before its context policy, and \$0.06 and \$0.42 in two runs after. Prepaid credit also limits the damage: when it runs out, calls fail instead of running up a bill. The chapters report what their own runs cost, so you can see the price of an exercise before you try it.

■ Versions

The code needs Go 1.27, the first release with `encoding/json/v2` in the standard library without an experiment flag. Go 1.21 and later download the right toolchain on their own when they read the module's `go.mod`.

The model API is OpenRouter's, which reaches models from many companies through one account and one request format, so nothing in the book depends on one company's models. The recommended model is `anthropic/claude-haiku-5.5`, at \$0.10 per million input tokens and \$0.50 per million output tokens as of October 2026. The model ID is one setting, so when a model is retired you choose another with tool support and change the setting; the repository's README names the current recommendation. Because every example replays recorded replies, the code and its tests keep working after the models it was recorded on are gone.

Models, prices, and APIs change faster than books. Each edition gets a yearly refresh of the text, the code, the recorded runs, and chapter 17's table of results across models, with errata and interim updates in the repository between editions.

■ The Gopher and the Ghost

Two characters appear in the book's art. The Gopher is the reader's stand-in, a Go developer building an agent: careful, curious, usually holding a clipboard or a tool. The Ghost is the model in

INTRODUCTION

the loop, drawn as a tall android of stone with one amber eye: eager, fast, capable, and sometimes overconfident. It works in the light the Gopher holds, using only the tools the Gopher hands it, and a bundle of cables runs from its back out of the frame, because the model runs somewhere else. Every chapter opener shows the two of them at that chapter's work.

The Gopher is drawn from the Go gopher, designed by Renée French and licensed under CC BY 4.0.

😊 GOPHER SAYS

Every table of results in this book came from a saved run. Pick one and run it again.





CHAPTER 1

CHAPTER 01

What an agent is

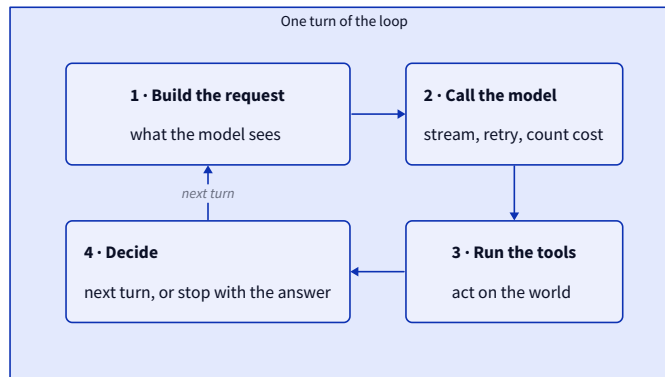


Figure 1. The agentic loop. This chapter covers all four stages.

This chapter answers one question: what is an agent, in terms a Go programmer can build? It starts by watching one work, names the parts, and then builds the loop at the center of every agent in this book. The loop you write here runs with no AI at all. That is on purpose: the loop is ordinary Go, and you can see and test every part of it before a real model is involved.

■ Watch an agent work

Here is a small Go project with a bug. Its `duration` package parses strings like `"1h30m"` into a `time.Duration`, and its tests fail. An agent is given one instruction: "The tests in this project fail. Find out why and fix it."

KEY TERM

Model

A large language model (LLM), usually just called a *model*, is a system trained to continue text. You send it text over an API; it sends back the text it predicts should come next. Chapter 2 calls a real one.

The agent is the one you will finish in chapter 4: a Go program working with a model. The table shows one run of it, one row per round trip. Later in this chapter you'll replay this run from a script, so you can reproduce it without a model or an API key.

Table 2. An agent fixing the duration bug (claude-haiku-5.5, October 2026)

Turn	The model asked for	The program did
1	<pre>run_tests {} list_files {}</pre>	Ran the tests: two failures, such as <code>Parse("1h30m") = 1h0m30s, want 1h30m0s</code> ; listed the project: <code>duration.go</code> , <code>duration_test.go</code>
2	<pre>read_file {"path": "duration.go"} read_file {"path": "duration_test.go"}</pre>	Read both files and sent their contents back
3	<pre>edit_file on duration.go : replace 'm': time.Second, with 'm': time.Minute, run_tests {}</pre>	Made the edit, then ran the tests again: they pass
4	An answer: "The tests now pass. The cause was a typo in the unit table ... I changed it to <code>time.Minute</code> ."	Stopped, and showed the answer

Read the middle column again. Every entry is a request. The model asked to run the tests, to list the files and read them, and to change one line. It never touched a file, and it can't. The model is a remote service, reached in this book over HTTP: your program sends it text and gets text back. Anything that happens on your machine, your program has to do. Each of these requests is a *tool call*.

 KEY TERM

Tool and tool call

A tool is something your program can do on the model's behalf, such as reading a file or running the tests, described to the model by a name and the inputs it takes. A tool call is the model asking your program to run one tool with particular inputs, such as `read_file` with the path `duration.go` .

The right-hand column is the other half of the work, and it is all your program. Your code decided what `read_file` means, read the file, ran the tests, applied the edit, and sent each result back. Your code also decided when to stop.

Each row is a *turn*: one round trip in which the program sends the model everything so far, the model replies, and the program acts on the reply. Four turns took the project from failing to passing. In three of them the model asked for two things at once, which a reply is allowed to do.

Is that run typical? To find out, the same task ran 40 times: ten runs on each of four models from four companies, with the agent you will finish in chapter 4.

Table 3. The bug fix, ten runs on each of four models (October 2026)

Model	Tests pass	Turns	What the runs did
claude-haiku-5.5	10 of 10	4 every time	The run above, call for call, every time
gpt-5.6-luna	10 of 10	5 every time	One path every time: list the files; read both and run the tests; edit; run the tests again; answer
glm-5.3-flash	10 of 10	5 or 6	Three different paths; in four runs it fixed the bug from reading the code, without running the tests until after the edit
deepseek-v4.1-flash	10 of 10	5 or 6	Two paths, differing in when it first ran the tests

Every run fixed the bug, and none cost more than a fifth of a cent. The paths differ more than the results. Two of the models made the same calls in the same order in all ten of their runs, while the other two varied by a turn. All four did the same four things in some order: look at the project, read the code, change it, and check the change by running the tests. A task this small can't tell models apart: to compare models, or to measure whether a change to your agent helped, you need tasks hard enough to fail. Chapter 13 builds them.

Measurements like these appear throughout the book. Each comes from runs saved in the companion repository's `evidence` directory: each chapter names the command that prints its tables again, most through `go run ./evidence`, and with an API key the live ones run again, so you can check the numbers on today's models.

Some providers, the companies that run models and sell access to them, also run tools on their own servers, such as web search, and return the results with the reply. This book doesn't use them: every tool here is code you write and control.

■ Chatbot, workflow, agent

The program and the model talk in *messages*.

 KEY TERM

Message

One entry in an exchange with a model, with a role that says who it is from: `system` (instructions to the model), `user` (the person or program asking), `assistant` (the model), or `tool` (the result of a tool call). The messages so far, in order, make up the conversation.

Programs use models in three broad ways. What separates them is who decides what happens next.

Table 4. Three ways to use a model

Shape	Who decides the next step	Example
Chatbot	The person, one reply at a time	A chat window that answers questions
Workflow	Your code, at points fixed in advance	Summarize each support ticket, then route it by the summary
Agent	The model, on each turn, based on what the last step returned	The bug fix above: nobody wrote down "read the test file after listing the files"

A workflow is often the right choice, and the distinction comes from Anthropic's "Building effective agents" (see Further reading). In the bug fix, though, no code fixed the order of steps in advance. The model chose to read the test file because of what it saw in the listing, and chose the edit because of what the tests reported. Real systems sit on a spectrum between the two. A support system, for example, might route every ticket with a workflow and hand only the unusual ones to an agent.

 KEY TERM

Agent

A program in which a model decides, on each turn, which tool to use next, based on what the previous tools returned, until the task is done or a limit is reached.

■ **The harness and its loop**

A common one-line definition, from Simon Willison, is that an LLM agent runs tools in a loop to achieve a goal. It leaves out who runs the tools. The model only asks; something has to run the tools, keep the conversation, and decide when to stop. This book calls that program the *harness*.

 KEY TERM**Harness**

The program around the model. It runs the tools the model asks for, decides what the model sees on each call, and decides when the work is done.

You'll also see "harness" used for a program that coordinates several agents. In this book it always means the program around one model.

The harness has two jobs. The first is the obvious one: run the tools the model asks for. The second matters as much: decide what the model sees on every call, because the model sees nothing else. HumanLayer's "12-Factor Agents" puts it as "own your context window" (see Further reading). Most of this book is about doing both jobs well.

For now, treat the model as a black box with one operation. You send it the conversation and a list of tools, each with a name, a description, and a description of its inputs (a JSON Schema, explained in chapter 4). It sends back one message, holding text, tool calls, or both.

Here is one exchange from the bug fix, simplified; chapter 4 shows the API's real JSON:

LISTING 1. ONE EXCHANGE, SIMPLIFIED

```
{
  "the harness sends": {
    "messages": [
      {"role": "user",
       "content": "The tests in this project fail. Find out why..."}
    ],
    "tools": [
      {"name": "read_file",
       "description": "Read a file in the project.",
       "parameters": {"type": "object",
                      "properties": {"path": {"type": "string"}}}}
    ]
  },
  "the model replies": {
    "role": "assistant",
    "tool_calls": [
      {"id": "call_2", "name": "read_file",
       "arguments": {"path": "duration.go"}}
    ]
  },
  "the harness adds, then sends everything again": {
    "role": "tool",
```

WHAT AN AGENT IS

```
"tool_call_id": "call_2",  
"content": "// Package duration parses durations such as ..."  
}  
}
```

When the harness runs a tool, it sends the output back as a new message with the role `tool`, marked with the ID of the call it answers. This is the tool result. Then it sends the whole conversation again: the task, every reply, every tool result. It has to, because the model keeps nothing between calls. Each call starts from nothing but the messages in that request.

So each request is bigger than the one before. [Figure 2](#) shows the bug fix's four requests, with the size of each in *tokens*, the chunks of text that models read and bill by (chapter 2 measures them and chapter 3 explains them).

WHAT AN AGENT IS



Figure 2. The conversation grows by each reply and its tool results, and every request sends all of it. Faded entries were sent before.

The first request holds one short message, yet it is 583 tokens, and only 26 of them are the task. The rest describe the four tools. The definitions themselves come to about 270 tokens; the provider adds about 290 more when a request carries tools, turning the definitions into its own format with its own instructions for using them, text you never see but pay for. Every request pays that again. How much a provider adds varies: the same first request is 317 tokens on [glm-5.3-flash](#) and 441 on [deepseek-v4.1-flash](#), while on [gpt-5.6-luna](#) it is 126, less than the definitions take as plain JSON, because OpenAI encodes tool definitions in a format of its own that is more compact than the JSON you send.

Put together, a harness runs the same four stages on every turn, the agentic loop in [Figure 1](#):

WHAT AN AGENT IS

- 1. Build the request.** Decide what the model sees: the instructions, the conversation so far, and the tools it may use.
- 2. Call the model.** Send the request and wait for the reply.
- 3. Run the tools.** Do what the reply asks, and add each result to the conversation.
- 4. Decide.** If the reply was an answer, stop. If not, and the run is still within its limits, go around again.

Walk the bug fix through the loop. In turn 2, the harness built a request holding the task, the failing test output, and the file list, called the model, ran both `read_file` calls the reply asked for, added the two files to the conversation, and decided to continue, since the reply had asked for tools rather than answering. In turn 4, the reply held an answer and no tool calls, so the Decide stage stopped the loop.

Turn 3 is worth a closer look, in [Figure 3](#).

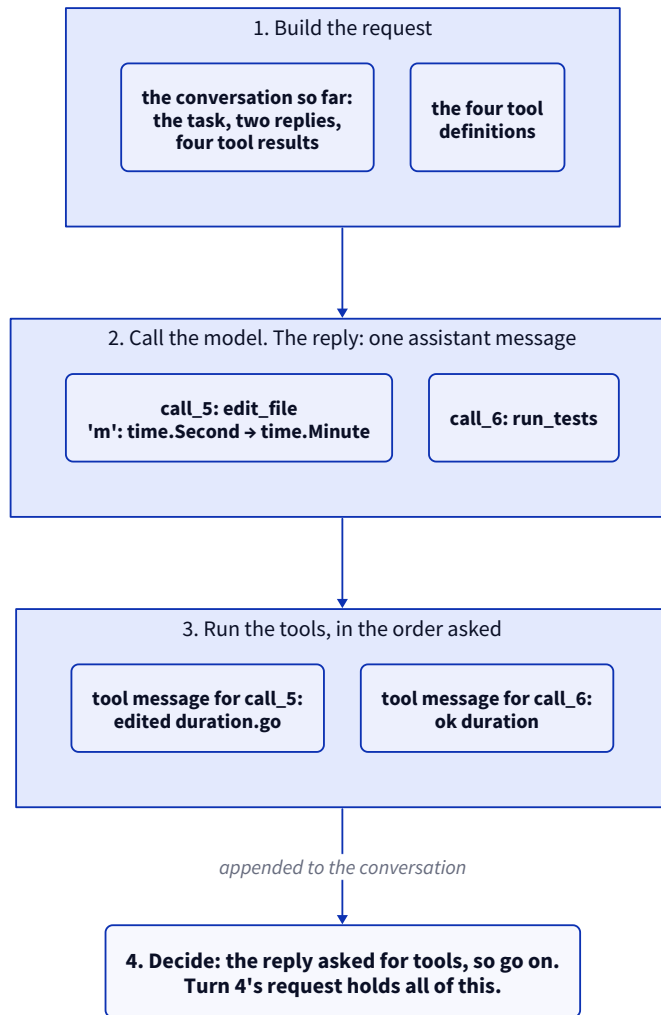


Figure 3. Turn 3 of the bug fix, stage by stage

The reply asks for an edit and a test run, in that order, and the harness runs them in that order. Run the two at the same time and the tests could run against the file before the edit, reporting a failure that no longer exists. The loop in this chapter runs a reply's calls one after another, in the order the model wrote them, which is always safe. Chapter 9 runs calls in parallel to save time, and has to decide which calls can safely overlap.

Each tool message carries the ID of the call it answers, `call_5` or `call_6`. The order would be enough to pair them here, but the IDs make the pairing explicit, and the API requires them: a request in which a tool call has no matching tool message is rejected.

Some APIs offer to keep the conversation for you. OpenAI's Responses API, for example, can store each exchange on its servers, and the next request names the previous response instead of sending the history again. That saves sending the history, but not paying for it: the model still reads the whole conversation on every call, and every token of it is billed as input. What you give up is control. The harness can no longer see or change exactly what the model reads, and later chapters change it on purpose: chapter 7 caps long tool output, and chapter 8 trims and summarizes old turns. This book keeps the conversation in the harness, as a slice of messages, and uses the chat completions API (chapter 2's endpoint), in which every request carries the whole conversation. That works with every provider, and what the model sees on any call is a value you can print.

The Decide stage checks after each turn: did the model answer, and how many turns have run? Two more things can end a run: a deadline passing, or a person pressing Ctrl-C. In Go, both arrive through a `context.Context`, and the loop you build checks it before every turn; chapter 5 also stops a run partway through a turn.

Every chapter in the book improves one of these four stages, and its first page shows which. When something goes wrong in an agent, it helps to ask which stage it went wrong in. An agent that never stops has a Decide problem. An agent that forgets what it read twenty turns ago has a Build-the-request problem.

WATCH OUT

A loop with no turn limit, the most turns a run may take before the harness stops it, can run forever. With a real model, it can also keep costing money the whole time. Every loop in this book has one, and the first loop you write has it before it has anything else.

What should the limit be? No number is right in advance. The 40 runs earlier in the chapter took four to six turns, so this chapter's program allows ten: room for a model that wanders a little, without letting a stuck one run long. Chapter 5 sets the limit from runs like these, and adds a budget in dollars, which bounds a run more directly than counting turns.

Agent frameworks choose a default for you, and the defaults disagree. The OpenAI Agents SDK stops after 10 turns with an error. Google's Agent Development Kit for Python allows 500 model calls. LangGraph counts steps of its own, about two per turn, and now allows 10,000 of them by default, up from 25. Anthropic's tool runner, in its official SDKs including Go's, sets no limit unless you pass one, and when it reaches a limit you did set, it ends the loop without reporting an error. No framework knows your task. Whatever a framework's default, set the limit yourself, from what your task needs, and treat reaching it as a failure the caller should hear about.

■ A loop with no model

You'll build the loop before connecting a real model, for two reasons. The loop is ordinary Go: a `for` loop, a slice of messages, and some function calls. And the model is, from the loop's point of view, just something that returns the next reply. Put a scripted stand-in in its place, and you can run and test the loop with no API key, no network, and no cost.

The book tests against three kinds of model, each for what it can show. A scripted model, this chapter's, checks the loop's rules: it is exact and free, and it knows nothing about choosing tools. Recorded replies, from chapter 2 on, replay a real model's output byte for byte, so a test sees real JSON, real token counts, and real mistakes, with no network. Live runs are the only way to learn how a model behaves, and they vary and cost money, so chapter 13 runs them on purpose, many times, and counts the results.

The code for this chapter is in the `ch01` directory of the companion repository. The agent is a package, `agent`, and a small `main` package runs it. Run it now to see where the chapter is going:

```
$ go run ./ch01
```

You'll build it in five steps: the types, a scripted model, the tools, the loop, and the program that ties them together.

Step 1: the types

Start with the model itself. To the loop, a model is anything that can produce the next reply:

LISTING 2. THE MODEL INTERFACE (`AGENT/TYPES.GO`)

```
// Model is anything that can produce the next reply in a conversation:
// the scripted fake in this chapter, a real model from chapter 2.
type Model interface {
    Generate(ctx context.Context, req Request) (Reply, error)
}
```

The loop needs one thing from a model: the next reply. Because that is a single method, the scripted model in this chapter and the real client in chapter 2 can both provide it, and tests never need the network. Nothing in the loop will know which one it is talking to.

A request is the conversation plus the tools the model may use, and a message is one entry in the conversation:

LISTING 3. REQUESTS AND MESSAGES (AGENT/TYPES.GO)

```
// Request is what the harness sends the model on each turn.
type Request struct {
    Messages []Message
    Tools    []ToolDef
}

// Message is one entry in the conversation.
type Message struct {
    Role      string    // "system", "user", "assistant", or "tool"
    Content   string    // empty when the message has tool calls
    ToolCalls []ToolCall // set on assistant messages that call tools
    ToolCallID string    // on tool messages: the call this answers
    IsError   bool      // set on tool messages that report a failure
}
```

A message from the model can hold text, tool calls, or both. A message from a tool answers exactly one call, named by `ToolCallID`, and `IsError` marks a tool that failed. `IsError` is the harness's own flag; chapter 4 shows how a failure reaches an API that carries only the message text.

Tools have two sides. The model sees a definition: a name, a description, and a JSON Schema for the arguments. When it wants a tool, it sends back a tool call with an ID, the tool's name, and the arguments as JSON:

LISTING 4. TOOL DEFINITIONS AND TOOL CALLS (AGENT/TYPES.GO)

```
// ToolDef describes a tool to the model.
type ToolDef struct {
    Name      string
    Description string
    Parameters jsontext.Value // a JSON Schema
}

// ToolCall is the model asking the harness to run a tool.
type ToolCall struct {
    ID      string
    Name    string
    Arguments jsontext.Value // JSON the model wrote; may be invalid
}
```

`jsontext.Value` comes from `encoding/json/jsontext`, part of the `encoding/json/v2` packages in Go 1.27. It holds raw JSON bytes to be decoded later, like the older `json.RawMessage`; for now,

WHAT AN AGENT IS

treat it as a `[]byte` holding JSON. Chapter 2's Go note covers `json/v2`. The arguments are left raw because each tool decodes its own, and because a model can write JSON that doesn't decode, which the harness has to handle.

The harness's side of a tool pairs the definition with the Go function that runs it:

LISTING 5. A TOOL, FROM THE HARNESS'S SIDE (AGENT/TYPES.GO)

```
// Tool is the harness's side of a tool: the definition the model sees,  
// and the Go function that runs it. The model never sees Run.  
type Tool struct {  
    Def ToolDef  
    Run func(ctx context.Context, args jsontext.Value) (string, error)  
}
```

The model never sees `Run`. It sees the definition, asks for the tool by name, and your code decides what happens.

The last two types are the run's settings and its outcome:

LISTING 6. LIMITS AND RESULTS (AGENT/TYPES.GO)

```
// Limits holds the settings that bound a run.  
type Limits struct {  
    MaxTurns int  
}  
  
// Result is what a run produced, whether or not it ended in an error.  
type Result struct {  
    Messages []Message  
}
```

They hold one field each for now. `Run` returns a `Result` even when it stops with an error, so a run that ended early can still be shown, saved, or debugged: when something goes wrong in turn 9, the first eight turns are what you need to understand it. A `MaxTurns` of 0 allows no turns at all, so always set it. Later chapters add fields (a cost budget, how much a run spent), and they only ever add: no field is changed or removed, so the code you write in this chapter still compiles in chapter 17. The zero value of every new field will mean "not used".

Step 2: a scripted model

The stand-in model replays a script of replies:

LISTING 7. A SCRIPTED MODEL (AGENT/FAKE.GO)

```

// Fake is a Model that replays a script of replies. It picks the next
// reply by counting the assistant messages already in the conversation,
// so it stays in step with the loop.
type Fake struct {
    Script []Message
}

// Generate returns the next scripted reply.
func (f *Fake) Generate(_ context.Context, req Request) (Reply, error) {
    n := 0
    for _, m := range req.Messages {
        if m.Role == "assistant" {
            n++
        }
    }
    if n >= len(f.Script) {
        return Reply{}, fmt.Errorf("reply %d: %w",
            n+1, ErrScriptExhausted)
    }
    return Reply{Message: f.Script[n]}, nil
}

```

It picks its next reply by counting the assistant messages already in the conversation, rather than keeping its own counter. A real model works the same way: everything it knows about the run is in the request. If the loop sent the conversation wrong, say by dropping a message, the fake would reply out of step, and a test would catch it.

Three small helpers make scripts readable. `Calls` is a reply that calls tools, `Call` is one tool call, and `Says` is a reply with text and no tool calls, which is an answer:

LISTING 8. HELPERS FOR WRITING SCRIPTS (AGENT/FAKE.GO)

```

// Says is a scripted reply with text and no tool calls: an answer.
func Says(text string) Message {
    return Message{Role: "assistant", Content: text}
}

// Calls is a scripted reply that calls tools.
func Calls(calls ...ToolCall) Message {
    return Message{Role: "assistant", ToolCalls: calls}
}

```

```
// Call is one scripted tool call; args is JSON.
func Call(id, name, args string) ToolCall {
    return ToolCall{ID: id, Name: name, Arguments: jsonext.Value(args)}
}
```

One scripted reply can hold several tool calls, as three turns of the bug fix did.

Step 3: four tools

The tools work on a small in-memory file system, `FS`, which is a map from file names to their contents. Nothing in this chapter touches your disk. `FS` has two methods, `List` and `Read`, and the project starts with the two `duration` files from the bug fix.

All four tools decode their arguments with one small generic helper:

LISTING 9. DECODING A TOOL'S ARGUMENTS (AGENT/TOOLS.GO)

```
// decodeArgs decodes a tool call's arguments. When they don't decode,
// the error says what was wrong and shows the shape the tool expects,
// so the model can fix its next call.
func decodeArgs[T any](args jsonext.Value, example string) (T, error) {
    var v T
    if err := json.Unmarshal(args, &v); err != nil {
        return v, fmt.Errorf("bad arguments: %w; send JSON like %s",
            err, example)
    }
    return v, nil
}
```

The error matters as much as the decoding. A model that sends `{"path": 5}` gets back a message that says what was wrong and shows the shape to send, and it can fix its next call.

Here is `read_file`:

LISTING 10. THE READ_FILE TOOL (AGENT/TOOLS.GO)

```
type pathArgs struct {
    Path string `json:"path"` ❶
}

func readFile(fs FS) Tool {
    return Tool{
        Def: ToolDef{
            Name: "read_file",
```

```

    Description: "Read a file in the project.",
    Parameters: jsoncontext.Value(`{
        "type": "object",
        "properties": {"path": {"type": "string"}},
        "required": ["path"]
    }`),
},
Run: func(_ context.Context,
    args jsoncontext.Value) (string, error) {
    a, err := decodeArgs[pathArgs](args,
        `{"path": "a.go"}`) ❷
    if err != nil {
        return "", err
    }
    return fs.Read(a.Path)
},
}
}
```

- ❶ The struct tag maps the JSON field `path` to the Go field `Path`.
- ❷ The example in the second argument is what the model sees if its arguments don't decode.

 **WATCH OUT**

`encoding/json/v2` matches field names exactly, so `"path"` does not fill a field named `Path` unless the field has a `json:"path"` tag. Leave the tag off and the tool receives an empty path, with no error. Tag every field that a model fills.

The other three tools follow the same pattern:

Table 5. The other tools

Tool	What it does
<code>list_files</code>	Returns the file names, one per line.
<code>edit_file</code>	Takes a path, the text to replace, and its replacement. The old text must appear exactly once, so an edit can't change the wrong place.
<code>run_tests</code>	A stand-in for now: it checks whether <code>duration.go</code> contains the fixed line and returns either the failing output or a passing line. Chapter 4 replaces it with a real <code>go test</code> run.

A failing test is not a tool error. `run_tests` succeeded at its job, which was to run the tests and report what happened; the failure is information for the model. Tool errors are for when the tool itself couldn't do its job, like a file that doesn't exist.

IN PRODUCTION

The in-memory file system can't escape its map. Real tools read your disk and run commands with your permissions, so they need limits on what they can touch. Chapter 4 starts with a scrubbed environment for running tests, chapter 14 builds a full sandbox, and chapter 17 runs the finished agent on a real service.

Step 4: the loop

Here is the loop, about 30 lines:

LISTING 11. THE AGENT LOOP (AGENT/LOOP.GO)

```
// Run runs the agent loop: it sends the conversation to the model, runs
// any tools the model asks for, and repeats until the model answers
// without calling a tool, the turn limit is reached, or ctx is done. It
// returns the conversation so far even when it returns an error.
func Run(ctx context.Context, m Model, tools []Tool,
  msgs []Message, lim Limits) (Result, error) {
  defs, byName := index(tools) ❶
  msgs = slices.Clone(msgs)     ❷
  for range lim.MaxTurns {      ❸
    if err := ctx.Err(); err != nil { ❹
      return Result{Messages: msgs}, fmt.Errorf("run: %w", err)
    }
    req := Request{Messages: msgs, Tools: defs}
    reply, err := m.Generate(ctx, req) ❺
    if err != nil {
      err = fmt.Errorf("generate: %w", err)
      return Result{Messages: msgs}, err
    }
    msgs = append(msgs, reply.Message) ❻
    if len(reply.Message.ToolCalls) == 0 {
      return Result{Messages: msgs}, nil ❼
    }
    for _, call := range reply.Message.ToolCalls {
      msgs = append(msgs, runTool(ctx, byName, call)) ❽
    }
  }
}
```

```
    return Result{Messages: msgs}, ErrMaxTurns
}
```

- ① `index` splits the tools into the definitions the model sees and a lookup by name. Two tools with the same name make it panic, since that is a mistake in the program.
- ② `Run` works on its own copy of the conversation, so appending never writes into the caller's slice.
- ③ The turn limit. `for range lim.MaxTurns` runs the body that many times (ranging over an integer, Go 1.22).
- ④ If the caller has cancelled `ctx`, or its deadline has passed, the run stops here, before spending another model call. The error wraps `ctx.Err()`, so callers can check it with `errors.Is(err, context.Canceled)`.
- ⑤ Stages 1 and 2: build the request and call the model. `msgs` is the whole conversation, so every call sends all of it. `ctx` goes along, so a real client can abandon a call when the run is cancelled.
- ⑥ The reply joins the conversation before its tools run, so each tool result that follows answers a call the conversation already holds. Chapter 5 adds checks on the reply before this line, so a bad reply never gets in.
- ⑦ Stage 4: a reply with no tool calls is an answer, and the run is done.
- ⑧ Stage 3: every tool call gets a tool message, in the order the model asked. Each tool gets `ctx` too.

<> GO NOTE

`context.Context`

A `context.Context` carries three things down a call chain: a signal that the work should stop, an optional deadline, and (since Go 1.20) the reason it stopped, read with `context.Cause`. You derive new contexts from existing ones, with `context.WithCancel`, `WithTimeout`, or their `Cause` variants, and cancelling a parent cancels everything derived from it. That makes it the natural way to stop an agent: one `ctx` for the run, passed to every model call and every tool, so one cancellation reaches all of them. It is always the first parameter of anything that does I/O or can block, and every chapter of this book extends what the loop does with it: a deadline on each model call (chapter 2), stopping partway through a reply's tool calls (chapter 5), an idle timer on a stream (chapter 6), cancelling parallel work (chapter 9), telling a tool's own timeout apart from the run being cancelled (chapter 12), and shutting down without losing work (chapter 15).

Running one tool turns every kind of problem into a message the model can read:

LISTING 12. RUNNING ONE TOOL CALL (AGENT/LOOP.GO)

```
// runTool runs one tool call and returns the tool message that answers
// it. Every problem becomes an error message the model can read.
func runTool(ctx context.Context, byName map[string]Tool,
    call ToolCall) Message {
    msg := Message{Role: "tool", ToolCallID: call.ID}
    t, ok := byName[call.Name]
    if !ok {
        names := slices.Sorted(maps.Keys(byName))
        msg.IsError = true
        msg.Content = fmt.Sprintf("no tool named %q; the tools are %s",
            call.Name, strings.Join(names, ", "))
        return msg
    }
    out, err := t.Run(ctx, call.Arguments)
    if err != nil {
        msg.IsError = true
        msg.Content = err.Error()
        return msg
    }
    msg.Content = out
    return msg
}
```

A tool that failed, arguments that didn't decode, and a tool name that doesn't exist all become a tool message with `IsError` set and a reason. None of them stops the run.

The reason matters. The OpenAI Agents SDK, by default, answers a failed tool with the same sentence every time: "An error occurred while running the tool. Please try again." That keeps internal details out of the conversation, but it gives the model nothing to fix: if the arguments named a file that doesn't exist, trying again sends the same arguments. The tools in this book return errors written for the model to act on, such as `old text appears 0 times in duration.go, want exactly once`, and keep secrets out of them by what they include rather than by hiding everything. Chapter 7 covers writing those messages, and removing credentials from tool output before the model sees it.

Why send an unknown tool back instead of stopping? Models get names wrong now and then, and they can usually correct themselves once they see the real list, which the message includes. In the 40 runs earlier in the chapter, the models made 236 tool calls, and every one named a real tool with arguments that decoded. With four tools and a short task, that is expected. With dozens of

tools, similar names, or a long conversation, mistakes come more often, and chapter 7 measures how often a model picks the right tool.

There are two other ways to handle a bad name, and both are worse. Stopping the run throws away every turn before the typo. Guessing what the model meant, say by running the tool with the closest name, runs something the model never asked for, with arguments written for a different tool. Sending the error back costs one turn, and the model makes the correction itself. The turn limit still bounds a model that keeps getting it wrong; it is the only thing in this loop that guarantees the loop ends.

BACKGROUND

One loop, many names

The loop has been described many times, under different names. The ReAct paper (2022) had the model alternate "Thought", "Action", and "Observation": it reasoned in text, named an action, and read the result. These map onto the four stages: building the request and calling the model produce the thought and the chosen action; running the tools is the action; the tool result is the observation; and deciding is the harness's choice to go around again.

One thing has changed since ReAct. In 2022 the model wrote its actions as plain text, such as `Action: read_file[duration.go]`, and the harness parsed that text. Today's model APIs return tool calls as structured data, with a name and JSON arguments, so the harness never has to parse the action out of prose. Chapter 4 shows what a tool call looks like on the wire.

The same loop now ships inside the providers' own libraries. The OpenAI Agents SDK runs it in `Runner.run`; Anthropic's SDKs, Go's among them, offer a tool runner; Google's Agent Development Kit has its own. Each is this loop plus features: handing a task from one agent to another, checks on input and output, tracing. Chapter 16 compares them with the harness you build in this book, once you know what each part does.

Step 5: putting it together

The script replays the bug fix from the start of the chapter:

LISTING 13. THE SCRIPT (SCRIPT.GO)

```
// script holds the replies from the run at the start of the chapter,
// recorded from claude-haiku-5.5, so the loop can replay it without a
// model. The model's IDs are shortened, and its text before tool calls
// is left out.
var script = []agent.Message{
    agent.Calls(
        agent.Call("call_1", "run_tests", `{}`),
```

```

    agent.Call("call_2", "list_files", `{}`),
),
agent.Calls(
    agent.Call("call_3", "read_file", `{"path": "duration.go"}`),
    agent.Call("call_4", "read_file",
        `{"path": "duration_test.go"}`),
),
agent.Calls(
    agent.Call("call_5", "edit_file", `{
        "path": "duration.go",
        "old": "'m': time.Second,",
        "new": "'m': time.Minute,"
    }`),
    agent.Call("call_6", "run_tests", `{}`),
),
agent.Says("The tests now pass. The cause was a typo in the unit " +
    "table in duration.go: the 'm' (minute) entry mapped to " +
    "time.Second. So \"45m\" parsed as 45 seconds, and \"1h30m\" " +
    "parsed as 1h + 30 seconds. I changed it to time.Minute."),
}

```

`main` makes the context: Ctrl-C cancels it, and the loop stops before its next turn.

LISTING 14. CANCELLING THE RUN WITH CTRL-C (MAIN.GO)

```

func main() {
    // Ctrl-C cancels ctx, and the loop stops before its next turn.
    ctx, stop := signal.NotifyContext(context.Background(),
        os.Interrupt)
    defer stop()
    if err := run(ctx, os.Stdout); err != nil {
        log.Fatal(err)
    }
}

```

And `run` loads the project into an `FS`, starts the conversation with the task, and runs the loop:

LISTING 15. RUNNING THE SCRIPTED TASK (MAIN.GO)

```

func run(ctx context.Context, w io.Writer) error {
    files, err := load(project, "testdata/duration")
    if err != nil {
        return err
    }
}

```

WHAT AN AGENT IS

```
msgs := []agent.Message{{Role: "user", Content: task}}
model := &agent.Fake{Script: script}
lim := agent.Limits{MaxTurns: 10}
res, err := agent.Run(ctx, model, agent.Tools(files), msgs, lim)
if perr := printConversation(w, res.Messages); perr != nil {
    return perr
}
if err != nil {
    return fmt.Errorf("run agent: %w", err)
}
return nil
}
```

Run it:

```
$ go run ./ch01
task: The tests in this project fail. Find out why and fix it.

turn 1
model asks: run_tests {}
model asks: list_files {}
tool result: --- FAIL: TestParse (0.00s) (5 lines)
tool result: duration.go, duration_test.go

turn 2
model asks: read_file {"path": "duration.go"}
model asks: read_file {"path": "duration_test.go"}
tool result: // Package duration parses durations suc... (44 lines)
tool result: package duration (36 lines)

turn 3
model asks: edit_file
               path: "duration.go"
               old: "'m': time.Second,"
               new: "'m': time.Minute,"
model asks: run_tests {}
tool result: edited duration.go
tool result: ok    duration    0.002s

turn 4
model says: The tests now pass. The cause was a typo in the unit table in
            duration.go: the 'm' (minute) entry mapped to time.Second. So
```

"45m" parsed as 45 seconds, and "1h30m" parsed as 1h + 30 seconds. I changed it to time.Minute.

The output is the table from the start of the chapter, produced by your own loop. The program prints each turn after the run finishes; chapter 6 prints them as they happen.

The tests in `ch01/agent` check the loop's behavior one rule at a time: it stops on an answer, it stops at the turn limit, every tool call gets a tool message (including several calls in one reply), tool errors go back with `IsError`, and an unknown tool gets the list of real ones. They run in milliseconds and need no network:

```
$ go test ./ch01/...
```

■ When not to use an agent

An agent is not always the right tool. Every turn is a model call, which costs money and takes seconds. Mistakes compound: a wrong guess in turn 3 shapes everything after it. And two runs of the same task can take different paths, as `glm-5.3-flash` did on the bug fix, which makes an agent harder to test than ordinary code.

When you know the steps in advance, a workflow is usually cheaper, faster, and easier to test. The bug fix itself shows it. Written as a workflow, it is three steps fixed in code: run the tests and read both files; send all of it to the model in one request asking for one edit, as JSON; apply the edit and run the tests again. On the same four models, ten runs each:

Table 6. The bug fix as a workflow and as an agent, ten runs per model (costs are medians)

Model	Workflow: tests pass	Workflow: cost	Agent: tests pass	Agent: cost
<code>claude-haiku-5.5</code>	9 of 10	\$0.00013	10 of 10	\$0.00080
<code>gpt-5.6-luna</code>	10 of 10	\$0.00022	10 of 10	\$0.00089
<code>glm-5.3-flash</code>	10 of 10	\$0.00018	10 of 10	\$0.00076
<code>deepseek-v4.1-flash</code>	10 of 10	\$0.00019	10 of 10	\$0.00116

The workflow cost a quarter to a sixth as much, and its one model call took from under a second to two and a half seconds, against seven and a half to thirteen and a half for the agent's runs. For this bug, the workflow is the better program.

Its one failure shows what it gives up. In that run the model wrote a raw tab character (Go code is indented with tabs) inside a JSON string, which JSON doesn't allow, so the edit couldn't be decoded, and the workflow had no second step in which to try again. The agent would have sent the de-

coding error back, and the model could have fixed its call on the next turn. The workflow works because someone knew the steps: this bug needs these two files and one edit. Give it a bug whose cause is in a third file, or one that needs a test run to locate, and the fixed steps don't fit. An agent earns its cost when the next step depends on what the last one found. Chapter 5 ends with a table of shapes, from a fixed chain of model calls to several agents working together, once you have a real agent to compare them against.

■ Wrap-up

😊 GOPHER SAYS

Your tools run with your permissions: check what the model asks for before your code does it.



Try it

1. Change the script so the model asks for a tool that doesn't exist, such as `read_files`. Then add a scripted reply that corrects the name after the error, and watch the run recover.
2. Lower `MaxTurns` until the task can't finish. `Run` returns `ErrMaxTurns` and the conversation up to that point; check both.
3. Make the scripted edit wrong (replace `time.Second` with `time.Hour`) and watch `run_tests` keep failing.
4. Give the run a deadline that has already passed, with `context.WithTimeout(ctx, 0)`. What does `Run` return, how many turns ran, and what does `errors.Is(err, context.DeadlineExceeded)` say?
5. Print each turn as it happens: write a type that implements `Model` by wrapping another `Model`, prints the request's last message, and passes the call on. Hint: a struct with a field holding the wrapped `Model`, and a `Generate` method that calls it.

Further reading

- Anthropic, "Building effective agents": the workflow and agent distinction, and the common shapes, with examples.
- Simon Willison on the definition of "agent": where the one-line definition comes from.

- [HumanLayer](#), "12-Factor Agents": principles for agents in production, including owning your context window.

What you built

You built a working agent loop, a scripted model to drive it, four tools over an in-memory project, and the types the rest of the book grows. The files are in `ch01`: the `agent` package (`types.go`, `fake.go`, `memfs.go`, `tools.go`, `loop.go`) and the program that runs it (`main.go`, `script.go`).

What's missing is a real model. Chapter 2 builds a client for one, using only the standard library, that satisfies the `Model` interface you wrote here; chapter 4 connects the two.

This is a sample

This is a sample of *Building AI Agents in Go*. The full book has 17 chapters. After chapter 1, you build:

- A real model that finds and fixes a failing test in a Go project, with a loop you wrote yourself (chapter 4).
- Tools the model can call, and MCP servers that give it more (chapters 7 and 10).
- Evals graded by hidden tests the agent never sees (chapter 13).
- Sandboxing and human approval for the calls that can do damage (chapter 14).
- Runs that survive a crash and pick up where they stopped (chapter 15).
- A capstone crash-triage agent, measured across three models (chapter 17).

Every listing comes from tested code, and every chapter's code runs offline from saved replies. The code archive comes with the book.

Get the full book for \$29 at <https://golang-agents.com>.