The AI Diagnostic Manual — Free Sample

Symptom-Based Troubleshooting for LLM Applications

Cynked Press

2026

First edition, 2026. Material marked as dated was checked in September 2026.

Start here

You have something broken and you do not want to read a book. Find the line that sounds like the thing you would say out loud, and go.

What you would say
“It sounds right but it’s wrong.”
“It’s making things up.”
“It’s citing a source that doesn’t say that.”
“It answered from the wrong document.”
“It was fine yesterday. Nothing changed.”
“It got worse and I don’t know when.”
“It’s stuck.”
“It keeps calling the same tool over and over.”
“It never stops.”
“It just won’t use the tool.”
“It says it’s calling the function but nothing runs.”
“It describes the tool call instead of making it.”
“It passes every test and falls over in prod.”
“It works in the playground, not in my app.”
“It works for me and not for them.”
“One customer says it’s broken and I can’t reproduce it.”
“The bill.”
“Spend tripled and traffic didn’t.”
“It’s right most of the time.”

"It fails maybe one time in ten."

"It worked when I ran it again."

"The JSON won't parse."

"It wrapped the output in explanation again."

"It's gone slow."

"The first token takes forever."

"It forgets what it was told."

"It ignores the instruction when the input is long."

"Something in the pipeline is wrong and I don't know which step."

"It did the thing I explicitly told it not to do."

"I've checked everything and I've got nothing."

If your symptom is not in that table, go to Chapter 1. It contains the four questions that come before every diagnosis, and about a third of the time one of them ends the investigation.

If you are here because something is happening in production right now and the cost is still running, go to Chapter 9, section *Stop the bleeding*, first. Then come back.

How to use this manual

This is a reference, not a course. It is built to be opened at the wrong page, read for ninety seconds, and closed.

Every chapter has the same shape.

You are here if — three or four lines. If none of them sound like your situation, you are in the wrong chapter and the line underneath says where to go instead.

First two minutes — the small number of actions that either fix the problem or cut the search space in half. Do these before reading further. They are ordered by how much they narrow things down per minute spent, not by how likely they are to be the answer.

Candidate causes — the things that produce this symptom, ordered by frequency multiplied by how cheap they are to test. Each one carries the same four lines:

- **What it is.**
- **Tell it apart** — a specific test, with a specific outcome, that separates this cause from the ones next to it. This is the part of the book you are paying for.
- **Confirms it / rules it out** — what you will see in each direction.
- **Fix.**

If none of the above — the honest ending. Sometimes the answer is that this chapter has nothing for you, and the chapter says so and tells you where the problem probably lives instead. A diagnostic that always finds something is not a diagnostic.

The durable part — a short closing note on what in the chapter will still be true when the tools have all changed.

The chapters are independent. You do not need to have read Chapter 5 to use Chapter 11. Where one chapter needs something from another, it says so and gives you the chapter number rather than assuming.

Two chapters are worth reading when nothing is broken. Chapter 2, because almost every diagnosis in this book requires a record of what actually happened, and the time to arrange that is not while you are on fire. And Chapter 17, because it is the method the rest of the book is an application of, and it is the part that will still work on failures nobody has catalogued yet.

Conventions

Cases are real unless labelled. Where a case has no label, it happened, it is public, and it is listed in *Sources* at the back with enough detail to find it. It has been read at source — not taken from a summary of it.

Cases marked [CONSTRUCTED] did not happen. They are invented to make a mechanism visible, and they are labelled every single time, in the heading. If it is not labelled, it is real. If it is labelled, no part of it is evidence of anything — it is an illustration.

Dated material is marked. Anything that depends on what a specific tool, model or API does right now carries a line like this:

> **Dated — September 2026.** Whatever follows was true when it was checked and may not be true when you read it. Check it.

Everything not marked that way is meant to outlast the current generation of tooling. That distinction is deliberate and it is the reason this book is organised by symptom rather than by product.

There are no web addresses in this book. Tools move, get bought, get renamed, disappear behind logins. A printed URL is a broken link with a publication date on it, and when a reader hits one they cannot tell whether the tool died or merely moved.

Documents, on the other hand, are stable, so sources are cited the way documents are cited — title, publisher, date, and for a code repository the project name and issue number. That is enough to find any of them and it will still be enough in five years.

Where this manual does not know, it says so. Phrases like *this is unverified* and *practitioners disagree about this* are not hedging. They mark the places where you should not act on this book alone.

What this manual will not do

It will not teach you to build an LLM application. It assumes you have one and that it is misbehaving. There are good books about building; this is not one, and a manual that also tried to be a tutorial would be worse at both.

It will not tell you which framework to use. Every chapter here applies regardless, because the failures in this book are not framework failures. They are failures of the seams between a system that is deterministic and a component that is not.

It will not make your application reliable. Diagnosis and design are different skills. This book will tell you why a thing broke and frequently that the answer is architectural, but it will not redesign your system for you.

It will not promise the cause is in here. Several chapters end by telling you the problem is somewhere this manual does not cover — your data, your infrastructure, your requirements, or an assumption you made so early you have stopped seeing it. That ending is a real result and it is written down as one.

And it will not pretend the model is always innocent. A body of advice has grown up insisting that LLM failures are always the application's fault. That is mostly true and it is not always true. Chapter 4 contains two documented occasions on which a model's behaviour changed underneath applications that had not been touched, both confirmed by the vendors in their own published postmortems. Knowing that this is possible — and knowing how rare it is, and how to tell it apart from the far more likely explanations — is part of the diagnosis.

A note on who wrote this

This book is published by Cynked Press under its own name, with no author pseudonym, because the expertise behind it is ordinary and real: it is written from daily work building and breaking these systems, and there is nothing to dress up.

The value here is not the author's authority. It is that every case has been read at source and every claim that could be checked has been. The manual will tell you when it is unsure. That is the only authority it asks for.

Chapter 1 — The four questions that come before everything

You are here if: you have a symptom and you do not yet know which chapter it belongs to. Or you are in a chapter and it is not working.

Four questions. They take about ten minutes together. They resolve a surprisingly large share of incidents outright, and when they do not, they cut the search down to something a single chapter can handle.

Ask them in order. The order matters — each one is cheaper than the next, and each one changes what the next one means.

Question 1 — Is it broken, or is it different?

The first question is whether there is a fault at all.

A conventional program that returns the wrong answer is broken. A sampling model that returns a different answer is doing what it does. Somewhere between those two sits the thing you are looking at, and you cannot diagnose it until you have decided which it is.

The test. Run the same input twenty times. Not once, not three times. Twenty.

Then count. You are looking for one of four shapes, and each one sends you somewhere different:

Fails 20 out of 20. This is deterministic. Whatever it is, it is not sampling variance, and it is almost certainly in your code rather than in the model's judgement — a malformed prompt, a parameter you did not mean, a field that is empty, a tool that is not registered. Deterministic failures are the good kind. Go to the chapter for your symptom and work the candidate list; you will find it.

Fails 0 out of 20 — you cannot reproduce it at all. Go to Chapter 8. The difference is in the input, the user, the environment, or the state, and the question is which.

Fails somewhere in between. This is the genuinely hard shape and it has its own chapter, 11. Do not treat it as a coin-flip to be re-rolled. An intermittent failure has a rate, that rate is a measurement, and the measurement is what you will use to tell whether your fix worked.

Fails 20 out of 20 but "used to work". Go to Chapter 4.

Why twenty. Because at five samples you cannot distinguish a one-in-ten failure from a one-in-three, and those are different problems with different causes. Twenty is not statistically luxurious; it is the smallest number that lets you say *roughly one in ten* rather than *sometimes*. If each run is expensive, run twenty anyway and consider it the cheapest twenty units of currency you will spend today.

And write the rate down. You will want it later, when someone asks whether the fix worked, and the honest answer will otherwise be "I think so."

Question 2 — What changed?

You will want to answer *nothing*. Nothing is never the answer. Something always changed; the question is only whether it was yours.

The reason this question gets skipped is that people interpret it as *did I deploy?* — and if they did not, they move on. But an LLM application sits downstream of a much longer list of moving parts than a conventional service does, and most of that list moves without you.

Things that change without a deploy:

- **The model.** A provider updated a version behind an alias you are pinned to, or rolled out a change to a version you thought was fixed. This is rarer than people claim and it is not zero; Chapter 4 has two documented instances.
- **The date.** Anything in your prompt that depends on today's date changes every day. This is a real and common fault and it is invisible in testing, because the test ran on a day when it still worked.
- **Your data.** A document was added to the corpus, a record was edited, a customer's account moved into a state your prompt has never seen.
- **Volume.** Nothing in the system changed except that there is more of it, and something that degrades under concurrency has started degrading.
- **Someone else's config.** A gateway, a proxy, a rate limit, a firewall rule, a budget cap, a feature flag, a caching layer, a dependency updated by a bot.
- **An upstream service the model calls through a tool.** Your application did not change. The API behind tool number six returned a new field, or started returning an error your handler swallows.
- **The prompt, via a template, a config file, or a system message edited in a console by a colleague** who does not think of that as a deploy, because it is not in the repository.

The test. Take the list above and put a timestamp against each one. Not "probably not" — an actual time of last change. Most of these are recorded somewhere. The ones that are not recorded anywhere are the ones to be suspicious of, because that is where change hides.

Then compare against the time the symptom started, which is a separate thing you need to establish and usually do not know as precisely as you think. "Since Tuesday" usually means "since someone noticed on Tuesday."

Question 3 — Can you make it happen again?

Reproduction is not a step towards the diagnosis. In this domain it very nearly is the diagnosis, because almost every technique in this book is a comparison between two runs, and you cannot compare against something you cannot produce.

What counts as a reproduction: a specific input, run through a specific path, that produces the symptom at a known rate.

What does not count: "it happens sometimes when users ask about refunds." That is a report, and a report is where you start.

Getting from a report to a reproduction is mostly a matter of getting the actual input. Not the user's description of what they asked — the bytes. Users paraphrase, and the paraphrase is almost always cleaner than the original: they drop the trailing whitespace, the pasted formatting, the emoji, the second question they tacked onto the end, the fact that this was message fourteen in a long conversation rather than message one.

That last one matters more than any other. A very large share of "I can't reproduce it" resolves the moment someone realises the failing case had twenty-five turns of history in front of it and the reproduction attempt had none.

If you genuinely cannot reproduce it, you are not stuck — you are in Chapter 8, which is about exactly that.

Question 4 — Are you looking at the real thing?

This is the question that ends the most investigations, and it is the one most often skipped, because it feels like a formality.

You are almost certainly not debugging the prompt you think you are.

Between the prompt you wrote and the tokens the model received there is a template engine, a variable substitution, a conversation history, a system message, a set of tool definitions serialised into a format you have never looked at, possibly a framework's own preamble that you did not write and cannot see, possibly a truncation step, and possibly a gateway that rewrites things.

Any one of those can be the entire problem. None of them are visible from the source code of your prompt.

The test, and it is the single highest-value action in this book:

> Print the exact, final, fully-rendered string — or the exact array of messages, byte for byte — that was handed to the API for the failing call. Then print the raw response object that came back, before any parsing.
>
> Read them both.

Not a summary. Not a truncated log line. Not the template. The thing that was actually sent, and the thing that actually arrived.

What you find when you do this is a small and depressingly consistent list:

- A variable rendered as an empty string, so an instruction reads *"You are helping with the account"*, and the model is improvising around a hole.
- A variable rendered as the literal text of its placeholder, because the substitution silently did not fire.
- Retrieved context that is empty, or is the wrong document, or is the right document truncated three sentences before the part that mattered.
- The conversation history containing a turn you did not expect, often an old error message that the model is now dutifully treating as instruction.
- A system message that is stale, because it is defined in a different file from the one you have been editing.
- Two instructions that contradict each other, added months apart by different people, neither of whom saw the other.
- Escaping damage — literal `\n` sequences, doubled braces, JSON encoded twice.
- The response was fine and your parser mangled it.

If you look at the rendered prompt and the raw response and they are both exactly what you intended, that is not a wasted ten minutes. That is a significant result: it moves the fault out of your assembly layer and into either the model's behaviour or your handling downstream, and roughly half of this book's candidate causes just became irrelevant.

What the four questions give you

Write down the four answers before you go further. They are what the rest of the manual takes as input:

1. **Deterministic, intermittent at a measured rate, or unreproducible.**
2. **What changed, with timestamps, and when the symptom actually started.**
3. **A reproduction, or an explicit note that you do not have one.**
4. **The rendered prompt and raw response — inspected, and either clean or not.**

Four lines. Most people skip straight past all four into the chapter they think their symptom belongs to, and then spend the afternoon testing hypotheses that question one would have eliminated in ten minutes.

If none of the above

If the four questions have left you with a deterministic, reproducible failure where the rendered prompt is clean, the raw response is clean, and nothing changed — you are in unusual and interesting territory. Two possibilities are worth holding:

It was always broken and nobody noticed. "It worked yesterday" is very often "nobody looked yesterday." Check whether the working behaviour was ever actually observed or

only assumed. This is extremely common with error paths, rare branches, and anything that only runs for one category of user.

The requirement changed rather than the system. Someone's expectation moved, or the definition of correct did, or a stakeholder saw the output properly for the first time. Nothing is broken. This resolves more incidents than anyone likes to admit and it resolves them faster than debugging does.

The durable part

These four questions are older than this subject. Reproduce it, bisect what changed, look at the real data rather than your model of it — that is how faults have been found in every kind of system for decades.

What the arrival of probabilistic components changed is only the first question. In a deterministic system, *is it broken or is it different* does not need asking; any deviation is a fault. Here it does, and getting it wrong in either direction is expensive: treat variance as a bug and you will chase it forever, treat a bug as variance and you will ship it.

Everything else in this book is these four questions applied to a particular symptom.

Chapter 3 — The output looks correct and is wrong

You are here if:

- The answer is fluent, specific, well-formatted and false.
- It invented a policy, a citation, a function, a price, a date, a person.
- It got a detail wrong inside an otherwise good answer, which is worse.
- Nothing errored. Nothing was logged. A human noticed.

Not here if: it cited a real source that genuinely says something else — that is Chapter 10, retrieval. Or if it was right last month and is wrong now — Chapter 4 first, then come back.

Before anything: this is a detection problem

Every other symptom in this manual announces itself. This one does not. The application returns a well-formed answer with a 200 and a normal token count and no exception anywhere, and the failure is discovered by a customer, or a regulator, or nobody.

So there are two investigations here and you need both:

- **Why was this answer wrong?** The rest of the chapter.
- **Why did we find out from a human?** The section *The detection problem* at the end. Do not skip it. The next wrong answer is already out there.

First two minutes

1. Get the rendered prompt (Chapter 1, question 4). You are looking for one specific thing: was the true fact present in the context at all?

2. Run the span test. This is the highest-yield test in the chapter.

> In a *separate* call — not a follow-up turn in the same conversation — give the model the same context and the original answer, and ask it to return the exact verbatim span from the context that supports the specific claim that was wrong. Instruct it to return nothing at all if there is no such span.
>
> Then take whatever it returns and search the context for it, as a literal string. Do not eyeball it. Search.

Three outcomes, three different chapters:

- **It returns nothing, or a refusal.** The fact was not in the context. The model generated it. Cause 1.
- **It returns a span that is not literally in the context.** The model generated the evidence as well. Cause 1, and more confidently — you have just watched it happen twice.

- **It returns a span that is literally in the context.** The context itself was wrong, stale, or contradictory. Cause 2, and Chapter 10.

The separate-call requirement matters. Ask inside the same conversation and the model is now defending an answer it can see, which is a different task with a different failure mode.

3. Re-ask with the correct fact inserted verbatim into the context, and see whether the answer changes. If it does not, you have a grounding failure rather than a knowledge gap, which is Cause 3 and a much more serious finding.

Candidate causes

Cause 1 — The fact was never in the context

The overwhelming majority of confidently wrong answers. The model was asked something it had no access to the answer for, and produced the most plausible completion instead. Not lying, not malfunctioning — doing the thing it does, against a gap you did not know was there.

The reason this is so easy to miss is that the gap is usually partial. Nine facts were retrieved and the tenth was not, so the answer is ninety per cent sourced and ten per cent invented, and it reads as a single coherent piece because it is one.

Tell it apart: the span test above. No supporting span, or a fabricated one.

Also tell it apart: search the rendered context for the specific true value as a literal string. If the string is not there, the model could not have used it. This takes five seconds and is oddly often skipped.

Confirms it: the true fact is absent from the rendered context, and the model cannot produce a span.

Rules it out: the fact is right there in the context, correctly, in a position that was not truncated.

Fix: get the fact into the context, or make the absence produce an abstention rather than an answer. The second half is the part people skip and it is the part that matters, because you will never close every gap. A system that cannot say *I do not have that* will improvise every time it does not have something, forever.

Cause 2 — The fact was in the context and the fact was wrong

Your source of truth is stale, or duplicated with two versions disagreeing, or correct-but-superseded. The model faithfully reported it. The model is not the broken component.

Tell it apart: the span test returns a real span, and you read the span and it is wrong.

Confirms it: the same wrong value appears in your data store.

Fix: in your data, not your prompt. And check for duplicates — two documents saying different things is a distinct failure from one document saying the wrong thing, because no prompt engineering resolves it. See Chapter 10.

Cause 3 — The fact was present and the model contradicted it

Rarer than Cause 1, far more alarming, and the one that most often indicates something structurally wrong rather than a gap to be filled. The correct information was in the context and the answer says otherwise.

Common reasons this happens, in rough order:

- **Position.** The fact was buried a long way into a long context, competing with a great deal of other material. See Chapter 14.
- **Contradiction.** Something else in the context says the opposite — often a system instruction written months ago, or an earlier conversational turn, or a second retrieved document. The model resolved a conflict you did not know it was being handed. It had to pick one.
- **Parametric pull.** The fact in your context contradicts what is overwhelmingly common in the world. If your product genuinely has a thirty-day return window in a category where everything has fourteen, expect to have to say so emphatically and to check that saying so worked.
- **Framing.** The instruction says *answer the user's question*, and the retrieved material is framed as background rather than as authority. The model is weighing them and you did not tell it how.

Tell it apart: re-run with everything removed from the context except the instruction, the question and the single correct fact. If it now answers correctly, the fault is contention — something else in that context was winning, and you can bisect the context to find what. If it still answers wrongly with nothing but the fact in front of it, the fault is in the instruction or the framing, and it is repeatable and easy to iterate against.

That bisect — halve the context, re-run, halve again — is the most mechanical and most reliable procedure in this chapter. It usually takes four or five runs to find the offending passage.

Fix: depends which. Contradictions get removed, not out-argued. Position gets fixed by putting the authority near the instruction. Parametric pull gets fixed by stating the unusual fact explicitly as a correction rather than merely including it.

Cause 4 — It was in the context and then it was not

Truncation. The fact was retrieved, assembled, and then silently cut before the call, or the model's output stopped before it got there.

Tell it apart: two numbers. Compare the input token count in the response against the token count of the context you believe you sent. Then check the stop reason field — if generation ended because of a length limit rather than because the model finished, the answer is not wrong, it is *incomplete*, and an incomplete answer read as a complete one looks exactly like a confident error.

Confirms it: a length-based stop reason, or an input token count materially below what you assembled.

Fix: raise the limit, shorten the context, or handle the length stop reason as the error it is. That last one is free and almost nobody does it.

Cause 5 — The schema made it answer

You asked for structured output with a required field, and the model did not have the value, and a required field must be filled. So it was filled.

This is a designed-in confabulation and it is extremely common in extraction work. Nothing about a required string field communicates *leave this out if you do not know*; the schema says the opposite.

Tell it apart: does the schema permit absence — null, or an optional field, or a sentinel? If not, this is a live candidate for every field in it.

Confirms it: make the field optional, re-run, and watch the invented values stop appearing.

Fix: every extracted field gets a representable *not found*. Then, and this is the part that gets forgotten, check that your downstream code handles it — because it will start arriving, and if the reason the field was required in the first place was that something downstream could not cope with a null, you have merely moved the failure.

Cause 6 — The user's premise was false and the model accepted it

Ask *why does the export fail on files over 4GB* and you will often get a fluent explanation of a 4GB limit that does not exist. The question asserted it; the answer elaborated it.

This is the most dangerous cause in customer-facing systems, because the user supplies the error and then receives it back with authority attached.

Tell it apart: re-ask the question with the premise removed — *does the export have a file size limit* — and then re-ask with a deliberately absurd premise — *why does the export fail*

on files over 3 bytes. If the second one also produces a confident mechanism, you have confirmed it in one call.

Fix: an explicit instruction to check the premise before answering it helps and does not solve it. What solves it is grounding the answer in retrieved material and requiring a citation, which makes the absence of support visible.

> **Dated — September 2026.** How strongly a given model resists a false premise varies between models and between versions of the same model, and is one of the things that moves when a provider updates. If this matters to your application, it belongs in your evaluation suite as a named test rather than as an assumption.

Cause 7 — The answer is right and your check is wrong

Before the afternoon disappears: is it actually wrong?

Tell it apart: find the authoritative source yourself, by hand, without the model. Read it.

This is worth the two minutes more often than pride suggests. Ground truth in evaluation sets rots. Requirements change without the tests changing. The "correct" answer in a test fixture was frequently written by whoever wrote the test, from memory, at speed.

How to tell them apart, in one grid

Test	Cause 1	Cause 2	Cause 3	Cause 4	Cause 5
Span test returns a real, findable span	no	**yes**	yes	no	no
True fact present in rendered context	no	wrong version	**yes**	assembled but absent	no
Answer changes when fact is inserted alone	yes	yes	**no**	yes	yes
Length-based stop reason, or short input count	no	no	no	**yes**	no
Field is	maybe	no	no	no	**yes**

required by
the schema

Work down the left column. The first row usually decides it.

Case — an airline chatbot and a policy that did not exist

Moffatt v. Air Canada, 2024 BCCRT 149, British Columbia Civil Resolution Tribunal, decided 14 February 2024.

In November 2022 a customer, booking flights on the day their grandmother died, asked an airline's website chatbot about reduced bereavement fares. The chatbot told him the reduced rate could be requested within ninety days of the ticket being issued, including after travelling, by submitting the refund application form. They booked, flew, applied within the window, and were refused: the airline's actual policy did not allow claims after travel was completed. The tribunal found negligent misrepresentation and ordered the airline to pay CAD $650.88 in damages plus interest and fees, $812.02 in total.

Three things in the decision are worth a diagnostician's attention.

The wrong answer was extremely specific. It named the correct form and gave a concrete number of days. Fluency checks, tone checks and plausibility checks all pass on an answer like that. Specificity is not accuracy, and in this domain specificity is generated with exactly the same machinery as everything else.

The correct information was one click away, inside the wrong answer. The tribunal records that the chatbot's reply contained a hyperlink to the airline's bereavement travel page — the page that contradicted it. The source of truth was not missing. It was not even distant. It was attached to the falsehood, and attaching it made the falsehood more convincing rather than less.

That is the lesson most worth taking. Shipping a citation alongside an answer is not grounding. Unless something verifies that the answer follows from the citation, a link is decoration, and a decorated wrong answer is harder to catch than a bare one.

And the "check it elsewhere" defence failed. The airline argued the customer could have found the correct information on another part of its website. The tribunal asked why one page should be considered inherently more trustworthy than another, and why a customer should have to check one part of a website against another part of the same website. It also rejected, in notably direct terms, the argument that the airline was not responsible for what its chatbot said.

Whatever the law is where you are, the design principle travels: *the user will check this elsewhere* is not a control. Nobody does, and no adjudicator seems inclined to think they should.

Case — a support bot explains a real bug with a false cause

Cursor / Anysphere, April 2025.

Users of a code editor began being logged out unexpectedly. They contacted support, and an AI support agent — replying by email under a human first name — told them this was expected: their subscription permitted only one active session at a time. Developers working across a laptop and a desktop read this as a deliberate product change, and some cancelled.

There was no such policy. A co-founder of the company stated publicly that the restriction did not exist and that customers were free to use the product on multiple machines. The real cause of the logouts was a race condition affecting slow connections, which was generating unwanted sessions. The company began labelling AI-generated support replies explicitly and refunded affected customers.

This case earns its place because of a structural feature the previous one does not have: **there was a real bug underneath.**

The model was asked to explain behaviour that was genuinely occurring. It produced a mechanism that accounted for the observation completely — the user was logged out on the second device, and a single-session policy explains that perfectly. The symptom corroborated the explanation. Every party to the conversation had evidence the answer was right.

A wrong answer contradicted by reality gets caught. A wrong answer that *predicts* reality does not. When you are triaging a report of a confidently wrong output, ask early whether the model was explaining something that was actually happening, because that is the class where your own reviewers will agree with the error.

The second feature is the blast radius. The failure was not a bad paragraph. It was cancelled subscriptions, a public apology and a support channel that had to be relabelled. When you are deciding how much to invest in detection, the question is not how often the answer is wrong. It is what the wrong answer authorises someone to do.

The detection problem

You found this one because a human told you. The others are in the logs.

Four things, in order of what they cost:

1. Require a citation and verify it mechanically. Not a link — a verbatim span. Have the model return the exact text it relied on, then check by string search that the span exists in the context you supplied. This is cheap, deterministic, and catches Cause 1, which is most of them. It is the span test from *First two minutes*, run automatically on every response instead of once during an incident.

2. Make abstention possible, then measure it. If your abstention rate is zero, your system has never once admitted it did not know, which is not a property of the world. A rate of exactly zero is a finding, not a triumph.

3. Sample and read. A person reading twenty outputs a week finds things no automated check is looking for, because automated checks only find what someone thought to look for. See Chapter 17 — this is the same lesson a major vendor learned publicly about its own evaluation suite.

4. Make it easy to report. The Air Canada case took fifteen months from the misleading answer to the decision. The February email in evidence shows the company had already conceded the chatbot's wording was misleading. Somewhere in there, the signal existed and did not reach anyone who could act on it.

If none of the above

If the true fact was present, in a short context, with no contradictions, the schema allows absence, the premise was clean, and the answer is genuinely wrong — then this chapter is done and the problem is one of two things this chapter does not cover:

The task is beyond the model as configured. Multi-step arithmetic, precise counting, reasoning over many interacting constraints, anything requiring an exact lookup across a large set. These fail in a characteristic way: the shape of the answer is right and the values are wrong. The fix is not a better prompt. It is moving the work out of the model — do the arithmetic in code, do the lookup in a database, and let the model choose and phrase rather than compute.

Or the question does not have one answer. If two competent humans would answer differently, the model's answer is not wrong, and your evaluation is measuring agreement with one person's opinion. That is a legitimate thing to measure, but it is not correctness, and treating it as correctness will send you chasing a fault that does not exist.

The durable part

A generative model has no mechanism that distinguishes *recalling* from *producing*. Both are the same operation. Nothing in the output marks which one happened, and confidence in the text is not correlated with the difference.

That is not a defect of the current generation of models and it is not being engineered away; it is what the thing is. Every technique in this chapter — grounding, span verification, abstention paths, optional schema fields, premise checks — is a way of adding an external mechanism that the model does not have.

Which means the durable question, on any system, in any year, is this: **for each fact in this output, what checked it?** If the answer is *the model was confident*, nothing checked it.

Write down the answer for the facts that carry real consequences, and you will know where the next Air Canada is in your application before a tribunal does.

End of sample

That is the front matter, Chapter 1 and Chapter 3 — about a sixth of the book.

The full manual continues with every other symptom, each built the same way: the recognition lines, the first two minutes, the candidate causes, **the test that tells each cause apart from the one next to it**, and the honest ending when the chapter has nothing for you.

Still to come

- Chapter 2 — You cannot debug what you cannot see
- Chapter 4 — It worked yesterday
- Chapter 5 — The agent is looping
- Chapter 6 — The tool call never fires
- Chapter 7 — It works in testing and fails in production
- Chapter 8 — It works for me and not for them
- Chapter 9 — Costs are spiking
- Chapter 10 — It answered from the wrong source
- Chapter 11 — It fails only sometimes
- Chapter 12 — The output will not parse
- Chapter 13 — It is too slow
- Chapter 14 — It forgets, or ignores the middle
- Chapter 15 — Something in the pipeline is wrong and you do not know which step
- Chapter 16 — It did something it was told not to do
- Chapter 17 — How to reason about a failure you have never seen
- Chapter 18 — This is not your problem

And the appendices: the bisect protocol, the minimum trace record, a fast check for every symptom, the pre-incident checklist, and a one-page triage.

Including two vendor postmortems in which models changed underneath applications nobody had touched — one where the evaluations passed, the A/B test passed, and the only instrument pointing at the real problem was a tester saying something felt slightly off.