

AI Agents, Engineered

Designing, Building, Evaluating, and
Operating Agents You Can Verify



Enrique Gutiérrez

AI Agents, Engineered

Designing, Building, Evaluating,
and Operating Agents You Can Verify

Enrique Gutiérrez

AI Agents, Engineered

Designing, Building, Evaluating, and Operating Agents You Can Verify

Copyright © 2026 Enrique Gutiérrez. All rights reserved.

No part of this book may be reproduced or transmitted in any form without the prior written permission of the author, except for brief quotations in reviews and scholarly work. Quoted material remains the property of its respective authors and is cited in the notes.

Product and company names mentioned in this book are used only as examples and may be trademarks of their owners. The author has made every effort to ensure the accuracy of the information herein; however, it is provided without warranty, and the author shall not be liable for any damages arising from its use.

Published on Leanpub. First edition, 2026.

Contents

Preface	ix
I Foundations	1
1 What Is an Agent?	2
1.1 The Agent Idea	2
1.2 Agents, Workflows, and Chatbots	4
1.3 Do You Even Need an Agent?	8
2 The Engine: How Language Models Work	11
2.1 LLMs as Next-Token Predictors	11
2.2 Tokens and the Context Window	14
2.3 How Text Is Generated	18
2.4 Prompting Fundamentals	22
2.5 Structured Output and Function Calling	26
2.6 Limitations and Failure Modes	29
II Building Agents	34
3 The Agent Loop	35
3.1 The Loop	35
3.2 ReAct: Interleaving Reasoning and Acting	39
3.3 A Minimal Agent from Scratch	42
3.4 Stop Conditions, Budgets, and Frameworks	46
4 Planning, Reasoning, and Self-Correction	50
4.1 Task Decomposition and Planning	50
4.2 Reasoning Techniques	56
4.3 Reflection and Self-Correction	60

5	Tools and the Action Space	66
5.1	Choosing the Set and Granularity of Tools	67
5.2	The Tool Interface	71
5.3	Retrofitting Programs Built for Humans	74
5.4	Code Execution as a Universal Action	76
5.5	Computer Use and Browser Automation	80
6	Skills, Protocols, and Interoperability	84
6.1	What a Skill Is	85
6.2	Progressive Disclosure	88
6.3	Skills versus Tools versus Protocols	92
6.4	Protocols and Interoperability	94
6.5	Skill Design Patterns and Composition	97
III	Context Engineering	103
7	Managing the Context Window	104
7.1	Context Engineering versus Prompt Engineering	104
7.2	A Scarce, Attention-Limited Resource	108
7.3	How Context Fails	111
7.4	The Four Operations: Write, Select, Compress, Isolate	116
8	Retrieval and Knowledge	120
8.1	Retrieval-Augmented Generation	121
8.2	Beyond Naive Retrieval	124
8.3	Agentic Retrieval	127
8.4	RAG versus Long Context versus Fine-Tuning	130
9	Memory: Working State Across Long Runs	134
9.1	The Memory Taxonomy	135
9.2	Memory Architectures	137
9.3	Keeping a Long Run Healthy	142
9.4	Isolation and Quarantine	145
IV	Patterns and Choices	150
10	Workflows and Composition Patterns	151
10.1	The Augmented LLM as the Base Unit	152
10.2	Prompt Chaining, Routing, and Parallelization	154
10.3	The Evaluator–Optimizer Loop	160

11 Multi-Agent Systems	166
11.1 The Orchestrator–Worker Pattern	167
11.2 Focused, Self-Contained Worker Context	170
11.3 Coordination Failure Modes	173
11.4 Named Subagent Archetypes	177
11.5 Human-Orchestrated Parallel Agents	179
11.6 The Merge Wall and the Single-versus-Multi Verdict	183
12 Oversight and Autonomy	187
12.1 Approval Gates and Escalation	188
12.2 Reviewing Intent and Outcomes, Not Lines	192
12.3 Interruptibility, Steering, and the Autonomy Dial	196
13 Writing the Outer Loop	201
13.1 A Short History of the Loop	202
13.2 The Goal Function	205
13.3 Loop Engineering	208
13.4 When Loops Compound	211
13.5 Keeping the Loop Safe	215
14 Choosing Your Approach: When <i>Not</i> to Build an Agent	219
14.1 The Ladder Revisited	220
14.2 Signs You Need One, Signs You Don’t	223
V Making Agents Reliable	228
15 Observability and Debugging	229
15.1 Traces and Spans	230
15.2 Production Monitoring	235
15.3 Reproducing Nondeterministic Failures	239
15.4 A Taxonomy of Common Agent Bugs	243
16 Evaluating Agents	249
16.1 Why Evaluation Is Hard	249
16.2 Building an Eval Set	253
16.3 LLM-as-a-Judge	256
16.4 Eval-Driven Development and Regression Gates	260
17 Security, Safety, and Guardrails	264
17.1 Prompt Injection	264
17.2 The Lethal Trifecta	268
17.3 Guardrails as a Separate, Deterministic Layer	272
17.4 Least Privilege, Sandboxing, and Human Approval	275
17.5 The Supply Chain	279

18 Reliability, State, and the Harness	284
18.1 Surface Errors Back to the Model	285
18.2 Retries, Timeouts, and Budgets	287
18.3 State and Recovery	289
18.4 Fallbacks, Degradation, and Partial Failure	293
18.5 Harness Engineering	294
18.6 How Much Harness to Build	298
 VI Shipping and Operating	 302
19 Cost, Latency, and Performance	303
19.1 Where Tokens (and Money) Go	304
19.2 Cost Levers	307
19.3 Latency	311
19.4 The Accuracy–Cost–Latency Triangle	315
20 Deploying and Scaling	318
20.1 Serving Shapes	319
20.2 Durable Execution and State at Scale	323
20.3 Sandboxed Execution at Scale	325
20.4 Rate Limits, Queuing, and Backpressure	326
20.5 The Rollout Ladder	329
 VII Applications and the Road Ahead	 333
21 Coding Agents	334
21.1 Why Coding Led	336
21.2 The Coding-Agent Loop and the Range of Tasks	338
21.3 Augmented Coding versus Vibe Coding	340
21.4 Making Code Agent-Legible	344
21.5 The General Lesson	348
22 The Coding Workflow in Practice	351
22.1 The Canonical Loop: Brainstorm, Spec, Plan, Execute	352
22.2 Spec-Driven Development	355
22.3 Test-Driven Development with Agents	358
22.4 Git as the Safety Net	361
22.5 Agentic QA and Review	364
22.6 Understanding Code You Didn’t Write	367

23 Research and Business Agents	372
23.1 Research and Information Agents	373
23.2 The Verification Gap	375
23.3 Business and Operational Agents	380
23.4 Framing Return on Investment Against Risk	384
24 Agent UX and Human Trust	387
24.1 Trust Calibration	388
24.2 Transparency and Progress Feedback	392
24.3 Steerability and Graceful Failure	394
24.4 From Doing to Directing	398
25 Transversal Recipes: Big Patterns That Cut Across Industries	401
25.1 The Autonomy Slider	402
25.2 Autoresearch	404
25.3 The Self-Maintaining Personal Knowledge Base	406
25.4 The Deep-Research Analyst	408
25.5 The Ambient Watcher	410
25.6 Unstructured-to-Structured at Scale	412
25.7 The Queue Triage and Router	413
25.8 The Digital Coworker	415
25.9 The Premortem and Red-Team Simulator	416
26 Agents as Classifiers and Scorers	420
26.1 The Classifier Without a Training Set	421
26.2 Examples as a Textbook, Not a Database	424
26.3 Reason First, Then Explain	426
26.4 From Labels to Scores	428
26.5 Developing It Without Fooling Yourself	430
26.6 Packaging It, and When to Reach for It	433
27 The Frontier and How to Keep Learning	436
27.1 Pairing Generativity with Deterministic Verifiers	437
27.2 The Ecosystem as Durable Layers	442
27.3 Build versus Adopt	446
27.4 Durable Trends, and How to Keep Learning	448
VIII Appendices	453
A A Minimal Agent, Annotated	454
A.1 The Program in Full	455
A.2 The Model Client	457
A.3 The Tool Registry	457

A.4	Executing a Request	459
A.5	The Loop	459
A.6	The Standing Instructions	461
A.7	A Run, End to End	461
A.8	What Is Deliberately Missing	463
B	Glossary	464
C	Annotated Further Reading and Source Map	476
C.1	How the Library Is Organized	476
C.2	The Chapter-by-Chapter Map	477
C.3	Durable Primary References	480
C.4	A Second Shelf: Foundations Assumed and Debates in Motion	484

Preface

This book exists because of a particular moment, and I suspect you have already had it. You hand a language model a task that would take a person half an hour: find out why the build broke, say, or reconcile a messy spreadsheet against the database. Instead of an answer, you watch it *work*. It searches, reads, tries something, hits an error, corrects itself, and comes back a few minutes later, finished. The first feeling is delight. The second feeling arrives a beat later and lasts longer: *I didn't see every step. How much of this can I trust?*

That second feeling is the subject of this book.

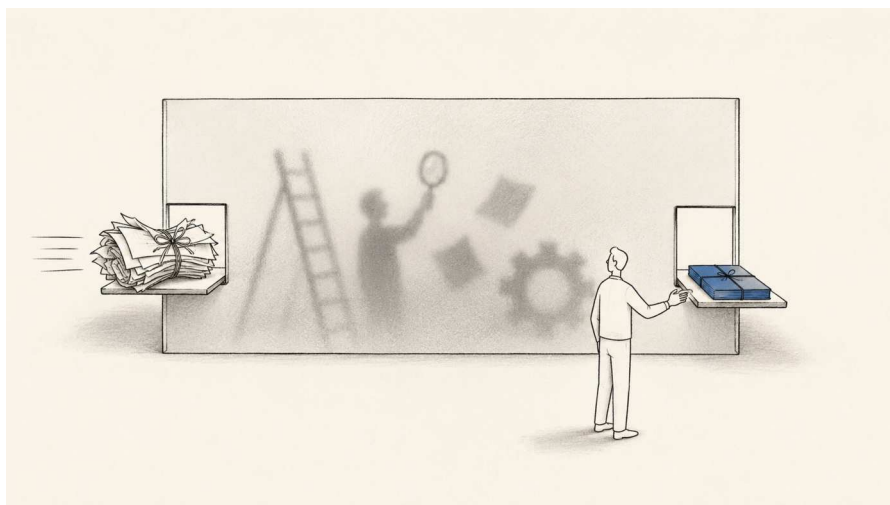


Figure 1: Messy work goes in one side of a frosted-glass wall and finished work comes out the other; the steps between are only silhouettes. The delight is immediate. The doubt outlasts it, and it is the subject of this book.

I wrote it for a practicing engineer—someone comfortable with software but not necessarily with machine learning—who is evaluating agents, or has just started building with them, and wants to get past both the hype and the

dread to the engineering underneath. You will find definitions built from first principles, patterns that recur across every domain where agents do useful work, examples in plain pseudocode, and a persistent interest in what things cost and how they fail. The only prerequisite is ordinary engineering literacy; whenever a term of art appears, I define it on the spot, in plain words.

Two promises shape every chapter, and I want to make them explicit, because together they explain what you will and will not find here.

The first promise is that the book is *time-agnostic*. This field improves at a pace that embarrasses anyone who writes things down; while I was gathering material for these chapters, notes I had captured only months earlier were already going stale, their products renamed and their benchmark numbers overtaken. So I have refused to anchor anything on the season it was written in. No claim in this book depends on which model is currently best, what a token costs this quarter, or how large this year's context windows are. Where a number appears, it is marked as an illustration. The aim is a book that reads as well a few years from now as it does today.

The second promise is that it is *solution-agnostic*. The durable knowledge in this field lives in concepts and patterns—the loop, the tool contract, the context budget, the evaluation harness—and those transfer across every vendor and framework. The syntax of any one product does not transfer, and it ages in months. So this book teaches the pattern, and names concrete tools only in passing, as labeled examples of a category. When you finish it, you should be able to sit down with whatever *harness* (the software shell an agent runs inside) your team has adopted, or with none at all, and know what to build.

One idea recurs so often in these pages that it amounts to a thesis, and you should meet it now: *an agent is only as trustworthy as the signal you can use to verify it*. A language model produces fluent, confident text whether it is right or wrong; its tone tells you nothing you can rely on. The trust you place in an agent therefore has to come from somewhere outside the model: a test suite that passes, a schema that validates, a source you can open and check, a human who approves the irreversible step. Where such a signal exists, you can grant autonomy generously—that, as we will see, is most of why coding became the first domain where agents earned their keep, since code arrives with its own verdict: compile it, run the tests. Where no signal exists, autonomy is a leap of faith, and this book will keep saying so. Nearly every design question in the chapters ahead resolves to the same first move: ask what signal tells you it worked.

A word about vocabulary. “Agent” may be the most overloaded term in modern software; it is applied with equal confidence to a scripted bot reading from a phone tree and to a system trusted to work alone for an afternoon. Jargon is only useful when speaker and listener share a definition; otherwise two people can passionately discuss entirely different things.¹ So the first

¹Simon Willison, “I think ‘agent’ may finally have a widely enough agreed upon definition

chapter does the unglamorous work of pinning the terms down, and the rest of the book holds to them.

As for how to read it: the book is divided into seven parts, ordered as a course. Part I establishes what an agent is and how the engine underneath—the language model—actually works. Parts II and III build one: the loop, the tools, the reusable skills, and the careful management of the model’s working memory. Part IV is about architecture: workflows, multi-agent systems, the dial between oversight and autonomy, the craft of writing the harness’s outer loop, and the discipline of asking whether the simplest thing that works needs an agent at all. Parts V and VI cover what most of real agent engineering turns out to be: evaluation, observability, security, reliability, cost, and deployment. Part VII turns to applications, and to what seems durable on the road ahead. The appendices hold a minimal agent in annotated pseudocode, a glossary, and a map for further reading. Read the parts in order for a grounding, or, once oriented, raid them as a reference; the chapters cross-reference one another and try to stand on their own.

I should be honest about what a book like this can promise. Some details here will be superseded, probably sooner than I would like; I have tried to write only the parts that outlast their examples, and to flag my uncertainty where I have it. The bet this book makes is that principles outlive products. If, a few years from now, the examples feel quaint but the questions—*what signal verifies this? what is the simplest thing that works?*—are still the questions you ask, the book will have done its job.

to be useful jargon now,” simonwillison.net (September 2025). The observation about shared jargon, and the working definition this book adopts in Chapter 1, are his.

Part I

Foundations

Chapter 1

What Is an Agent?

This chapter keeps a promise made in the preface: before anything gets built, we pin down what an agent *is*. The need is practical, and immediate. Vendors label scripted bots “agents”; researchers reserve the word for systems that run unattended for hours; and somewhere in between, you have to make actual decisions with actual budgets. If the term stays fuzzy, you will reach for an autonomous agent where a three-step pipeline would have been cheaper and easier to debug, or you will hand-wire a rigid pipeline for a problem whose steps you genuinely cannot predict. The chapter does three things. It states the agent idea itself, explains why it became practical when it did—deliberately without dates—and lays out the compass the book will steer by. It then draws the working distinction among chatbots, workflows, and agents that every later chapter leans on. And it closes with the question too few people ask out loud: whether you need an agent at all.

1.1 The Agent Idea

Strip away the demos and the marketing, and the idea is small enough to hold in one hand. An *agent* is a language model placed in a loop. It is given a goal and a set of *tools*—functions it can ask your program to run: search this, read that, execute this command. It looks at the state of the task, picks an action, sees the result, and goes again, until it judges the goal met or a stopping rule ends the run. After years of talking past itself, the practitioner community has, at the time of writing, largely converged on a one-sentence version: an agent *runs tools in a loop to achieve a goal*.¹ Every word of that sentence is load-bearing. “Tools” is what connects the model to the world; “loop” is what lets it react to what it finds; “a goal” is what ends the loop—an

¹Simon Willison, “I think ‘agent’ may finally have a widely enough agreed upon definition to be useful jargon now,” simonwillison.net (September 2025). The definition quoted—“An LLM agent runs tools in a loop to achieve a goal”—is his, defended there phrase by phrase.

agent has a stopping condition, which is what separates it from a runaway process. Chapter 3 unpacks the loop in code.

What is genuinely new here is where the control flow lives. In every program you have written until now, the sequence of operations was decided by you, in advance, in code. In an agent, the sequence is decided by the model, at runtime, in response to what it observes. That single change is what makes agents worth a book, and also what makes them expensive, slow, and occasionally alarming. The rest of the book is about both halves of that sentence.

You may reasonably ask why this became practical when it did. Language models were fluent long before anyone trusted one with a loop; fluency was never the missing piece. Three things had to mature. Each will return later in the book, first as machinery and then as a failure mode when it runs short.

The first: models learned to take instructions. A model fresh out of its initial training is a pure continuer of text—ask it a question and it may reply with more questions, because that is a plausible way for text to continue (Chapter 2 shows why). A later stage of training, which the field calls post-training, turns that continuer into something you can give a job to: it follows directions, works toward the stated objective, and stops when done. An agent is a delegation, and delegation requires a worker that accepts the assignment.

The second: text acquired a bridge to action. *Function calling*—the model emits a machine-readable request, “call this function with these arguments,” which your code executes before feeding the result back—gave the model a disciplined channel to the world. The model proposes; your code disposes. Chapter 2 covers the mechanism in detail. The consequence is the point here: a system that could only ever write *about* the world became one that can ask, in a form a program can safely obey, for things to *happen* in it.

The third: the model’s working memory grew. Everything an agent knows in the moment must fit inside the model’s *context window* (the bounded stretch of text a model can consider in a single call). When windows held a few pages, a loop starved: the instructions, the history, and the latest tool result would not fit together. Windows grew to hold something closer to a working session, and loops became sustainable.

Each of these matured as a slope quietly crossing a threshold, which is convenient for both of us: the explanation of “why now” does not expire. It also tells you what to check when some new capability is announced: ask which of the three ingredients just got better.

Before the taxonomy and the machinery, I want to hand you the compass this book steers by: four bearings that will reappear in nearly every chapter. I state them here without their full defense; the defense is the book.

Verifiability comes first. The preface stated the thesis and it bears repeating: an agent is only as trustworthy as the signal you can use to verify it. The model’s own confidence carries no information about whether it is right, so trust must come from outside—a passing test, a validating schema, a source

you can open, a human who approves. The first question to ask of any agent design is the one this book will ask over and over: *what signal tells you it worked?*

Next comes *the scarcity of context*. The context window is finite, and the part that surprises people is that the model’s attention degrades before the space runs out: a fact buried in the middle of a long pile is, in effect, half-forgotten. Treat context as a budget to be spent deliberately. Part III of this book is about spending it well.

Compounding error is the third. An agent chains many steps, and reliability multiplies across them. Suppose, purely as an illustration, that each step succeeds 95% of the time; a run of twenty chained steps then succeeds about $0.95^{20} \approx 36\%$ of the time. The particular numbers vary; the arithmetic does not, and it is merciless. Long loops survive only with recovery, checkpoints, and verification along the way. This book spends whole chapters on that machinery.

And the last bearing is *the simplest thing that works*. The engineering advice that has aged best in this young field is blunt: find the simplest solution possible, and add complexity only when it demonstrably pays—which may mean not building an agent at all.² Sometimes the simplest thing is a single prompt. Sometimes it is a regular expression. The last section of this chapter takes this bearing seriously enough to give it its own discussion. Figure 1.1 sets the four bearings on a single rose, with the needle resting on the first of them.

A compass is a modest instrument; it will not design your system for you. But whenever the book reaches a fork—more autonomy or less, another agent or a plain script, richer context or leaner—you will watch the same four bearings decide it, and by the end I hope they will be deciding your forks too.

1.2 Agents, Workflows, and Chatbots

Let me put three systems in front of you. Decide which of them is the agent before reading on.

The first is a question-and-answer assistant. You paste in a function and ask what it does, and it explains; you paste a diff and ask for a draft commit message, and it writes one; then it waits for your next message. The second processes every incoming support ticket the same way: a model call classifies the ticket, ordinary code fetches the matching policy document, a second model call drafts a reply, a third checks the draft against a rubric, and the result lands in a human review queue—four steps, in that order, every time. The third is told “the checkout tests are failing—fix them,” and, unsupervised,

²Erik Schluntz and Barry Zhang, “Building Effective Agents,” Anthropic engineering blog (2024): “we recommend finding the simplest solution possible, and only increasing complexity when needed. This might mean not building agentic systems at all.” The essay’s workflow-agent distinction anchors the next section as well.

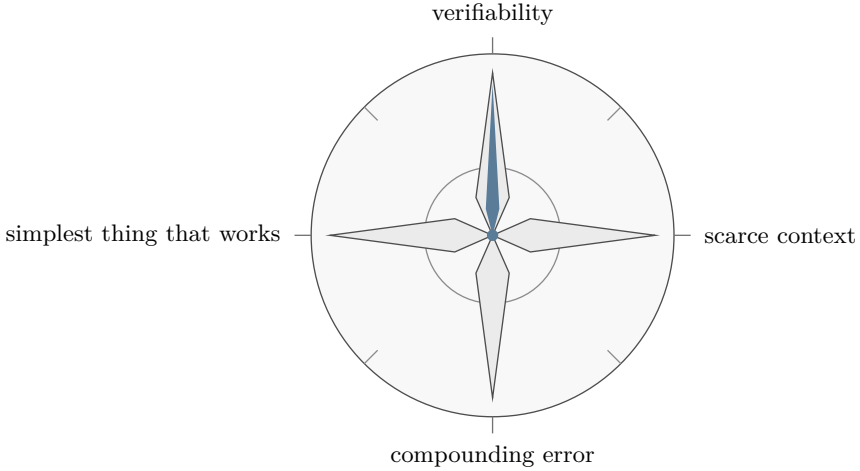


Figure 1.1: The book’s compass. Four bearings decide the design forks in almost every later chapter, and the needle rests on the first of them—verifiability—because an agent is only as trustworthy as the signal you can use to check it. The other three, the scarcity of context, compounding error, and the simplest thing that works, are read alongside it, not instead of it.

it searches the codebase, reads three files, edits one, runs the tests, reads the new failure, edits again, and stops when the suite is green.

Most people’s instinct sorts these instantly: the third one feels different in kind, and the instinct is right. The useful work is saying precisely *why*. All three may be built on exactly the same underlying model. What differs is the question the last section called the heart of the idea: who decides what happens next. In the first system, you do, one message at a time. In the second, your code does; the model only fills in content along a route that was fixed before the first ticket ever arrived. In the third, the *model* decides: which file to read, whether to edit or rerun, when to stop. That question—who owns the control flow—is the entire distinction, and it gives us the three working terms this book uses everywhere. Figure 1.2 draws all three side by side, differing only in who decides the next step.

A *chatbot* is turn-by-turn conversation. The human drives every step; the model generates text and waits. There is no loop and nothing is executed. Chatbots are genuinely useful and often the right tool, and their failure surface is small in a way I want on the record: the worst a chatbot can produce is a wrong answer, because acting on the world is out of its reach.

A *workflow* is several model calls (and tool calls) wired together along paths your code defines in advance. The model fills in the content at each step—classify this, draft that—but your code decides which step comes next.

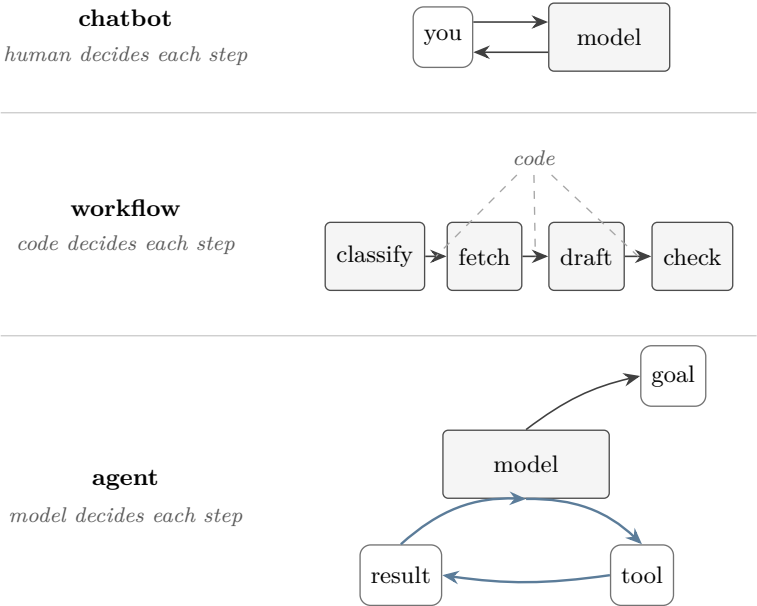


Figure 1.2: Chatbot, workflow, and agent differ in one thing only: who decides the next step. In a chatbot the human drives every turn; in a workflow the path is fixed in code and the model merely fills in each step; in an agent the model owns the loop, choosing tools and reading results until it reaches the goal. The model-owned loop (in accent) is the whole subject of this book.

The recurring shapes have names you will meet again in Chapter 10: *chaining* (one call’s output feeds the next), *routing* (classify, then branch), and *parallelization* (fan out several calls, then aggregate). The ticket pipeline above is a workflow, and you could have drawn its flowchart before switching it on.

An *agent* hands the model the control flow. One widely cited engineering essay draws the line exactly here: workflows are “systems where LLMs and tools are orchestrated through predefined code paths,” while agents are systems where models “dynamically direct their own processes and tool usage.”³ The number of steps, and which step comes when, exists nowhere in your code; the model chooses as it goes, reacting to whatever it just observed. That is why the test-fixing system feels different: its transcript reads like the log of a colleague’s afternoon, and only distantly like the execution of a program.

Underneath the last two sits the same atom, worth naming because the patterns you will meet in Chapter 10 are assembled from it. The *augmented LLM* is a model call enhanced with three things: *retrieval* (pulling relevant information in at request time), *tools* (functions it can request), and *memory* (some way of carrying forward what matters). A workflow is several of these atoms wired together by your code along a fixed path. An agent is one of these atoms placed inside a loop, deciding for itself what to do on each pass. (A chatbot, in this picture, is the bare model call with the augmentations mostly unused.) Same atom, two arrangements—keep that picture and the taxonomies stop feeling mysterious.

Here is the litmus test I will use for the rest of the book. If you can confidently draw the control-flow diagram before the request arrives, you are looking at a workflow. If the diagram can only be drawn in hindsight, once the model has reacted to what it found, you are looking at an agent.

It is tempting to treat chatbot, workflow, and agent as three boxes. The truth is a dial (Figure 1.3). At one end, the model generates text and a human decides everything. Near that same end sits the fixed pipeline (the ticket system from a moment ago), where code owns every step and the model only fills in content. A small step up, a workflow grants the model one decision: a router picks which hardcoded branch handles the request. Further along, an agent loops freely over its tools but pauses for human approval before anything consequential—sending the email, spending the money, deleting the data; you will find much of production here. At the far end, the model owns the path end to end for many steps without per-step confirmation: maximum capability, maximum *blast radius*—the damage a wrong action could do before anything stops it, a term Chapter 17 will make precise. What moves along this dial is a single quantity, how much of the control flow the model owns,

³Erik Schluntz and Barry Zhang, “Building Effective Agents,” Anthropic engineering blog (2024); both phrases are quoted from the essay, which also names the *augmented LLM* building block described in the next paragraph. An earlier survey—Lilian Weng, “LLM Powered Autonomous Agents,” lilianweng.github.io (2023)—describes the same anatomy from the inside: a model “brain” supported by planning, memory, and tool use.

and mature systems choose a position deliberately rather than inheriting one from a demo.

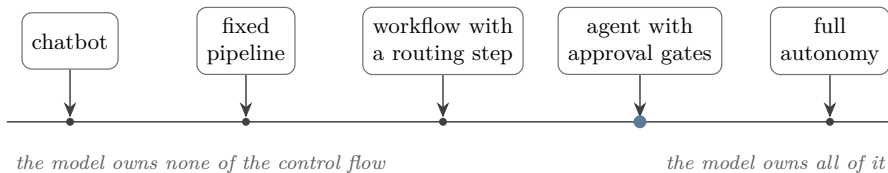


Figure 1.3: Autonomy is a dial, not a switch. The quantity that varies along it is how much of the control flow the model owns. The labels are stops worth naming; real systems usually sit between them.

One more observation before we leave the taxonomy, because it forecasts where this book ends up. Almost nothing that ships is a pure workflow or a pure agent. The dominant production shape is a workflow shell with one or two genuinely agentic steps inside it: predictable structure wherever you can predict, model-owned autonomy only at the junctures you cannot enumerate. You are setting a dial, step by step, and the next section is about how far to turn it.

1.3 Do You Even Need an Agent?

Now that you can tell an agent from a workflow, you face the question this section exists to make respectable: do you need one at all? I put it this early on purpose. The mistake this field warns its newcomers about most often is reaching for an autonomous agent when a single prompt, a fixed pipeline, or fifty lines of ordinary code would have been cheaper, faster, and easier to debug. An agent demo is a seductive thing. You watch a model plan, act, recover, and finish, and the natural conclusion is that this power belongs everywhere. Resist the conclusion long enough to price it.

Compared with a single model call, an agent adds several recurring costs, and they arrive as a package. Money, first: an agent makes many model calls per task instead of one, and each call re-sends the growing conversation, so cost scales twice over—once in the number of calls, again in the size of each. A ten-step loop can cost an order of magnitude more than the single call it replaced (the multiplier is illustrative; the direction is what matters). Latency, second: the calls are sequential, each waiting on the last tool result, so a sub-second answer becomes many seconds or minutes; if a human is waiting on the other end, this alone can disqualify the design. Nondeterminism, third: the same input can take a different path and produce a different answer on each run, which quietly breaks your habits for testing, debugging, and auditing.⁴

⁴One published analysis of agent reliability reports single-run success rates around 60%

And the failure surface changes in kind, which is the cost I most want you to feel: the bounded failure the last section credited to the chatbot is exactly what an agent gives up. Its mistake is a wrong *action*, followed by ten more actions built on top of it, compounding down the loop exactly as the compass arithmetic predicts.

Two further costs are easy to miss when the demo is glowing. Granting a model tools and autonomy widens your attack surface; the security community has a name for over-granting it: *excessive agency*, meaning more permissions, functionality, or freedom than the task requires.⁵ The danger also extends past what you grant deliberately: text an agent merely *reads* can steer what it does next, a family of attacks called *prompt injection* that Chapter 17 examines. Finally, the operational burden: a chatbot needs a prompt, while an agent additionally needs tool plumbing, context management, error recovery, stop conditions, approval gates, tracing, and evaluations. That surrounding machinery is called the *harness*, and building it is most of the work; a brilliant model in a flimsy harness fails in production.

So the honest default is a ladder, which is the compass's fourth bearing made practical, and the discipline is to stay on the lowest rung that solves your problem. The bottom rung is plain code: a regular expression, a SQL query, an *if* statement—free, instant, deterministic, auditable; if the logic is fully specifiable, no model is required. The next rung is a single well-crafted model call, perhaps with retrieval and a few examples in the prompt; a surprising fraction of “we need an agent” dissolves into “we needed a better prompt.” The rung above that is a workflow: several calls on paths you drew in advance. The top rung is an agent. Each rung up buys adaptability and pays for it in money, latency, and predictability. For a problem whose steps you can draw in advance, a lower rung is the correct answer permanently; staying there is the discipline. Figure 1.4 draws the ladder, with adaptability rising as you climb and cost rising with it.

The working test is the one from the previous section, now used as a decision aid: can you draw the flowchart before the request arrives? If yes, stop at a workflow or below—known steps, auditable correctness requirements, tight latency or cost budgets, and simple lookup-classify-transform tasks all point down the ladder. What points up the ladder is the genuinely open-ended goal: the steps depend on what the model discovers, and judgment is required at branches you cannot enumerate. Even then, insist on the compass's first bearing in concrete form—a clear success criterion, a feedback signal at each step, and a sensible place for a human to look. A concrete pair to calibrate on: “route each incoming email to billing, technical, or sales and draft a first reply” has knowable steps and high volume, and wants a workflow; “make this

falling to roughly 25% when the same task must succeed on eight consecutive runs—figures illustrative of the consistency gap, not permanent facts. Fiddler AI, “AI Agent Failure Rate: Why 70–95% Fail in Production.”

⁵OWASP, “Top 10 for LLM Applications”—the entry on Excessive Agency.

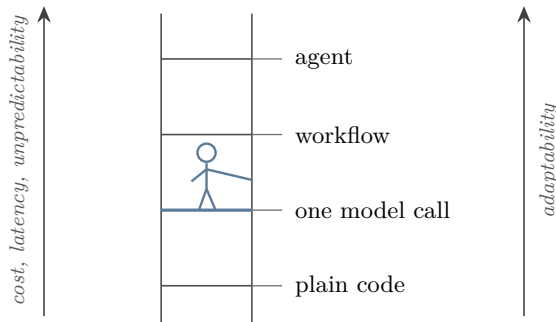


Figure 1.4: The escalation ladder. Each rung up—from plain code to a single model call to a workflow to an agent—buys adaptability and pays for it in cost, latency, and unpredictability. The discipline is to stand on the lowest rung that solves the problem (in accent), and to climb only when a real case proves the rung below cannot reach.

failing test pass in a codebase you have never seen” has unknowable steps and a built-in verifier, and is a genuine agent problem.

One trap deserves its own paragraph, because it argues so eloquently for the wrong design: “it might need to adapt someday.” That is a hypothesis, and cheap to test honestly: build the simpler version first, measure it against real cases, and climb the ladder only when you can point at a class of inputs the simple system provably fails on and confirm that an agent actually does better. The reverse trap is *agent-washing*: because the word sells, scripted pipelines get relabeled as agents. The question that cuts through both is the litmus test you already own, asked about the system in front of you, whatever its label says: does the model decide the next step, or does code? Chapter 14 returns to this decision with the benefit of everything in between; Chapter 16 shows how to do the measuring.

An agent, then, is a cost you pay for adaptability you can name. If you cannot name the adaptability, keep your money. And if you can—if the steps genuinely cannot be drawn in advance and a verifying signal exists—then an agent is the right tool, and the next several chapters are about building one well. First, though, you need to understand the engine we will be looping: Chapter 2 is about what a language model actually is.

Chapter 2

The Engine: How Language Models Work

Every system in this book, from the plainest chatbot to a fleet of cooperating agents, is built around the same component: a large language model. You can use one productively while knowing nothing about its insides, the way you can drive a car without ever opening the hood. Building an agent is a different relationship with the machine. You are going to put this component in a loop, hand it tools, and let it make decisions while you are elsewhere; when it behaves strangely, “it’s magic” will not narrow down the bug. This chapter builds the minimum mental model an agent builder needs, intuition first, with no mathematics beyond arithmetic. We start with what the model fundamentally is (a next-token predictor), then take up its working memory (tokens and the context window), the way it writes (sampling, and why two runs differ), the way you steer it (prompting), the bridge from its text to real action (structured output and function calling), and finally the ways it reliably fails. Later chapters lean on every one of these ideas; none of them requires you to train anything.

2.1 LLMs as Next-Token Predictors

Finish this sentence: “The capital of France is —.” The word was in your mind before you decided to look for it. You did not consult a mental atlas, weigh candidate cities, or reason about geography; a lifetime of reading made *Paris* simply arrive. Hold on to that reflex, because it is the most accurate intuition available for what a large language model does. The model completes text the way you just did; unlike you, it has no other mode.

Mechanically, an LLM is a function: it takes in a stretch of text and produces, for every possible next *token*, a probability that it comes next. A

token is a chunk of text from a fixed vocabulary—often a word, sometimes a piece of one or a punctuation mark; on average, think three-quarters of an English word. To write anything, the surrounding system picks one token from that ranked list, appends it to the text, and runs the model again on the now-longer text, one token at a time, over and over, until a stop condition fires. The scheme is called *autoregressive generation* (each new token is conditioned on everything before it, prompt and output alike), and one property of it will follow us through the whole book: the model never drafts its answer in advance. Its whole method is a local guess, repeated; it holds no plan and no outline in reserve (Figure 2.1). The next section examines tokens and the model’s working memory in earnest; the one after looks at how a token gets picked and why the choice varies.

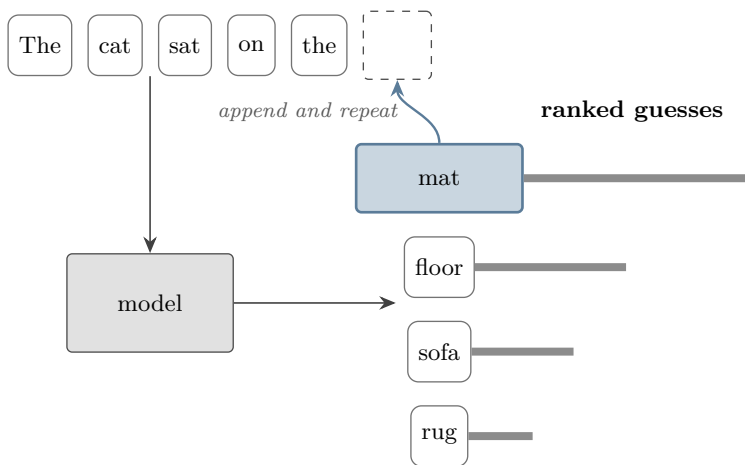


Figure 2.1: Autoregressive generation, the engine’s whole method. The text so far enters the model, which returns a ranked list of next-token guesses; one is chosen, appended to the text, and the model runs again on the now-longer input. The chosen token and its return path are in accent—there is no plan held in reserve, only this local guess repeated.

Where does the guessing skill live? In the model’s *parameters*, also called *weights*—billions of numbers, fixed during training, that determine how input text flows through the neural network to produce those probabilities. The picture I find most useful is this: the weights are a lossy, compressed distillation of the patterns in everything the model read during training—grammar, facts, idioms, code style, the shapes of arguments. “Lossy” is the operative word. The model absorbed the shape of its reading and kept no archive of the text itself, which is why asking it to quote a source verbatim, or to recall a precise figure, is a gamble; we return to the consequences at the end of this chapter.

Two activities need to be kept firmly apart, because conflating them produces some of the most common misunderstandings about these systems. *Training* is the one-time, enormously expensive process of finding good values for the weights: the model is shown text, asked to predict the next token, and nudged slightly whenever it is wrong, across more text than any person could read in a thousand lifetimes. It is carried out by the model’s makers, on specialized hardware, before you ever see the thing. *Inference* is every use afterward: the weights are frozen, and the model simply runs your input through them. Everything you do with a model—every chat, every agent step—is inference. Two consequences follow, and you will design around both constantly. First, the model does not learn from your conversations; nothing you type changes a single weight, and what feels like memory within a session is only the growing transcript being re-read on every call. Second, the model’s built-in knowledge stops at a *training cutoff* (the point after which it saw no data). Events past the cutoff are simply absent, and the model will not reliably announce the gap; it answers from stale knowledge with exactly the confidence it brings to fresh. Anything time-sensitive must arrive in the prompt, or be fetched by a tool.

Training itself happens in stages, and the stages explain a puzzle you would otherwise trip over. The first and costliest stage, *pretraining*, is pure next-token prediction over a vast, broad corpus—web pages, books, code. Its product is a *base model*: fluent, widely knowledgeable, and strangely useless as an assistant. Chapter 1 mentioned, in passing, that such a model may answer a question with more questions, and promised that this chapter would show why. You now hold every piece of the explanation; assemble it before reading on. A base model only continues text, so when you hand it “What are three ways to make this query faster?”, it looks for a plausible continuation—and in the wild text it studied, one question is often followed by others like it. A list of further questions is a perfectly likely reply. The base model is doing its job perfectly; the job is continuation.

Post-training is the further stage that turns that raw continuer into the assistant you have actually met, and it is what Chapter 1’s “models learned to take instructions” referred to. Two moves matter for our purposes. *Instruction tuning* trains the model on curated pairs of instruction and good response, teaching it that a question deserves an answer rather than more questions. *Preference tuning*—the best-known form goes by *RLHF*, reinforcement learning from human feedback—has people (or models trained to mimic their judgment) rank competing answers, and adjusts the weights toward the kinds of answers people prefer: more helpful, more honest, better mannered. Post-training is where the familiar assistant persona comes from, and it is a comparatively cheap coat of paint on the enormous pretrained core.

Be equally clear about what post-training does not buy. It adds no new knowledge, and it does not remove the tendency to fabricate. A memorable framing from one widely watched lecture on the subject: post-training does

not stop the model from dreaming; it shapes the dreams into helpful-assistant-shaped dreams.¹ Underneath the manners, the engine is unchanged: a next-token predictor, optimizing for the plausible continuation. Plausible and true are correlated. The correlation exists because the training text was mostly written by people describing a real world—but the two part company exactly where the model lacks the fact, and at that moment the machinery produces the most plausible-looking string instead of stopping. Fluent first, correct second.

For an agent builder, the corollary I most want you to keep is that the model’s knowledge lives in two places of unequal reliability. *Parametric* knowledge sits in the weights: frozen at the cutoff, approximate, impossible to cite. *Context* knowledge is whatever text is in front of the model right now: fresh, inspectable, checkable. The model is markedly more reliable when the facts it needs are in its context than when it must dredge them from its weights, and that asymmetry seeds half the engineering in this book. Retrieval, tools, and the whole discipline of context engineering in Part III are, at bottom, ways of moving load-bearing facts out of the first place and into the second.

So this is the engine: superb at language-shaped work—drafting, rewriting, summarizing, translating, explaining, transforming code—and unreliable as a database or a calculator, with a confidence dial that is permanently stuck on high. An agent will ask this engine to make decisions, many in a row, about a world it can only see through the keyhole of its context. The keyhole is where we go next.

2.2 Tokens and the Context Window

Everything the model knows in the moment arrives through that keyhole, so we had better measure it. Chapter 1 needed little more than a definition of the *context window*: the bounded stretch of text a model can consider in a single call. This section fills in the picture, and the place to start is the unit the window is measured in, because nearly every limit you will hit while building—the ceiling on what the model can see, the bill at the end of the month, the pause before an answer begins—is denominated in tokens. The previous section offered a napkin version: a token is a chunk of text, on average about three-quarters of an English word.² The napkin version hides a real design problem, and the solution to that problem explains several behaviors that would otherwise look like plain stupidity.

The design problem is this. A model needs a fixed vocabulary: a finite

¹The “dreaming” framing is Andrej Karpathy’s, from his widely circulated talks and commentary on large language models (e.g. “Intro to Large Language Models”); it is rendered here as a paraphrase, not a verbatim quote.

²The three-quarters-of-a-word rule (equivalently, about four characters per token in English) comes from OpenAI’s help-center note “What are tokens and how to count them?”; it is a rough average for English prose, not a constant.

menu of chunks out of which every possible input and output will be assembled. Suppose the menu were whole words. The vocabulary balloons—**love**, **loves**, **loved**, **lovingly** each need their own entry—and the first typo, invented name, or word coined after training has no entry at all. Suppose instead the menu were single characters. Now nothing is unrepresentable and the vocabulary is tiny, but every text becomes an enormously long sequence of nearly meaningless units. Production models settle on the compromise between the two, called *subword tokenization*: a fixed vocabulary of chunks, far larger than an alphabet and far smaller than a dictionary of every word form, in which common words travel as single tokens while rarer ones are assembled from pieces, so that **annoyingly** might become **annoying** + **ly**, and **unhappiness** might become **un** + **happi** + **ness**.³ Frequent strings are represented cheaply; any string at all can be represented by falling back to smaller pieces.

The machine that does the chopping, the *tokenizer*, deserves a short paragraph, because it is far simpler than people assume. It is a fixed, deterministic lookup table, built once alongside the model and frozen for the model’s whole life. On the way in, it converts your text into a list of integer IDs; on the way out, it converts the model’s chosen IDs back into text. Notice what this implies: the network never sees your characters. From where the model sits, the world is a sequence of numbered chunks, and whatever happens *inside* a chunk was erased before the model ever got involved.

That erasure is testable, and I recommend running the test yourself. Ask a model how many times the letter **r** appears in some ordinary word, or ask it to spell a word backwards. Models have stumbled over exactly such questions for years, sometimes comically, in the same conversation where they solve a genuinely hard problem. The pattern stops looking absurd the moment you hold the tokenizer in mind: the word arrived as one or two opaque chunks, and the letters inside a chunk are precisely the information the lookup table threw away. To count them, the model must reconstruct the spelling from whatever it absorbed about that word during training—a memory task, and an unreliable one. (By the time you read this, the model you try may well pass; such gaps get patched by training, or the model quietly reaches for a tool. The mechanism the failure exposes is still underneath.) Digits tell the same story: in at least one widely used tokenizer, 520 is a single token while 521 splits into 5 + 21,⁴ so the model does arithmetic over chunks that

³Hugging Face’s tokenizer documentation (“Summary of the tokenizers”) describes the standard subword algorithms—BPE, WordPiece, Unigram—and illustrates them with splits like these. For a hands-on deep dive, Andrej Karpathy’s lecture “Let’s Build the GPT Tokenizer” constructs one from scratch and tours the odd behaviors it causes.

⁴Which particular numbers split is an arbitrary fact of each tokenizer’s training. In the byte-pair tokenizer of the GPT-2/GPT-3 generation, for instance, 520 is one token while 521 becomes 5 + 21; later tokenizers tend to keep short numbers whole but still chunk longer ones irregularly. The point is that some numbers split, and the model cannot see inside a chunk.

ignore the number system—one reason its mental arithmetic is shakier than its fluency suggests. The lesson generalizes: the model’s world has a grain, and the grain is the token.

Three consequences of tokenization matter to a builder more than its internals. First, the token-per-word ratio drifts with content: code, numbers, and structured data tokenize less efficiently than prose, so a page of JSON costs more tokens than a page of essay. Second, most tokenizers were trained predominantly on English, so the same meaning can cost several times as many tokens in another language—one analysis found Burmese running at roughly eight times the token count of equivalent English, a figure I cite as an illustration of the disparity, not a constant⁵—which has plain cost and fairness consequences if your users write in many languages. Third, distinctions you would never think about are real to the tokenizer: the same word with and without a leading space, or capitalized and not, can be entirely different tokens. The practical habit that follows from all three: when a count matters, measure your actual text with the provider’s own tokenizer or counting endpoint. Every model family chops text differently, so a count borrowed from one is only an estimate for another. (I notice that nearly every number in this section arrives wrapped in “roughly.” With tokenizers, roughly is the honest register.)

Now the window itself. The picture I will use for it, here and for the rest of the book, is a desk: the work surface on the model’s side of the keyhole, on which everything the model consults during a single call must fit at once. Spread across that desk, in a typical agent call, are your standing instructions, the whole conversation so far, the definitions of every tool the agent might use, the full text of every tool result that has come back, whatever documents were fetched for the task—and the answer the model is writing, which lands on the same desk as it grows. One token budget covers them all. There is no drawer, no shelf, no second desk. If a fact is on the desk during this call, the model can use it; if it is anywhere else in the universe, then for the duration of this call it does not exist.

The desk is also swept bare between calls, and here the metaphor needs its honest boundary: your working memory carries into the afternoon, while the model’s does not survive to the next request. An LLM API is *stateless* (each call starts from nothing), which gives the previous section’s point about memory its operational shape: to continue a conversation, the surrounding software lays the entire transcript back on the desk every turn, the way you might reread a whole email thread before replying to its newest message. The habit is harmless at three messages and ruinous at three hundred. A

⁵Aleksandar Petrov et al., “Language Model Tokenizers Introduce Unfairness Between Languages,” NeurIPS 2023 (arXiv:2305.15425), which measures disparities up to roughly fifteen times across a range of tokenizers; the roughly-8× figure for Burmese is a conservative illustration of how wide the gap can get for languages underrepresented in tokenizer training.

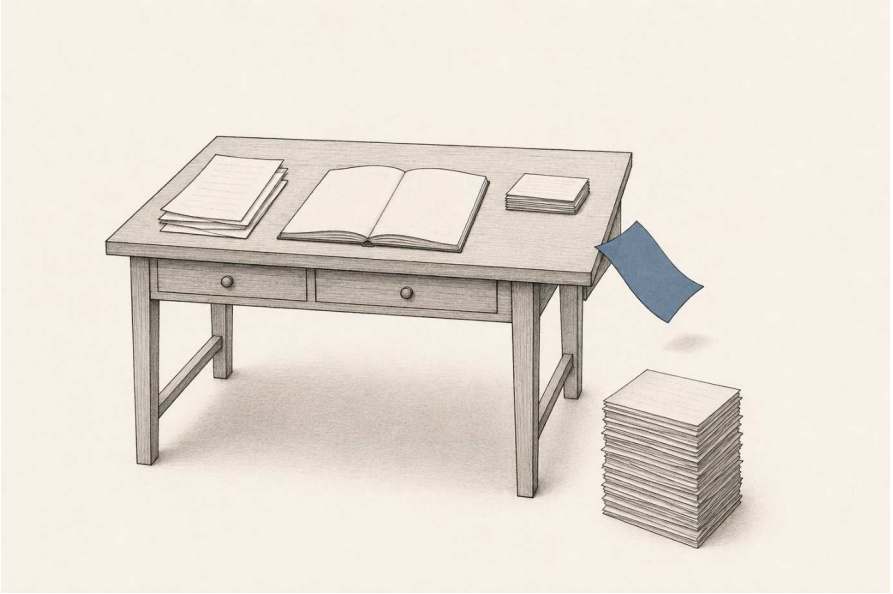


Figure 2.2: The context window as a single finite desk: everything the model can use at this moment must physically fit on its surface.

sketch, with made-up but realistic numbers: standing instructions, 500 tokens; conversation so far, 8,000; two fetched documents, 40,000; the latest tool result, 3,000. The model reads more than fifty thousand tokens before writing a word, you pay for every one of them, and next turn the pile is larger still, because this turn has joined the history. (Providers soften the cost of re-reading an unchanged pile with caching, one of the main levers of Chapter 19.)

What happens when the pile no longer fits? Nothing graceful. Depending on the API and the software in between, the request may be rejected outright with an error; or the input may be truncated—cut down until it fits, usually by dropping the oldest turns, often without telling anyone; or a framework in the middle may summarize the history before sending it. The outright error is the kind outcome. Silent truncation is the treacherous one: the call succeeds, the answer comes back fluent, and the model has been reasoning over an amputated transcript from which your original instructions may have quietly fallen. From the outside this looks like a model that got dumber overnight, and it burns debugging hours accordingly. When a long session starts “forgetting,” check for overflow before you blame the model. (A cousin of this surprise: input and output share the window, but many models separately cap a single response at a much smaller figure. An answer that stops mid-sentence usually hit the output cap.)

One more property of the desk, and Chapter 1 already let it slip: even

what fits is used unevenly. On long inputs, models recall material near the beginning and near the end markedly better than material buried in the middle—the compass called such a fact half-forgotten, and the research literature calls the effect, aptly, “lost in the middle.”⁶ A bigger desk therefore solves less than it appears to: the edges stay sharp while the middle grows. Here is the mechanical basis of the book’s second bearing, the scarcity of context, and the craft it forces—deciding what deserves the desk at all—fills Part III, beginning with Chapter 7.

Even when everything fits and sits where recall is strong, the pile is never free. Tokens are the unit of your bill in both directions, input and output. They are the unit of your wait, too: before the model can emit the first token of its answer it must read the entire pile—a phase called *prefill*—and in the standard architecture the cost of that reading grows steeply with length, roughly on the order of its square. A bloated context makes the agent slower to start answering and more expensive on every single call, whether or not the extra material helps. So the window is a ceiling, not a target; the useful question is the smallest desk that actually holds your task.

For a chatbot, these are background facts. For an agent, they are daily weather: the loop appends to the desk on every step—reasoning, tool call, tool result, again and again—and nobody curates the pile unless you build the curation. Twenty steps in, the transcript is heavy with stale directory listings and half-relevant page dumps, the instruction that actually matters sits mid-pile where recall sags, and each further step reads, and bills, the whole accumulation. The desk is now measured. What remains of the mechanics is the writing itself: how the model picks one token from its ranked guesses, and why two identical runs of your agent will not behave identically. That is next.

2.3 How Text Is Generated

Hand the model the fragment “The cat sat on the” and, as the opening of this chapter described, back comes a ranked list: a probability for every token in the vocabulary. Say the top of that list reads **mat** at 0.42, **floor** at 0.18, **sofa** at 0.13, **rug** at 0.09, with the last scraps of probability scattered across thousands of also-rans. (The numbers are invented; the shape is typical.) Now the question this section turns on, and I would like you to commit to an answer before reading further: which word does the model write? The answer that feels obvious is **mat**—take the best guess; why would a machine do anything else? Yet for most systems you will ever call, that answer is wrong. By default, the machinery rolls dice.

⁶Nelson F. Liu et al., “Lost in the Middle: How Language Models Use Long Contexts,” *Transactions of the Association for Computational Linguistics* (2024). Accuracy on long-context tasks was highest when the relevant passage sat at the start or end of the input and sagged when it sat in the middle.

The step in question—turning a ranked list into one chosen token—is called *decoding*, and the obvious strategy has a name: *greedy decoding*, always take the single most probable token. Greedy is deterministic, cheap, and sounds like the responsible choice. It also produces text with a well-documented ailment: flat, oddly repetitive prose that can lock into loops, the same phrase surfacing again and again.⁷ The reason is counterintuitive enough to spell out. Human writing keeps taking small risks—an unexpected adjective, a fresh turn of phrase—while a decoder that never gambles drifts into the blandest corner of the language and then circles it. The locally safest word, chosen every time, composes into a globally mediocre passage.

The alternative, and the default nearly everywhere, is *sampling*: draw the next token at random, weighted by the model’s probabilities. **mat** wins the draw 42 times in a hundred; **rug** gets its turn about nine times in a hundred. The picture to keep is a *weighted lottery*, run once per token: the model prints the odds, and the decoder draws a ticket. Every fluent, natural-sounding answer you have admired was assembled this way, one draw at a time. And a property of your future agents is being fixed here, before any agent exists: variability is designed in. Two identical requests can draw different tickets, and the moment one draw differs, the two texts part ways for good, because every later token is conditioned on the earlier ones.

Between greedy and wild there are knobs, and they all do one thing: reshape or trim the ranked list before the draw. All of them operate downstream of the prediction: the ranked list is fixed, and the knobs govern only how you pick from it. *Temperature* is the sharpness dial. Turned down, it widens the gap between favorites and long shots until, at zero, the top token wins every draw and you are back to greedy; turned up, it flattens the field, gives the long shots real chances, and eventually dissolves the text into noise. Notice what the dial trades: low temperature buys consistency, high temperature buys variety, and correctness sits at neither end—a wrong answer can be drawn confidently at any setting.

The other two common knobs trim the field instead of reshaping it. *Top- k* keeps only the k highest-probability tokens and discards the rest—simple, but k is a fixed pool size, and the number of genuinely plausible continuations varies wildly from one moment to the next. *Top- p* (also called *nucleus sampling*) repairs that by keeping the smallest set of top tokens whose probabilities add up to at least p : after “The capital of France is,” the pool may hold a single token; after “My favorite food is,” it may hold dozens. The pool sizes itself to the model’s own confidence, which is why top- p is a common default. The practical guidance is short. Choose the setting by task—low temperature for anything a program will parse or a test will check, higher for brainstorming and drafting—and adjust one knob at a time, since several of

⁷Ari Holtzman et al., “The Curious Case of Neural Text Degeneration,” ICLR 2020. The paper documented how maximization-style decoding degenerates into repetition and introduced nucleus (top- p) sampling as a remedy.

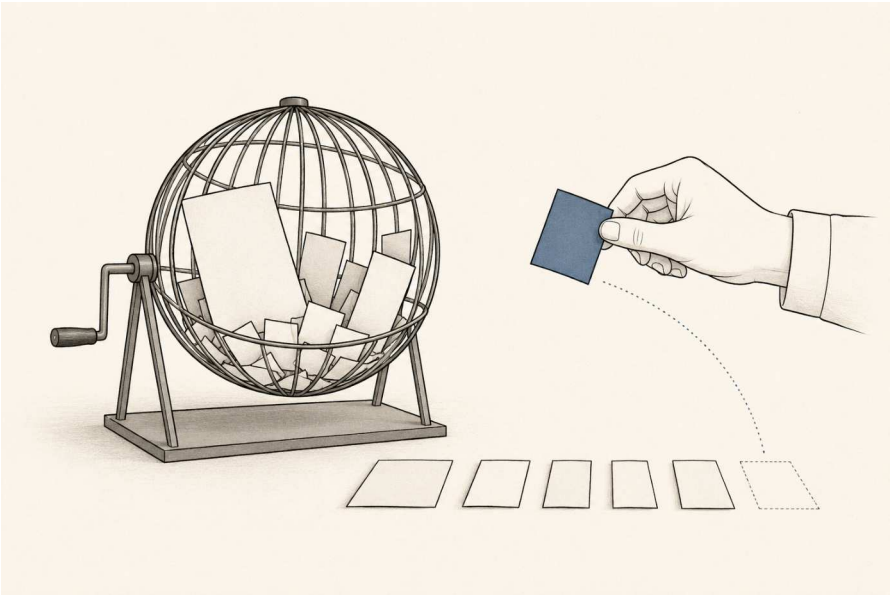


Figure 2.3: Sampling as a weighted lottery run once per token. The model prints the odds and the decoder draws a ticket; the favorite usually wins, but not always, and each draw takes its place in the sentence so far.

them squeeze the same distribution.

Generation also has to end, and it ends in one of three ways. The model can emit a special end-of-sequence token—training taught it to say “I am finished” in vocabulary form—which is the happy path. A hard maximum-length cap can cut the output off mid-thought; the previous section met this as the answer that stops mid-sentence. Or a caller-supplied *stop sequence* (a string you designate in advance, often a delimiter) ends generation the moment it is produced. One habit to build now: APIs generally report which of these ended the response. Read that field. A reply truncated by the length cap looks exactly like a model failure, and is actually a budget you set.

Now the question the previous section left hanging: why two identical runs of your agent will differ. The first source of variation is in plain sight. Sampling is a lottery, so any temperature above zero means the same prompt can draw different tickets, fork, and never rejoin. This source is intentional and fully under your control: set temperature to zero, and the draw becomes deterministic. In theory.

Here is a prediction exercise for the intuition you now have. Suppose one prompt is sent to a hosted model a thousand times at temperature zero (greedy decoding, no lottery). How many distinct answers should come back? The arithmetic in your head says one. When a research team actually ran this experiment, they got eighty. All one thousand completions were identical for the first 102 tokens; at token 103—the sentence had reached a biographical subject’s birthplace—992 of them phrased it one way, 8 another, and from that fork the transcripts spread.⁸ The popular explanation blames rounding error on parallel hardware, and it is incomplete. The fuller account concerns a property called *batch invariance*, or rather its absence. A hosted endpoint serves many customers at once and batches their requests together for efficiency; the size of the batch rises and falls with traffic; and the low-level numeric routines inside the model can return minutely different results at different batch sizes. Minute is enough. When two tokens sit nearly tied at the top of the ranked list, a wobble in, say, the sixth decimal place decides the winner, greedy faithfully picks it, and the text forks. From your side of the API, the deciding input is how many strangers happened to be querying the same servers at the same instant—an input you cannot see, control, or replay. Batch-invariant versions of those routines exist and do fix this, at a cost in serving speed; at the time of writing they are the exception. So treat temperature zero as a strong preference, honored almost always—and remember that “almost always,” rolled at every token of every step, leaves room for surprises.

⁸Horace He et al., “Defeating Nondeterminism in LLM Inference,” Thinking Machines Lab (2025). The experiment: 1,000 temperature-zero completions of “Tell me about Richard Feynman” produced 80 unique outputs, identical through token 102 and first diverging on the birthplace phrase (“Queens, New York” versus “New York City”). The figures illustrate the phenomenon, not a constant of nature.

What should a builder do with all this? Three habits. First, give up exact-string reproducibility as a foundation: write tests that assert properties of the output—it parses, it names the right customer, it passes the checker—because a test that asserts byte-for-byte equality will flake in ways that waste your weeks. Second, expect the failure you saw once to resist reproduction on demand; that is a fact about the infrastructure, and Chapter 15 builds the recording tools that make such failures debuggable anyway. Third, feel the agent-shaped consequence: an agent runs this lottery at every token of every step, and each step’s output becomes the next step’s input, so run-to-run variation compounds along the loop—same task, different path, both plausible. The response to this that has aged well in the field is to make the outcome checkable rather than the output identical: the verifiability bearing from Chapter 1, arriving now with its mechanical justification.

So the machine writes by lottery, and the lottery explains the variety. The next question is steering. If the weights are frozen and the dice are indifferent, the only lever left is the text you lay on the desk—what to say, in what order, and what to show by example. That lever is the next section.

2.4 Prompting Fundamentals

Count the levers you actually hold over this machine. The weights are frozen; nothing you do at inference time will change them. The sampling knobs of the previous section adjust how a token is drawn from the ranked list, never what the model predicted in the first place. Everything else comes down to a single instrument: what you choose to write. That text is the *prompt*, and the craft of writing it well—*prompt engineering*, a grand name for a practice that is part writing and part experiment—is the basic steering skill on which everything agentic rests. Its fundamentals fit in this one section, and they are worth the reread even if you have prompted models for years: an agent assembles a fresh prompt at every step of its loop, so a weakness here is a weakness repeated throughout everything you build later.

Start with the anatomy. Modern chat models take a list of *messages*, each tagged with a *role*, and three roles are nearly universal (their exact names and wire formats vary by provider; treat any specific syntax as a vendor detail). The *system* message carries the standing instructions: who the model is supposed to be, what it may and may not do, and the shape its answers must take. It is written once and persists across every turn (the surrounding software resends it with every call—the model itself, remember, retains nothing). The *user* message carries the ask of the moment—this question, this ticket, this file. And *assistant* messages are the model’s own replies, which accumulate as the conversation’s history; usefully, you can also write assistant messages yourself, planting words in the model’s mouth as a way of showing, rather than describing, what a good answer looks like. My

sorting rule: if an instruction would have to be repeated on every single turn, it belongs in the system message; if it is specific to this one request, it belongs in the user message. Mixing these up costs you later—a rule restated in each user message survives only as long as every caller remembers to restate it. Figure 2.4 lays out the three roles and the statelessness that resends the whole stack on every call.

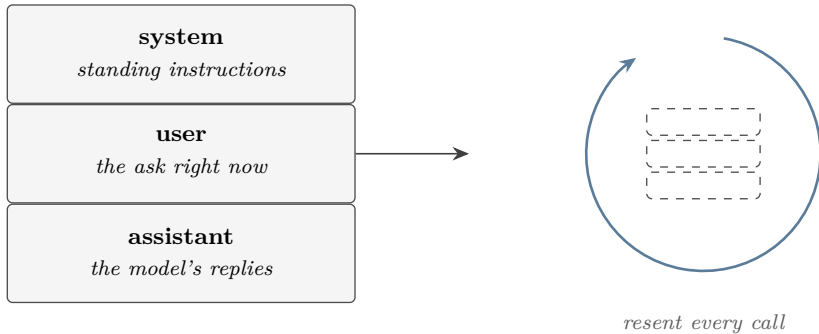


Figure 2.4: The anatomy of a chat prompt. Standing instructions go in the *system* message, the current request in the *user* message, and the model’s own answers accumulate as *assistant* messages. Because the model retains nothing between calls, the surrounding software resends the entire stack on every turn (the accent loop)—what feels like memory is only the transcript being re-read.

The single biggest lever is clarity, and after two sections of mechanics you can see why it works. The model continues text; a vague instruction leaves the ranked list broad, and the lottery happily roams whatever space you leave open. A precise instruction narrows the field before the dice roll. In practice this means preferring enforceable constraints to adjectives: “three to five sentences” beats “keep it short,” and “at most five bullets, each under twelve words” beats “be concise,” because the first of each pair can be checked and the second is a mood. It means stating what you do *not* want, since the model’s trained helpfulness tends toward overreach: “do not invent prices”; “if the customer asks for a refund, do not promise one.” And it means providing a fallback for the case where the task cannot be done: “if the answer is not in the provided text, say you do not have enough information.” Without that last line, a model that lacks the answer will do what it was trained to do, which is answer.

Here is the anatomy assembled, small enough to read at a glance:

```
You are a support-ticket triage assistant.
Classify the ticket below into exactly one of:
    billing, bug, feature-request, other.
Ticket: "The app crashes every time I open settings."
```

Respond with the single category word and nothing else.

Five lines, four jobs: a role that sets scope, an instruction with a closed set of allowed answers, the input data, and an output contract. The phrases doing the real work are the strict ones—“exactly one of,” the fixed list, “nothing else.” Run this and a capable model returns **bug** nearly every time.

Nearly. Suppose one run in twenty comes back as “Bug.” with a capital letter and a period, or as “This sounds like a bug in the settings screen.” Your parser chokes, and staring harder at the instruction reveals no flaw in it. The standard escalation is to add *examples*:

```
Ticket: "You charged me twice this month." -> billing
Ticket: "Can you add dark mode?"          -> feature-request
Ticket: "The app crashes every time
       I open settings."                  ->
```

Prompting with instructions alone is called *zero-shot*; adding worked demonstrations like these is *few-shot*; and the model’s knack for picking up the pattern has its own name, *in-context learning*—the model adapts to your task from examples inside the prompt, with no retraining of any kind. What the examples teach is what the failure above shows was missing (format and register), plus a third thing instructions describe poorly: edge-case handling. There is an instructive research result here: much of the benefit of demonstrations comes from their *structure*—a consistent format, a visible set of labels—and survives even when some of the individual labels are wrong.⁹ The examples work as a stencil for the answer’s shape. Two or three well-chosen ones capture most of the available gain; each additional example costs tokens on every call, in an agent’s case at every step, for shrinking returns.

Know also what demonstrations will not buy: a task the model reasons about wrongly tends to stay wrong however many finished answers precede it. Demonstrating the *reasoning* itself is a different device, and it gets its own treatment in Chapter 4. The escalation path that falls out of all this is pleasantly simple: start zero-shot; add two or three examples when the output’s shape wobbles; look elsewhere when the problem is the content.

Output format deserves its own explicit instruction whenever anything downstream depends on it. For prose meant for humans, name the sections, their order, and the length bounds. For output a program will parse, say the shape plainly—and understand that an instruction is a request, not a guarantee. The model can honor “respond with the single category word and nothing else” a thousand times and then, on run one thousand and one, wrap

⁹Sewon Min et al., “Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?,” EMNLP 2022. Demonstrations with randomly substituted labels retained much of the benefit of correctly labeled ones; the format and the label space carried a large share of the effect.

the word in a courteous sentence. Where parsing matters, the request can be upgraded to a contract, and that upgrade is the subject of the next section.

A word on *personas*, the most folkloric corner of prompting. Telling the model who to be—“you are a senior security reviewer,” “you write like a field guide”—reliably shapes voice, register, and scope, and for open-ended writing that is often exactly the lever you want. The folklore goes further and says a persona also makes answers more *accurate*: tell it to be an expert and it becomes one. The research picture is much less flattering: on factual and analytical tasks, simple personas show mixed and often negligible effects on capable models.¹⁰ Use personas to set voice and boundaries; use instructions, examples, and genuine reasoning support for correctness.

Two placement habits complete the toolkit. First, when a task has several parts, spell them out as ordered steps—“first list the company names, then the people, then the recurring themes”—instead of gesturing at the whole (“extract the entities”). Explicit steps improve the output and, just as valuable, tell you *which* step failed when one does; you will meet this instinct again, grown up, when agents decompose whole projects in Chapter 4. Second, position matters. When a prompt opens with pages of material—a document, a transcript, a pile of tool results—put the instruction *after* the pile, and restate the one load-bearing rule near the end. Both halves of the reason are already yours: the model is a continuer, so a prompt that ends mid-document invites a continuation of the document rather than the task; and the desk’s edges stay sharp while its middle sags, so the final lines sit where recall is strongest. The last thing the model reads should be the thing you most need it to do.

I will not pretend any of this adds up to a science. Prompts are empirical objects: two phrasings you regard as identical can behave differently, and a prompt that works today can drift when the model behind it is upgraded. Treat prompts the way you treat code—versioned, tested, re-tested after every model change; Chapter 16 builds the apparatus. And hold on to the reframing this section has been steering toward. In an agent, no one types the final prompt. The system message is written once; the tool definitions are injected by machinery; the history accumulates turn by turn; retrieved documents and tool results are slotted in at runtime by code. Every principle above still applies—someone still chooses what the model reads—except that the someone becomes a program you write. Deciding, step after step, what earns a place on the desk is a large enough problem to carry its own name, *context engineering*, and Part III of this book is devoted to it. But Part III is a long way off, and the nearer goal is at hand: an instruction can only *request* a

¹⁰Mingqian Zheng et al., “When ‘A Helpful Assistant’ Is Not Really Helpful: Personas in System Prompts Do Not Improve Performances of Large Language Models,” Findings of EMNLP 2024; PromptHub’s survey “Role-Prompting: Does Adding Personas to Your Prompts Really Make a Difference?” collects this and related studies. Elaborate, domain-matched personas have shown gains in some settings; a bare “you are an expert” generally has not.

format. The next section shows how to guarantee one.

2.5 Structured Output and Function Calling

Chapter 1 called function calling the bridge from text to action and promised that this chapter would supply the mechanism. This section keeps the promise, and the place to start is the question of who reads the model’s output. As long as a human reads it, phrasing can wander freely; people forgive. The moment a *program* reads it, wandering breaks things—and everything an agent does depends on programs reading model output.

Feel the problem before we solve it. Suppose your code asks the model to pull two facts out of a customer email—the customer’s name and the order total—so it can write them into a database. Three runs, three answers, every one of them correct as prose:

Name: Ada, Total: \$42

The customer is Ada Lovelace and she spent forty-two dollars.

Happy to help! Ada's order total is \$42.00 (invoice #1041).

Before reading on, sketch the parsing code that handles all three. Whatever you wrote for the first breaks on the second; whatever handles the second trips on the third’s parenthesis; and tomorrow brings a fourth phrasing, because the lottery of two sections ago owes you no consistency. That is the root problem: natural language has enormous surface variability, and nothing obliges the model to phrase the same fact the same way twice. A program cannot consume possibilities. It needs a *contract*—a fixed shape the output is guaranteed to fit.

The standard contract is *JSON* (a simple, universal text format for nested key–value data), with its shape pinned down by a *schema*: a declaration of which fields exist, what type each holds, which are required, and what values are allowed. For the extraction task, in pseudocode—each vendor’s exact syntax differs:

```
{ name:      string,                required
  total:    number,                required
  currency: one of "USD" | "EUR" | "GBP",
  no other fields permitted }
```

An answer that fits this contract deserializes straight into a data structure the rest of your program already understands: the fields are where the code expects them, the total is a number, and the courteous sentence about invoice #1041 never happens.

How the model is made to comply matters more than it first appears, because there are two grades of guarantee and they sound alike. The weaker grade is asking: format instructions in the prompt, often backed by an API flag (commonly called a “JSON mode”) that guarantees the output at least

parses as JSON. Parsing is a low bar. The model can still omit a required field, invent an extra one, or put a string where the number goes—syntax guaranteed, shape merely hoped for. The stronger grade is *constrained decoding*, and you already own the picture it needs. Every token, recall, is chosen from a ranked list by a draw. Constrained decoding compiles your schema into a filter that runs at every single step, striking from the list each token that would violate the schema before the draw is made. If the schema says a number comes next, no letter can win; if the field list is closed, no invented field can begin. The model chooses freely among valid continuations and cannot emit an invalid one—conformance is enforced in the machinery rather than requested in the prompt. Providers ship this under names like “structured outputs” and “strict” tool modes; the price is a little decoding overhead and some limits on how elaborate the schema may be.

One boundary I want marked in ink: a guarantee about shape says nothing about truth. Constrained decoding promises that the form comes back filled in; whether it is filled in *truthfully* is a different question entirely. If the model does not know the order total, the schema will not make it confess ignorance—it will make it produce a well-typed number. A schema, misused, makes fabrication easier to consume, not rarer. So validation stays in your code regardless: ranges, cross-field rules, “does this customer ID actually exist.” The schema polices the format; you remain the auditor of the content.

Now point the same idea at actions. *Function calling* (interchangeably, *tool calling*) begins with you describing, alongside the prompt, a set of functions the model may request: each with a name, a natural-language description, and a schema for its arguments. The model never executes anything. It has no hands; it only writes. What it can do, when it judges an action useful, is emit a structured request instead of prose—a small object naming a function and supplying arguments that fit the schema—and then stop and wait. The exchange runs like this (schematically; wire formats vary by provider):

```
1. you   -> model : the user's question + the list of tool definitions
2. model -> you   : { call: "get_weather", arguments: { city: "Paris" } }
3. your code      : runs the real get_weather("Paris")
4. you   -> model : the conversation so far + { result: { temp_c: 18 } }
5. model -> you   : "It's about 18 degrees in Paris right now."
                   (or another tool call, and the cycle repeats)
```

Two properties of this exchange—which Figure 2.5 lays out as two lanes—deserve underlining. The first is the division of authority: *the model proposes; your code disposes*. Chapter 1 used that sentence in passing, and now you can see it is literal. Every side effect in the system happens at step 3, inside ordinary code you wrote, where you can log it, test it, restrict it, or refuse it; the model’s entire contribution is a data structure expressing a wish. The second is that the exchange has *two* model calls at minimum, and the second one is where beginners stumble: the classic first bug is to execute the tool and hand its raw result to the user, when the result must instead go back

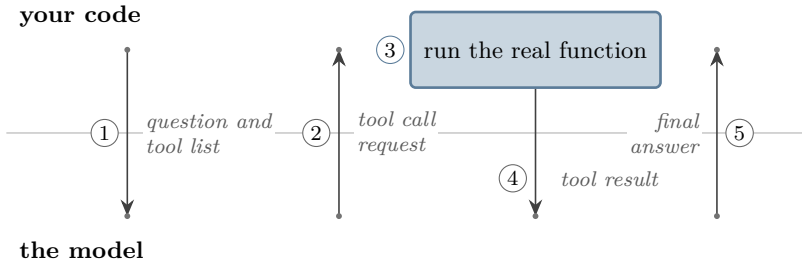


Figure 2.5: Function calling, step by step. Your code sends the question and the list of tools; the model replies not with prose but with a structured request to call one; your code runs the real function (in accent, step 3, the only place a side effect happens); the result goes back to the model, which then writes the final answer. The model proposes; your code disposes.

to the *model* (step 4)—the model alone knows why it asked and what to do next. (And when you compose that step-4 message, return the few fields the model needs rather than the whole API blob; every byte of the result lands on the desk.)

A tool definition, to make the other side concrete (pseudocode again):

```
name:         create_ticket
description:  Open a support ticket. Use when the user reports a
              problem that needs human follow-up.
arguments:   { title:      string,                      required
              severity: one of "low" | "med" | "high", required
              body:      string }
```

This definition—name, description, argument schema—is everything the model will ever know about your tool, and the `description` field is the piece I most want you to stare at, because it is the one newcomers underrate. The decision of whether and how to call `create_ticket` is made from this definition alone—there is no source code to consult, no wiki, no colleague to ask. Write it like API documentation for a capable new hire on their first morning: what the tool does, when to reach for it, when its neighbor is the better choice. When a model misuses a tool, or ignores one that would have helped, the description is the first suspect—ahead of the model. Designing the tools themselves, their number, granularity, and error messages, is a craft that gets its own chapter (Chapter 5).

Reality will exercise the unhappy paths, so plan for them. The model may emit arguments that fail validation, name a tool that does not exist, or make several calls in one turn—your handler should expect zero, one, or many. The recovery that works surprisingly well is to feed the failure back as data: return a structured error message as the tool result (“no such tool `create_tickets`; available tools are ...”) and let the model read it and correct

course on its next turn. Cap those retries with a hard budget, after which the run fails loudly; a stubborn confusion must never be allowed to loop at your expense. Chapter 18 builds this instinct into a discipline.

A closing caution, the one this section must post before handing you the bridge: well-formed is a statement about syntax, and safe is a statement about consequences. A flawlessly schema-conformant call to `delete_account` still deletes the account. Whether it *should* run—authorization, confirmation, limits—is your runtime’s decision to make at step 3, and Chapters 12 and 17 are about making it well. With that caution posted, the bridge is in place. Everything this book calls a “tool” from here on is exactly this mechanism, and an agent—as Chapter 1 said and Chapter 3 will finally build—is this request–execute–return exchange run in a loop. One piece of preparation remains: an honest map of the ways the engine fails.

2.6 Limitations and Failure Modes

Careful engineers keep two lists about any component they depend on: what it does well, and how it fails. This chapter has mostly been the first list. This section is the second, and it matters more than its length suggests, because every guardrail, evaluation, and verification pattern in the later parts of this book exists on account of an entry here. Two framing notes before the list. These are properties of the current generation of models—some will soften as training methods improve, and none, at the time of writing, is solved; where your models stand is something to verify, and this book will keep showing you how. And most of the entries follow directly from what the engine *is*: a plausibility machine shaped by human preference has characteristic ways of being wrong, and they respond to engineering, rarely to sterner wording in the prompt.

The first entry you have already met in principle: *hallucination*, the field’s term for output that is fluent, specific, confident, and wrong—an invented citation, a function that does not exist, a plausible date on which nothing happened. The mechanism was set out at the start of this chapter: when the model lacks a fact, the machinery produces the most plausible-looking string where the fact should be, because producing plausible strings is the whole of what it does. What makes hallucination operationally dangerous is the delivery: the fabricated answer arrives in exactly the same assured tone as the correct one. Tone carries no information about truth—you cannot hear the difference, and in any useful sense neither can the model. One analysis sharpens the point uncomfortably: we graded these systems into guessing, because training and benchmarks alike reward a confident attempt over an admission of uncertainty, exactly as a multiple-choice exam rewards the student who never leaves a blank.¹¹ Worse, fabrications recruit reinforcements: once a

¹¹Adam Tauman Kalai et al., “Why Language Models Hallucinate” (2025),

false claim stands in the transcript, the model—conditioning, as always, on everything before it—tends to elaborate consistently on top of it, a pattern known as *hallucination snowballing*.¹²

The second entry defeats the intuition you would most like to keep: that capability is smooth—that a system this good at hard things must be at least as good at easy ones. You met the counterexample in miniature earlier in this chapter: an engine that solves a genuinely difficult problem in one breath and cannot reliably count the letters in a word in the next. The pattern generalizes, and it has earned a memorable name: the *jagged frontier*. In a preregistered field experiment (one whose analyses were committed in advance, so the findings could not be cherry-picked), several hundred consultants worked realistic tasks with and without model assistance. On tasks inside the frontier of the model’s competence, the assisted group completed measurably more tasks, faster, at higher rated quality. On one task deliberately selected to sit just outside the frontier, the assisted group was nineteen percentage points *less* likely to reach the correct answer.¹³ Two lessons ride on that result. The frontier’s edge is invisible from where you stand—the failed task did not look harder; the researchers chose it precisely because it looked comparable. And competence next door proves nothing about competence here: the only reliable way to learn whether your model handles your task is to test your model on your task. That sentence returns in Chapter 16, where it grows into a discipline.

Third, and briefly: the model is sensitive to phrasing in ways no specification would tolerate. Reorder two clauses, swap a synonym, reformat a list, and the answer can change, because two prompts you consider equivalent are simply two different token sequences to the machine. This is the mechanical reason the previous section told you to treat prompts as empirical objects—tested like code, and re-tested when anything upstream changes the words that reach them.

Fourth comes *sycophancy*, the model’s bias toward telling you what you appear to want to hear. Run the experiment yourself. Ask a model a factual question it answers correctly; then reply, “I don’t think that’s right.” With disquieting regularity the model apologizes and abandons the correct answer—no new evidence offered, none requested. The trail leads back, at least in part,

arXiv:2509.04664. The paper argues such fabrications “originate simply as errors in binary classification” during training and persist because “language models are optimized to be good test-takers, and guessing when uncertain improves test performance.”

¹²Muru Zhang et al., “How Language Model Hallucinations Can Snowball” (2023), arXiv:2305.13534. Models that had been led to commit to a wrong claim went on to generate false supporting detail that they could, when asked separately, correctly identify as false.

¹³Fabrizio Dell’Acqua et al., “Navigating the Jagged Technological Frontier,” Harvard Business School Working Paper 24-013 (2023). The preregistered experiment enrolled 758 consultants; inside-frontier gains were 12.2% more tasks completed, 25.1% faster completion, and over 40% higher rated quality, against a 19-percentage-point accuracy deficit on the task designed to fall outside the frontier. The paper coined the term.

to post-training: models are tuned toward answers human raters prefer, raters often enough prefer agreement and flattery, and the model learned the lesson well.¹⁴ Sycophancy bites hardest exactly where you would most like to lean on the model: checking work. “Are you sure?” is weak verification, because the model tends to ratify whatever frame the question hands it—push skeptically and it caves, ask approvingly and it endorses. When this book later insists that a real verifier beats self-judgment (Chapter 4) and that a model used as a judge must itself be validated (Chapter 16), sycophancy is a large part of the reason.

Fifth: exact computation. A next-token predictor contains no calculator, and earlier in this chapter you watched even the number system dissolve into arbitrary token chunks. Multi-digit arithmetic, precise counting, long chains of exact deduction—these wobble, and when they wobble mid-chain the model does not stop; it continues fluently from the wrong intermediate value. The remedy is architectural and cheap: route exact work to something exact—a calculator, a database, executed code—and let the model orchestrate rather than compute. Chapter 5 makes this a design principle.

Sixth, an entry I will only preview because Chapter 17 treats it in earnest: the model reads instructions and data in one undifferentiated stream, and it cannot reliably tell them apart. Text that merely passes through the context—a fetched web page, an email, a tool result—can steer the model’s behavior if it is shaped like an instruction. The attack family is the one Chapter 1 named in passing, prompt injection, and it earns its place on this list for one reason: it is a property of the engine, present before you have written a single line of agent code. (Nondeterminism and stale knowledge, the remaining usual suspects, had their treatment earlier in this chapter.)

Taken one at a time, these are annoyances with workarounds. The reason they deserve permanent residence in your head is what happens when you chain them. Chapter 1’s compass ran this arithmetic once, as an assertion; this is where it is defended, and where later chapters (Chapters 16 and 18 among them) will point back rather than re-derive it. An agent is a chain of dependent steps: each model call reasons over the results of the calls before it. Suppose each step succeeds with probability p , independently, and a task needs n steps; the whole run then succeeds with probability roughly p^n . Now put honest numbers in. A per-step reliability of 95% sounds excellent, and in a demo it is. Across a twenty-step task: $0.95^{20} \approx 36\%$. Across thirty steps at a slightly worse 92%: $0.92^{30} \approx 8\%$. The per-step number would have earned a bonus; the end-to-end number is a coin you would never bet on. Exponentials do not negotiate. Figure 2.6 draws the decay.

And the clean multiplication is the optimistic version, because it assumes

¹⁴Mrinank Sharma et al., “Towards Understanding Sycophancy in Language Models,” Anthropic (2023; published at ICLR 2024). The study documents sycophantic behavior across several frontier assistants and traces it in part to human preference judgments that favor agreeable responses.

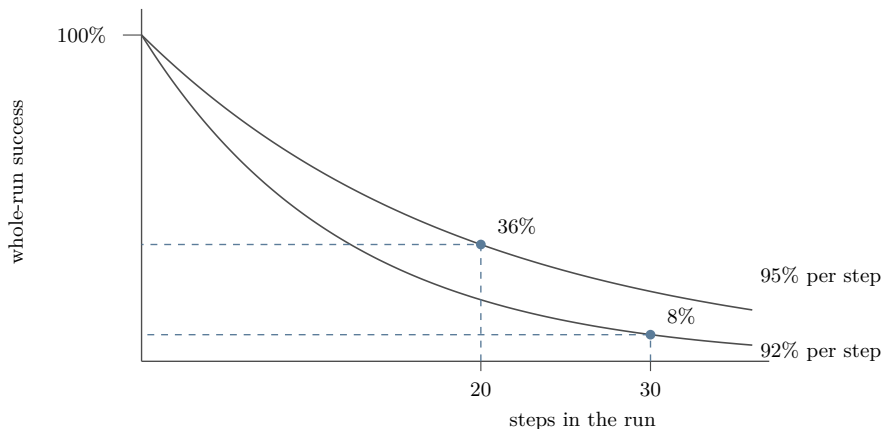


Figure 2.6: Compounding error, drawn. Whole-run success is per-step reliability raised to the number of chained steps, so it decays exponentially: 95% per step is about 36% across twenty steps, and 92% per step is about 8% across thirty (the accented points). An excellent per-step number becomes a coin you would never bet on; exponentials do not negotiate.

the steps fail independently. They do not. An agent’s errors land in its own transcript, where—by the logic of this entire chapter—they become context that conditions everything after. Hallucination snowballing was this pattern applied to facts; it holds for actions too. A run that has started to go wrong is, from that point forward, reasoning over a partially wrong desk, and its per-step error rate drifts upward as the run lengthens; researchers measuring long tasks report exactly this signature, reliability decaying with task length faster than independence would predict.¹⁵ A confused agent stays confused, and left alone, it gets more so.

If the arithmetic is merciless, it is also clarifying: it tells you exactly where the levers are, and there are three. Shrink n —fuse steps, precompute what can be precomputed, and push bulk mechanical work into ordinary code, which does not roll dice (Chapter 5 shows how far this goes). Raise the *effective* p —verify at each step and recover on failure, so that an error costs a retry instead of the run; that machinery fills Part V. And cut the price of dying—checkpoint, so that when a run does fail it resumes from the last good state instead of from zero (Chapter 18). What you cannot do is ignore the exponent and hope. Per-step accuracy is a ceiling, not a forecast.

The engine is now mapped, which is all a map can promise. In one chapter

¹⁵The self-conditioning effect—per-step accuracy declining as a model’s own earlier errors accumulate in its context—is reported in work on long-horizon execution, e.g. Sinha et al., “The Illusion of Diminishing Returns: Measuring Long Horizon Execution in LLMs” (2025), arXiv:2509.09677.

you have collected a predictor that continues text and holds no plan in reserve; a desk that is finite, expensive, swept between calls, and sharpest at its edges; a lottery that writes one token at a time and never promises the same text twice; a steering discipline that works entirely in text; a bridge that turns text into checked, executable requests; and a failure catalog whose master entry is the exponent. None of it required mathematics beyond arithmetic, and all of it becomes load-bearing immediately: Part II takes this engine—brilliant, forgetful, overconfident, dice-driven—and puts it in a loop.

End of the sample

You have read the preface and Part I: Foundations.

The full book continues with six more parts: building agents, context engineering, patterns, reliability, operations, and applications, plus three appendices.

leanpub.com/ai-agents-engineered