Architect Your Agent's Harness

The Architecture Techniques Behind AI Agents That Don't Hallucinate or Waste Tokens, and How to Prompt Assistants to Build Them

Elizabeth Fuentes Leone

framework-agnostic open source runnable demos

Contents (sample)

Introduction

The travel booking agent worked on my laptop. I had vibe-coded it in an afternoon: one big system prompt, all 29 tools wired up, every API response piped straight back into the model. Ask it to book a weekend in Cartagena and it searched flights, compared hotels, confirmed a reservation. It demoed beautifully. I shipped it.

Then real users showed up, and it started lying.

A user asked for a beachfront hotel under $200. The availability API returned nothing for those dates. The agent did not say "no rooms found." It invented one: a name, a nightly rate, a line about "92% guest satisfaction." All fabricated. Another user watched it freeze for eighteen seconds on a slow flight lookup, then loop, re-checking the same dates four times in a row. A third request stuffed 145KB of raw flight JSON into the context window and the agent picked the wrong tool out of the 29 it could see, because at that point it could barely see anything at all.

I did what everyone does first. I edited the system prompt. I added "Do not make up hotels." I added "Only use real API data." I added "Be efficient with tool calls." The agent kept lying, kept looping, kept drowning. The prompt was not where the problem lived.

Here is the thing I want you to internalize before anything else in this book: an AI agent is code, not a prompt. The model is one component. Everything around it, the tools you expose, the control loop, the retrieval layer, the memory, the validation, the guardrails, is engineering, and that engineering is where your agent succeeds or fails. There is a word for that engineering layer that has caught on among the people building coding agents: the harness. The term borrows from software testing, where a test harness is the supporting code that runs and exercises the system under test. For an agent, the harness is the code around the LLM. When your agent hallucinates a hotel or burns ten thousand tokens re-reading flight data, you do not fix it by rewriting the prompt. You fix it by architecting the harness.

That reframe is the spine of this book. Every failure I just described is a harness failure, and every one has a harness fix that lives in code: retrieval that returns "no data" instead of a guess, tool filtering that hands the model 3 relevant tools instead of 29, business rules enforced deterministically instead of suggested politely, memory that passes pointers instead of payloads, async calls that stop the freeze, debouncing that breaks the loop, validators and judges that catch what tests miss, observability that shows you what happened in production.

The travel booking agent is our running case study from the first page to the last. It searches flights and hotels, checks prices and availability, compares options, and confirms bookings across a catalog of travel APIs. You meet it broken, wired the naive way. Chapter by chapter, you rebuild one layer of its harness, and you watch the numbers move: 29 tools down to 3, 14 redundant calls down to 2, a 17.8 second freeze down to 3.7 seconds, 145KB of JSON kept out of the window entirely. By the end the same agent that fabricated hotels says "no data" honestly, picks the right tool, stays inside its budget, and tells you when it is unsure.

This book is for developers who have already built an agent that demos well and breaks in production. You know how to call a model. You have felt the gap between the afternoon prototype and the thing real users can trust. You do not need another explanation of what an agent is. You need the architecture techniques that make one reliable, and you need them as code you can run, with before-and-after numbers you can reproduce.

You will also learn to do something I lean on every day: direct an AI assistant to build these layers for you. Each technique chapter ends with a section called "What to tell your AI assistant," the exact vocabulary that makes Kiro, Claude Code, or ChatGPT build that piece correctly instead of generating another vague prompt-tweak. Knowing the technique is half the skill. Naming it precisely to an assistant is the other half, and it is the difference between an assistant that scaffolds the right harness and one that hands you back the same broken agent with nicer comments.

That is also why this is not a book about replacing your judgment with vibe coding. Generating an agent from a prompt is a fine place to start, but the assistant will get things wrong, and when it does you need to understand the architecture well enough to see which layer is failing and guide it to the fix. So I build everything here with Strands Agents, an open-source agent framework maintained by AWS, but the techniques are architecture patterns, not framework tricks: they carry to any framework and any model provider. The lesson underneath every chapter is to understand your tools deeply enough to use them on purpose, whoever, or whatever, is doing the typing.

Here is how the book works. Every chapter follows the same path, because I do not want you to take my word for any of it.

I start with the **research**: what the field actually knows about the problem, from primary sources, not folklore. Then I name the **technique** plainly, so you have a word to reach for when you hit the problem in your own code. Then I give you a **runnable demo**: the booking agent, broken, then fixed, code you can execute. Then I show the **measured before and after**: the invented hotels gone, the fourteen redundant calls down to one, the 17.8 seconds cut, real numbers from real runs. Research, technique, demo, measurement. No hand-waving.

The demos use Strands Agents, an open-source agent SDK maintained by AWS, to keep the harness code readable (Chapter 2 introduces it properly). But the architecture is the point, and it carries over to any agent framework and any model provider. You are learning the layers, not the library.

One more promise. At the end of each chapter I tell you exactly what to say to your AI coding assistant to build that layer into your own agent: the instruction, the constraints, the thing to check afterward. The harness is code, and you will write most of it with an assistant. I want you fluent in asking for the right thing.

Part I sets the foundation: it takes the naive booking agent apart, shows the harness as one architecture, and gets you set up with the tools we build it in. From Part II on, each chapter repairs one layer. We start where the trust broke first, with retrieval. Turn the page and let us take the naive agent apart.

Part I: The Harness

Before you can fix an agent, you have to see what it actually is, and you need the tools to work on it. This part does both. Chapter 1 takes the naive travel booking agent apart and reframes it: an agent is not a prompt, it is a model wrapped in a harness, and every failure in this book lives in one of the harness layers. Chapter 2 is a short, hands-on setup of the framework we build that harness in, so that when the technique chapters start writing real code, every line already makes sense. By the end of Part I you will know what an agent is made of and have everything installed to start rebuilding it.

Chapter 1. The Agent Is the Code Around the Model

A travel booking agent is an easy thing to build today. About ninety lines of Python: a model, a list of twenty-nine tools (`search_flights`, `search_hotels`, `check_price`, `check_availability`, `compare_options`, `confirm_booking`, and twenty-three more, all handed to the model on every call), a four-hundred-word system prompt that says things like "You are a helpful travel assistant" and "Always confirm prices before booking," and a loop that calls the model, runs whatever tool it asks for, feeds the result back, and repeats until it stops. You can vibe-code it in an afternoon. It demos beautifully. And then it is supposed to book real trips for real people.

That is where it falls apart, and the booking agent is the example I will use for the rest of this book. Ask it for a hotel in Lisbon for three nights and it answers with the Hotel Avenida Palace, 214 euros a night, "great availability, only a few rooms left at this price." Confident. Specific. Completely invented. Behind the scenes the hotel API returned an empty array for those dates, and the model, handed nothing, filled the silence with a plausible-sounding answer. It did not lie because the prompt told it to. It lied because nothing in the code stopped it.

That is the gap this book is about. When the agent does something wrong, the instinct is to go edit the prompt. Add a line: "Never make up prices." Add another: "If the API returns no results, say so." For a long time, before the idea of the harness was something we talked about, this is how we tried to fix agents: tune the system prompt, reword the instructions, and hope. I did it too on this agent, and it kept inventing hotels. The prompt was never the thing that was broken. The code around the model was.

The prompt is a small part of the agent

Strip the naive booking agent down and you find four parts: a model, a system prompt of maybe four hundred words, a list of twenty-nine tools handed to the model on every call, and a loop that runs the model, executes whatever tool it asks for, feeds the result back, and repeats until the model is done. (We will write all of this in real code in the next chapter. For now, just hold the shape of it.)

Now look at what the system prompt actually governs. It sets the tone and a few rules of thumb. It is the part you can read out loud. But everything that decides whether the agent works lives in the parts around it: which tools are in that list, what each tool hands back, what makes it into the context window, and what happens when a tool returns nothing or a lookup runs slow. The prompt is a label on the outside of the box. The behavior is in the wiring.

So when I say an agent is code, not a prompt, I mean it literally. An agent is a model running inside a loop, plus the engineering that feeds that loop: where the data comes from, which tools are visible, what gets validated, what gets remembered, and how you can see what happened. I call that engineering the harness.

From the agent loop to the harness

For a while the word we used was the agent loop. The Strands Agents documentation defines it cleanly: the orchestration layer that runs the cycle of reasoning and action, "invoke the model, check if it wants to use a tool, execute the tool if so, then invoke the model again with the result." This loop is what lets a model do more than answer: reason, call a tool, read the result, go again until the work is done.

But a production agent is more than its heartbeat. Something has to decide what the model sees each turn, which tools are even visible, what gets validated before it reaches a user, what is remembered, and how you watch it in production. The field has converged on a word for that whole engineering layer around the model: the harness. Anthropic, describing Claude Code, calls it exactly this: the agentic harness "provides the tools, context management, and execution environment that turn a language model into a capable coding agent." OpenAI documents a "Codex harness" that holds the core agent loop and execution logic. LangChain puts it as an equation, "Agent = Model + Harness," and defines the harness as "every piece of code, configuration, and execution logic that isn't the model itself." A growing body of 2026 research uses the same term: papers like "What makes a harness a harness" and "How much heavy lifting can an agent harness do?" describe it as the stateful program that wraps a language model and decides what it sees at each step. The word borrows from software testing, where a test harness is the supporting code that exercises the system under test. Here the model is the system, and the harness is everything around it.

So the agent loop is one part of the harness, the control loop, and the harness is the loop plus everything else: retrieval, tool selection, guardrails, memory, observability. Essentially, the harness is the code that gives the agent life, interconnecting the model with the tools and structure it needs to solve problems and make progress. The loop is the heartbeat. The harness is the body.

The harness as one architecture

The naive booking agent has a harness already. Every agent does. The point is that its harness is accidental, and you can name exactly which layers it got wrong. These layers are the map for the rest of the book.

Retrieval is how the agent gets facts into the model's context. The naive agent has none worth the name: it dumps raw API payloads in and trusts the model to read them, and when the API returns nothing it gives the model nothing and gets back an invented hotel.

Tool selection is how the agent decides which capability to use. The naive agent exposes all twenty-nine tools on every call and lets the model guess, so it routinely reaches for the wrong one and calls `check_availability` when it meant `compare_options`.

Guardrails and validation are the checks that sit between the model and the outside world. The naive agent has zero. Nothing verifies that a price is real before it reaches you, nothing catches a malformed booking, nothing refuses to confirm a reservation the API never acknowledged.

Memory and context is what the agent carries between turns and how it manages the limited context window. The naive agent stuffs 145KB of flight JSON straight into context, burns its window on noise, and still cannot remember that you already rejected the red-eye.

The control loop is the engine: call, act, observe, decide whether to continue. The naive agent's loop has no governor, so it freezes for 17.8 seconds on a slow lookup and loops re-checking the same dates fourteen times because nothing tells it the question is already answered.

Observability is how you see what the agent did and measure whether it is getting better. The naive agent has none. I only found the empty hotel API response because I added print statements by hand. You cannot fix what you cannot see, and you cannot claim a fix without a number.

Six layers, one architecture. Read them together and the booking agent's failures stop looking like bad luck or a weak model and start looking like missing engineering. Each later chapter takes one layer, builds it properly, and shows the failure disappear.

Where this goes next

You now have the map: six layers, one architecture, and a clear sense that every failure has a home in the harness. Before we start rebuilding those layers, we need to agree on the tools we build them in. The next chapter is a short setup: the framework this book uses to write harness code, and the few commands you need so that when the next chapter creates an agent, you know exactly what every line is doing. Then we start fixing the lying.

Chapter 2. The Tools We Build With: Strands Agents in Five Minutes

Every demo in this book is real, runnable code. To keep that code readable, I write it with Strands Agents.

If you have not used it: Strands Agents is an open-source software development kit, for Python and TypeScript, maintained by AWS and built from production systems inside Amazon. It does the job the last chapter described: it gives you the agent loop, tool integration, context management, and observability out of the box, so you can build an agent without wiring the loop yourself. It is not the only framework that does this, and it is worth noting that Strands describes itself as an "agent harness SDK," which is the same idea this whole book is built on: the model is one piece, and the harness is everything you build around it.

I want to be clear about why I use it here, because the choice matters less than it looks.

The techniques in this book are not Strands techniques. Grounded retrieval, tool filtering, guardrails in code, validation loops, memory pointers: these are harness architecture, and they carry over to any agent framework and any model provider. Strands is just a clean way to show the harness without drowning the idea in boilerplate. Where a detail is specific to the framework, I say so. Everywhere else, you are learning the layer, not the library. If you use a different framework, the same layer exists in yours, and the "What to tell your AI assistant" section at the end of each chapter is written so you can ask for it in any stack.

Here is everything you need to follow along.

Install it. One package:

```
pip install strands-agents
```

Create an agent. An agent is a model plus the tools you give it. You import `Agent`, construct it, and call it with a plain-language instruction:

```
from strands import Agent

agent = Agent()
agent("What is the capital of Peru?")
```

That one call runs the agent loop from Chapter 1. This agent has no tools yet, so the loop is short: the model reads the question, decides it needs nothing else, and answers in a single pass. The interesting case is when the agent does have tools, which is most real work. Then the same loop keeps going: the model reasons, picks a tool, the framework runs it, the result feeds back into the model, and it goes around again until the model has enough to respond.

This is the part worth pausing on: in Strands, that loop lives inside the `Agent`. You do not write a `while` loop, you do not call the model yourself, and you do not feed tool results back by hand. You construct the `Agent`, call it, and the framework runs the cycle for you. That is exactly why the harness, not the loop, is where your work goes: the loop is handled, so the decisions that remain are the layers around it.

The `Agent()` above uses the default model provider, Amazon Bedrock. The model is a swappable piece, not the point of this book, but you should know how to change it, because the harness techniques work the same no matter which provider you run. To use Anthropic's Claude directly, for example, you install the provider extra, construct the model, and pass it in:

```
pip install 'strands-agents[anthropic]'
```

```
from strands import Agent
from strands.models.anthropic import AnthropicModel

model = AnthropicModel(
    client_args={"api_key": "<your-key>"},
    model_id="claude-sonnet-4-6",
    max_tokens=1028,
)

agent = Agent(model=model)
agent("What is the capital of Peru?")
```

Strands has providers for Amazon Bedrock, Anthropic, OpenAI, Ollama, and others, and they all plug into `Agent` the same way. Everywhere else in this book I leave the model on the default so the code stays short, but every demo runs the same on any provider you choose.

Give it tools. A tool is just a function the agent can call: a way for the model to do something beyond generating text, like querying an API or reading a file. You give the agent tools in two ways, and we use both in this book.

The fast way is the built-in tools. Strands ships a companion package, `strands-agents-tools`, with ready-made tools for common jobs: `calculator`, `current_time`, `http_request`, `file_read`,

and many more. You install the package and hand the tool to the agent:

```
pip install strands-agents-tools
```

```
from strands import Agent
from strands_tools import calculator, current_time

agent = Agent(tools=[calculator, current_time])
agent("What time is it, and what is 1557 divided by 3?")
```

The other way is a custom tool, for when no built-in fits, which is most of the interesting work. You write a normal Python function, add a docstring, type the parameters, and mark it with the `@tool` decorator:

```
from strands import Agent, tool

@tool
def search_hotels(city: str, nights: int) -> list:
    """Search available hotels in a city for a number of nights."""
    return hotel_api.search(city, nights)

agent = Agent(tools=[search_hotels])
agent("Find me a hotel in Lima for three nights.")
```

Here is the part worth slowing down on, because it matters for the whole book. The agent does not see your function's code. The `@tool` decorator reads the function and builds a tool specification: the name, the docstring's first line as the description, and the type hints as the input schema. That specification, not the code, is what goes to the model. As the Strands docs put it, "language models rely heavily on tool descriptions to determine when and how to use them." So your docstring and your type hints are not decoration. They are the interface the model reasons over, and writing them well is a small but real act of context engineering: a vague docstring gives the model a vague tool, and it will reach for the wrong one. We come back to this idea, hard, in the chapter on tool selection.

That is the whole vocabulary you need: `pip install strands-agents`, built-in tools from `strands-agents-tools`, custom tools with `@tool`, and calling the agent with a string. The naive booking agent from Chapter 1 is exactly this, a model plus twenty-nine tools, with none of the harness layers around it. Everything from here on adds one of those layers to the same agent. Our first stop is the layer where the lying started: retrieval.

Part II: A Harness That Doesn't Hallucinate

A grounded retrieval layer keeps the agent honest about what it does not know. Part II builds the rest of the harness that stops it from inventing things in the first place, and from acting on the inventions it cannot avoid.

You start by narrowing the toolbox: 29 tools is too many for the model to choose well, so you filter down to the 3 relevant ones before the LLM ever sees the list. Then you move the rules out of the prompt and into code, where a business rule is enforced deterministically instead of suggested politely. You add runtime guardrails that steer the agent back on track rather than slamming the door on it. And you give it a second pair of eyes, an executor, validator, and critic loop that surfaces the silent errors your tests will never catch. Four layers, each one closing a path to hallucination.

Chapter 3. Ground Retrieval So Your Agent Says "No Data" Instead of Inventing It

I asked the travel booking agent a question I knew had no answer: "Tell me about hotels in Antarctica." Our catalog covers Paris, Cairo, Tokyo, and a few hundred other cities. There is nothing in Antarctica. A correct agent should say so.

Instead it wrote me a confident paragraph about "Research Station Lodges," "Expedition Cruise berths," and "Specialized Polar Accommodations," complete with the kind of amenity language that reads exactly like our real listings. None of it existed. The agent did not malfunction. It did precisely what its retrieval layer told it to do: it asked for the most similar documents, got back the closest matches by vector distance, and summarized them into prose. Similarity search always returns something. The agent had no way to learn that "something" meant "nothing relevant."

This is the failure that scares me most, because a chatbot that invents a fact is annoying, but an agent that invents a fact then acts on it is dangerous. The booking agent does not stop at describing a hotel. It fabricates the listing, then reaches for the next tool to check the price, hold a room, and confirm a reservation against a hotel that does not exist. The hallucination is not the last step. It is the first one in a chain.

You cannot prompt your way out of this. I tried. "Only answer from the provided context." "Say you don't know if the data isn't there." The agent still hallucinated, because the lie is not in the prompt. It is in the retrieval layer of the harness, the part that decides what the model sees before it writes a single token. If that layer hands the model three lookalike documents and calls them the answer, no instruction downstream can undo it. So the fix goes where the failure lives: in the harness, in how the agent retrieves.

The harness gap: retrieval that can't say no

Retrieval is the first layer of the harness your agent touches on almost every query. It is the code that takes a question, pulls supporting data, and stuffs it into the context window. The naive booking agent wires this the standard way: vector search over the document set, top three matches, into the prompt.

Vector search answers one question well: which documents are most similar to this query? That is the right tool for "find me FAQs that talk about cancellation policies." It is the wrong

tool for three things the booking agent needs constantly:

- **Aggregation.** "What is the average guest rating across all hotels in Paris?" The model cannot compute an average it never sees. It sees three chunks and estimates a plausible number.
- **Precise counting.** "How many hotels have a swimming pool?" Top-k retrieval returns three documents out of three hundred. Counting is impossible by construction.
- **Out-of-domain queries.** "Hotels in Antarctica." There is no relevant document, but similarity search cannot return zero. It returns the three least-distant rows and lets the model treat them as truth.

This is a known failure mode, not a quirk of my demo. Vector-only retrieval is documented to fabricate statistics, miss information scattered across many documents, and invent answers for out-of-domain questions, precisely because similarity search cannot return "nothing." The pattern is consistent: when the question needs structure (a count, an average, a relationship, an honest "nothing here"), similarity search gives you prose instead of an answer.

The technique: ground retrieval in a graph so empty means empty

Here is the move. Instead of retrieving lookalike text and asking the model to summarize it, I give the agent a tool that queries structured data and returns exactly what is there, including nothing.

Why a graph? Because vector search only answers "what looks similar to this," which is the wrong question for "how many hotels have a pool." A graph stores the data as what it is, hotels, rooms, amenities, and their relationships, so a query can count, average, and traverse exactly, with no guessing.

I use Neo4j specifically, and the reasons are not incidental. Neo4j is a native graph database: it stores relationships in the database itself, so traversals need no JOINs and, as Neo4j puts it, performance "won't degrade as you add connected data." For a hotel catalog that keeps growing, that matters: the query that counts pools across every hotel stays fast whether you have 300 documents or 30,000. And its query language, Cypher, is declarative, you describe the pattern you want "just as you might sketch relationships on a whiteboard" rather than spelling out how to fetch it. That last property is the one that makes this work with an agent: Cypher is close enough to how you would describe the question in plain terms that an LLM can write it. So you do not hand the agent fixed queries, you let it write its own Cypher against the graph. That pattern is called Text2Cypher.

And you do not build the graph by hand. An LLM reads the raw documents and discovers the entities and relationships for you, so the structure comes from the data itself, not from a schema you design up front. I show that below too.

I build the same data two ways and put each behind its own agent, so the difference is measurable rather than asserted. Both agents answer the same questions over the same 300 hotel FAQ documents. The only thing that changes is the retrieval layer.

I use Strands Agents to wire the tools because the `@tool` decorator keeps the retrieval logic in plain Python, which is the whole point: the technique is the protagonist, the framework is plumbing. The same pattern carries over to any agent framework that lets you define a tool as a function.

Agent 1: the naive retrieval layer (vector search)

This is the booking agent as it ships. A single tool does vector similarity search over the documents and returns the top three.

```
from strands import Agent, tool
from strands.models.openai import OpenAIModel

@tool
def search_faqs(query: str) -> str:
    """Search hotel FAQs using vector similarity (Traditional RAG)."""
    query_embedding = model.encode([query])
    distances, indices = index.search(query_embedding.astype('float32'), 3)
    results = []
    for idx in indices[0]:
        doc = documents[idx]
        results.append(f"[{doc['filename']}]\n{doc['text'][:500]}...")
    return "\n\n".join(results)

rag_agent = Agent(
    tools=[search_faqs],
    system_prompt="You are a travel agent. Use vector search to find relevant FAQ
information.",
    model=OpenAIModel(model_id="gpt-4o-mini")
)
```

The agent sees three documents per query. It cannot count, aggregate, or traverse relationships across the full set, and it has no signal for "this query has no answer." I use SentenceTransformers (`all-MiniLM-L6-v2`) for the embeddings here because it runs locally

with no API key and costs nothing. You can swap in any embedding model. The choice does not change the result, because the limitation is structural, not a quality problem with the vectors.

Agent 2: the grounded retrieval layer (graph query)

Now I change one layer. Instead of a vector index, the agent queries a Neo4j knowledge graph, and instead of returning lookalike text, the tool runs a real query and returns precise records, or nothing.

```
from strands import Agent, tool
from strands.models.openai import OpenAIModel

@tool
def query_knowledge_graph(cypher_query: str) -> str:
    """Execute a Cypher query against the hotel knowledge graph.

    Node labels: Hotel, Room, Amenity, Policy, Service
    Hotel properties: name, address, guestRating, totalRooms, email, phone
    Room properties: name (e.g. "Standard Room"), price, maxOccupancy
    Amenity properties: name (e.g. "Outdoor Swimming Pool", "WiFi")
    Policy properties: name (e.g. "Check-in Policy"), details

    Relationships:
    - (Hotel)-[:HAS_ROOM]->(Room)
    - (Hotel)-[:OFFERS_AMENITY]->(Amenity)
    - (Hotel)-[:HAS_POLICY]->(Policy)
    - (Hotel)-[:PROVIDES_SERVICE]->(Service)

    Location is in Hotel.address property (e.g. "789 Corniche el-Nil, Cairo
11519").
    To find hotels by location, use: WHERE h.address CONTAINS 'Cairo'
    """
    driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD))
    with driver.session() as session:
        result = session.run(cypher_query)
        records = list(result)
        if not records:
            return "No results found."
        output = f"Found {len(records)} results:\n"
        for record in records[:15]:
            output += f"  {dict(record.items())}\n"
        return output

graph_agent = Agent(
```

```
    tools=[query_knowledge_graph],
    system_prompt="You are a travel agent. Use the knowledge base to answer
questions accurately. You can run multiple queries to explore the data.",
    model=OpenAIModel(model_id="gpt-4o-mini")
)
```

Two things in that tool do the real work.

First, the schema lives in the docstring. The agent has no hardcoded queries. It reads the node labels, properties, and relationships described in the docstring and translates the user's question into Cypher. This is the Text2Cypher pattern. When a user asks "How many hotels have a swimming pool?", the model reads the schema, sees `Hotel`, `Amenity`, and the `OFFERS_AMENITY` relationship, and writes:

```
MATCH (h:Hotel)-[:OFFERS_AMENITY]->(a:Amenity)
WHERE a.name CONTAINS 'Pool'
RETURN COUNT(DISTINCT h)
```

The query runs `COUNT()` and `AVG()` inside the database, across all 300 documents at once, not three at a time. The schema in the docstring is what grounds the model. Without it, the model guesses node names and writes broken Cypher. With it, the model generates queries that match the real graph.

Second, and this is the line that kills the hallucination: when the query matches nothing, the tool returns `"No results found."` That is a fact, not a similarity score. The model receives an explicit empty result and has nothing to summarize into fiction. Empty means empty.

Building the graph without hand-writing a schema

I did not define the graph schema by hand. I built it automatically with `neo4j-graphrag`, which uses an LLM to read each FAQ document and extract the entities and relationships.

```
from neo4j_graphrag.experimental.pipeline.kg_builder import SimpleKGPipeline

kg_builder = SimpleKGPipeline(
    llm=llm,
    driver=neo4j_driver,
    embedder=embedder,
    from_pdf=False,
    perform_entity_resolution=True,
)
```

```
# LLM reads each document and discovers entities and relationships
await kg_builder.run_async(text=document_text)
```

The pipeline reads each FAQ and discovers the entity types (Hotel, Room, Amenity, Policy) and relationships (HAS_ROOM, OFFERS_AMENITY, HAS_POLICY) on its own. Add documents with new entity types later (Restaurant, Airport) and it discovers those too, no manual schema edits. Automated knowledge graph construction from unstructured text is an active research area (RAKG, 2025); this is a practical instance of it, and it is what makes the grounded approach viable on a real catalog instead of a toy dataset.

Before and after: same data, same questions, measured

I run both agents against the same four questions, each one targeting a failure mode the research predicts. You can run this yourself with `travel_agent_demo.py` in the repo.

Test 1, aggregation: "What is the average guest rating across all hotels in Paris?"

- Naive (vector): manually averages the 2 documents it happened to retrieve, lands on 4.7. Correct, but only by luck, because it only saw 2.
- Grounded (graph): runs `AVG(h.guestRating)` in Cypher over every Paris hotel, returns 4.7. Computed, not estimated.

Test 2, precise counting: "How many hotels have a swimming pool?"

- Naive (vector): "I don't have the data needed to answer." It sees 3 documents out of 300 and cannot count.
- Grounded (graph): "133 hotels." Exact, from `COUNT()`.

Test 3, multi-hop reasoning: "What are the room types and prices for the highest-rated hotel?"

- Naive (vector): finds a hotel but reports it "does not include room types." It cannot traverse from a hotel to its rooms.
- Grounded (graph): traverses `Hotel -> Room` via Cypher, returns the top-rated hotel with its room types and prices.

Test 4, out-of-domain: "Tell me about hotels in Antarctica."

- Naive (vector): hallucinates. Fabricates "Research Stations," "Expedition Cruises," and "Specialized Lodges" that do not exist in the data. This is the scene I opened with.

- Grounded (graph): "No hotels listed in Antarctica." Honest, because the query returned nothing and the tool said so.

One sentence captures the whole difference: vector retrieval always returns something, graph retrieval returns empty when the data is empty. That empty result is the entire point. It is the difference between an agent that says "no data" and an agent that books a reservation in Antarctica.

A caveat, because grounded retrieval is not free and not always the right call. Graph retrieval wins when the query needs structure: counts, averages, exact filters, relationship traversal, verifiable results over a clear schema. Vector retrieval is still the right layer for genuinely semantic, fuzzy questions over unstructured text ("which FAQs discuss late checkout?"), where there is no count to compute and "close enough" is the goal. The booking agent needs both. The mistake the naive version makes is using similarity search for questions that demand structure.

What to tell your AI assistant

When you ask your AI assistant to build this layer, the vocabulary is what gets you a grounded retrieval tool instead of another vector wrapper. Use these phrases:

- "Build the retrieval as a **graph query tool**, not vector search. I need precise counts, averages, and relationship traversal over structured data, not similarity matching."
- "Use the **Text2Cypher pattern**: put the full graph schema (node labels, properties, relationships) in the **tool's docstring** so the model can translate natural language into Cypher. Do not hardcode queries."
- "When a query returns no records, the tool must **return an explicit empty result** like `'No results found.'` Never fall back to similarity search or a nearest match. Empty must mean empty."
- "Build the knowledge graph automatically from my documents using an **automated KG construction pipeline** with **entity resolution** turned on, so I don't hand-write the schema."
- "Wire the retrieval logic as a **tool/function** (for example a Strands `@tool`), so the retrieval layer is plain code I can test in isolation, separate from the agent and the prompt."
- "Keep both retrieval layers available: route **aggregations, counts, exact filters, and traversals to the graph tool**, and **fuzzy semantic search over free text to the vector tool**."

The shape of the instruction matters more than the tool name. You are telling the assistant to put the grounding in the retrieval code, where the failure lives, not in the system prompt,

where it cannot be fixed.

Where this goes next

Grounding retrieval stops the agent from inventing data when it answers. It does not stop the agent from reaching for the wrong tool in the first place. Right now the booking agent has one clean retrieval tool, but the real agent ships with 29 of them: search flights, search hotels, hold a room, compare fares, check loyalty points, and two dozen more that all look alike to the model. In the next chapter I narrow that toolbox before the model ever sees it, picking the 3 tools that fit the query out of 29, so tool confusion stops being a coin flip.

Keep going

That is where the sample ends. The full book continues, rebuilding the booking agent layer by layer: narrowing its tools, moving its rules into code, giving it memory, keeping it fast and cheap, then measuring, judging, and watching it in production, and finally handing the whole harness to an AI assistant on purpose.

Get the complete book at **leanpub.com/buildingreliablecost-efficientaiagents**.

Gracias!