THE SKILL NOBODY TEACHES AND EVERYONE NEEDS

Debugging AI Systems

Non deterministic output, no stack trace, five components that can each fail silently. Here is how to find the actual problem.

Why This Is Different

FRAMING

Normal software fails loudly at the line that broke. AI systems produce a fluent, confident, wrong answer and no error at all.

NORMAL SOFTWARE

- Fails with a stack trace
- Same input, same output
- The bug is in one place
- A breakpoint shows you the state

AI SYSTEMS

- Fails with a plausible answer
- Same input, different output
- Five components can each be at fault
- There is nothing to breakpoint

SO THE METHOD HAS TO CHANGE

- Inspect every stage, not the end result
- Measure distributions, not single runs
- Bisect the pipeline before theorising
- Log enough to reconstruct any request

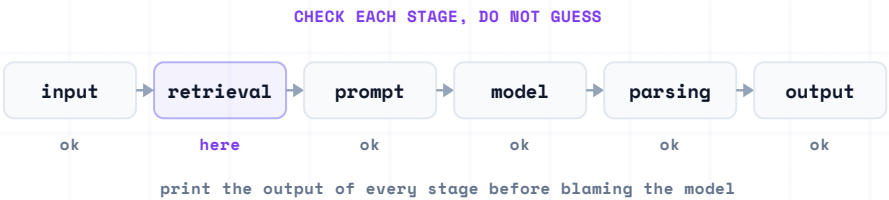
> The hardest part is accepting that there is no line number. You are debugging a distribution, not an execution.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Bisect the Pipeline

FIRST MOVE

Before forming any theory, print the output of every stage. The bug is almost always visible, and almost never where people first look.



STAGE	WHAT TO PRINT
Input	The exact string after your own preprocessing
Retrieval	The chunks, their scores and their sources
Prompt	The fully assembled text, not the template
Model	The raw response and the stop reason
Parsing	What your code extracted from that response

> Nine times out of ten the answer is obvious the moment you print the assembled prompt. Teams skip it because it feels too simple.

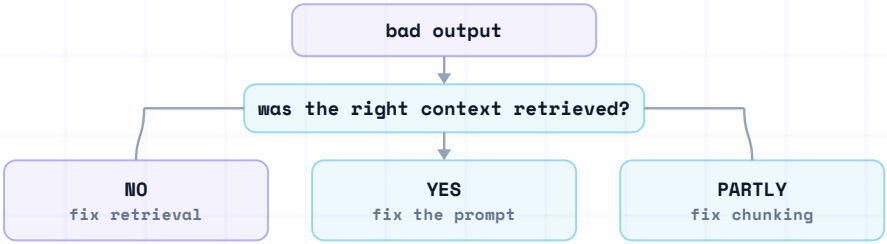
↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

Triaging a Bad Answer

DECISION TREE

For RAG there is one question that splits every bug into three buckets. Ask it before you touch anything.



one question splits every RAG bug into three buckets

ANSWER	MEANS	FIX
Not retrieved	Retrieval failed	Chunking, hybrid search, reranking
Retrieved, ignored	Prompt failed	Grounding instructions, context order
Retrieved partially	Chunking failed	Bigger chunks, overlap, parent linking

> Most teams jump straight to prompt engineering. Two thirds of the time the passage was never retrieved.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

Retrieval Bugs

MOST COMMON

The majority of RAG complaints are retrieval problems wearing a generation costume. These are the checks in order of how often they find it.

CHECK	FINDS
`Search the chunk text directly`	The passage was never indexed at all
`Print scores with results`	A threshold quietly filtering everything
`Try the exact keyword`	Vector only search missing an identifier
`Inspect chunk boundaries`	The answer split across two chunks
`Check the metadata filter`	A tenant or date filter excluding it
`Confirm the embedding model`	Index and query using different models

Try the exact keyword before anything else

Confirm one embedding model, both sides

> The last one is silent and devastating. Nothing errors, recall just quietly collapses and every score still looks plausible.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

Prompt Bugs

ASSEMBLY

The template looks right in your editor. What actually reached the model is a different string, and that difference is the bug.

```
# the single most useful line in AI debugging
print(json.dumps(messages, indent=2))

# and the one that catches truncation
print(resp.stop_reason, resp.usage)
```

SYMPTOM	CAUSE
Instructions ignored	Buried in the middle of a long context
Empty context section	The template variable never got filled
Answer cut off	stop_reason was max_tokens, not end_turn
Format ignored	No example of the format in the prompt
Wrong language	A stray instruction earlier in the history

> A truncated answer looks like a complete answer to your parser. Checking the stop reason costs one line and catches a whole bug class.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

When It Is Actually the Model

RARE

Sometimes it really is the model. These are the signatures, and each one has a specific fix that is not prompt fiddling.

SIGNATURE	FIX
Confident, specific, wrong	Ground it, do not ask it to be careful
Ignores half a long context	Move key content to the start or the end
Refuses a reasonable request	Rephrase the framing, check the system prompt
Output drifts between runs	Lower temperature, or constrain with a schema
Broke after a provider update	Pin the model version, then test the new one

Temperature 0 first, to remove a variable

Pin the model snapshot in production

> Blaming the model is the last step, not the first. It is genuinely the cause far less often than it feels during an incident.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Reproduce It First DISCIPLINE

You cannot debug what you cannot reproduce. With a non deterministic system, reproduction means capturing the whole input state.

```
def capture(req, resp):
    store.save({
        'messages': req.messages,        # fully assembled
        'model': req.model,
        'params': req.params,           # temperature, top_p, seed
        'retrieved_ids': req.chunk_ids,
        'response': resp.text,
        'stop_reason': resp.stop_reason,
        'ts': now()})
```

Capture the assembled messages, not the template

Timestamp everything, providers change under you

> Set temperature to 0 and replay. If it still misbehaves you have a real bug. If it does not, you have a variance problem.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Run It Ten Times

VARIANCE

A single bad output tells you almost nothing. Ten runs of the same input tell you whether it is a bug or a tail.

```
outs = [run(q) for _ in range(10)]
ok   = sum(is_acceptable(o) for o in outs)
print(f'{ok}/10 acceptable')
```

RESULT	DIAGNOSIS
0 of 10	A real, deterministic bug, go find it
3 of 10	Ambiguous prompt or weak retrieval
8 of 10	A variance problem, tighten the constraints
10 of 10	Not reproducible, look at the specific input

> This single habit prevents the most common wasted afternoon in AI engineering: fixing a bug that was actually a one in ten sample.

- Ten runs before you believe anything
- Then fix the pattern, not the sample

One bad output is an anecdote. Ten runs is a measurement.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Ablate the Prompt ISOLATION

Remove one piece at a time and measure. It is the only reliable way to find out which part of a long prompt is doing the work.

```
variants = {
  'full':        base,
  'no_examples': base.replace(EXAMPLES, ''),
  'no_persona':  base.replace(PERSONA, ''),
  'no_caveats':  base.replace(CAVEATS, ''),
}
for name, p in variants.items():
    print(name, score_on_evalset(p))
```

- Sections that change nothing can be deleted
- That is a cost saving as well as a clarity win
- Change one thing per run, never two
- You need an eval set for this to mean anything

> Most long prompts contain sections that make the engineer feel safer and change nothing the model does. Ablation is how you find them.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Debugging a Loop AGENTS

An agent failure is a sequence, not a moment. Without the full trace you are guessing at which of twelve steps went wrong.

SYMPTOM	USUAL CAUSE
Loops forever	Termination condition never satisfiable
Calls the wrong tool	A vague tool name or docstring
Repeats the same call	No memory that it already tried it
Gives up too early	max_iter too low for the task
Wanders off task	The goal is vague, or the context got lost
One bad step, then nonsense	No validation between steps

The trace is the only real debugging surface

Check the iteration cap before anything else

> Log every step: the thought, the tool, the arguments and the result. An agent bug is obvious in the trace and invisible in the output.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

When It Only Fails in Prod

ENVIRONMENT

It works locally and fails for users. The cause is almost always one of these five, and none of them are the model.

DIFFERENCE	EFFECT
Different model version	Provider updated the snapshot under you
Different index	Production has documents your laptop does not
Longer conversation history	Context window pressure, truncation
Real user phrasing	Typos, slang, multiple languages
Concurrency	Rate limits, timeouts, partial failures

- Pin the model version in every environment
- Test against a production sized index
- Replay real user inputs, not your own phrasing
- Load test before you trust the latency numbers

> Works on my machine has a specific meaning here: your index is smaller, your history is shorter and your phrasing is cleaner.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

What to Instrument

OBSERVABILITY

Decide this before the incident. Every field here has been the one thing someone needed at 2am.

LOG	ANSWERS
Assembled prompt	What did the model actually see
Retrieved ids and scores	Was the right context even present
Model, version, parameters	Did behaviour change with a release
Stop reason and token usage	Was it truncated, what did it cost
Every tool call and result	What did the agent actually do
Latency per stage	Which component is slow

- Redact PII before it reaches the logs
- Make traces searchable by user and request id
- Keep enough history to investigate weeks later
- Correlate by a request id across every service

> Decide what to log before the incident. Afterwards, the one field you did not capture is always the one you needed.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

Debug With an Eval Set

MEASUREMENT

Without a test set you are debugging by anecdote. Fifty labelled examples turns every change from an opinion into a number.

```
# 50 questions with known good answers
before = evaluate(system, testset)
# ... make one change ...
after  = evaluate(system, testset)
print(before, after, after - before)
```

- Fifty examples is enough to start today
- A fix that helps one case often breaks three
- Add every reported bug as a new test case
- Track the score over time, not per change

> The most expensive habit in AI engineering is fixing individual complaints without a test set, then discovering months later that quality never moved.

Fifty labelled examples turns opinion into arithmetic.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Debugging the Bill

SPEND

Cost bugs are silent until the invoice. These are the four that account for most surprise spend.

CAUSE	SIGNATURE
No iteration cap	A few requests costing hundreds of times the median
Full history resent	Cost growing linearly with conversation length
Retrieval too wide	Large prompts on every single request
No cache	Identical questions billed repeatedly

```
# log this on every call, then group by feature
log.info('cost', extra={'in': u.input_tokens,
    'out': u.output_tokens, 'feature': feature})
```

> Track the p99 cost per request, not the average. The average hides exactly the runaway cases that produce the bill.

- Alert on the p99, not the mean
- Cap iterations before you cap anything else

Attribute cost per feature, or you cannot manage any of it.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Debugging Slowness PERFORMANCE

Latency complaints are usually one component, and it is usually not the one people assume.

STAGE	TYPICAL	SUSPECT IF
Embed query	20 to 50 ms	Over 200 ms, network or batching
Retrieval	5 to 50 ms	Over 300 ms, missing index
Rerank	50 to 200 ms	Over 1s, running on CPU at scale
Generation	1 to 4 s	Dominates, so shorten the output

- Time every stage before optimising any of them
- Stream the answer to fix perceived latency
- Shorter outputs beat every other optimisation

> Generation dominates the budget. Optimising retrieval to save 30ms while the model writes for three seconds is wasted effort.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

Ten Practical Tips

FIELD NOTES

The habits that separate an hour of debugging from a week of it.

- Print the assembled prompt first, every time
- Set temperature to 0 while investigating
- Run it ten times before believing one result
- Bisect the pipeline, do not theorise
- Pin the model version everywhere
- Keep a fifty case eval set from day one
- Log the retrieval ids and scores
- Change one thing per experiment
- Turn every incident into a test case
- Read a hundred real user inputs by hand

> If you adopt only two of these, print the prompt and keep an eval set. Those two find and prevent most of what the rest catch.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

What Wastes Time

MISTAKES

Six habits that feel productive and are not. Most people do at least three of them.

ANTI PATTERN	WHY IT FAILS
Adding please to the prompt	Politeness is not a control mechanism
Switching models first	Hides the bug instead of finding it
Fixing one complaint at a time	No idea what else you broke
Blaming the model immediately	It is retrieval or the prompt far more often
Debugging on one example	Non determinism makes it meaningless
Adding more instructions	A longer prompt gets less attention, not more

THE ONE THAT COSTS THE MOST

Debugging on a single example

With no eval set to check against

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

The Order to Check

CHECKLIST

When something is wrong and you do not know why, work down this list. It is ordered by how often each step finds the problem.

STEP	CHECK
1	Print the fully assembled prompt
2	Print the retrieved chunks with scores
3	Check the stop reason and token usage
4	Set temperature to 0 and rerun
5	Run it ten times and count the failures
6	Search for the answer text directly in your corpus
7	Compare against your eval set, not your memory
8	Only now consider the model itself

↓ Work down, do not skip

Steps 1 to 3 take five minutes and usually find it

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

Making It Everyone Job

TEAM

Debuggability is a property you build in, not a skill you apply afterwards. Four things make it a team capability rather than one person heroics.

- A trace viewer anyone can open, not just engineers
- A shared eval set that reviews block on
- A written triage order, like the one on the last slide
- Incidents written up with the trace attached

> The teams that ship reliable AI are not the ones with better prompts. They are the ones who can answer why did it do that in under ten minutes.

ARTEFACT	MAKES DEBUGGING
A trace viewer	Possible for non engineers
A shared eval set	Objective instead of anecdotal
A written triage order	Repeatable under pressure
Incident write ups	Cumulative instead of forgotten

Debuggability is designed in, never bolted on afterwards.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT

PRACTICAI

The Debugging Toolkit

- Print the prompt
- Bisect the pipeline
- Retrieval triage
- Ten run variance
- Temperature 0
- Prompt ablation
- Agent traces
- Eval sets
- Cost tracing
- Latency per stage

**Anyone can say the model is wrong.
Engineers can point at
the stage that broke.**

Save this before your next production incident.

↓ CHECK THE CAPTION FOR THE FULL INTERVIEW KIT