It Was Never the Model

A post-mortem on Oracle of Delphi: four failures in a local voice assistant, none of them in the language model.

Abir Deol · University of Alberta, Computing Science · August 2026

Abstract. Oracle of Delphi is a fully local voice assistant: five cooperating processes, a language model running on my own GPU, and a Three.js interface in a native window, all behind one double-click. Getting it working was almost entirely a debugging project. This is a post-mortem on four of the failures, chosen because each one lived at a different layer of the stack and each one presented as something other than what it was. The through line is that none of them were in the language model, including the one where the model started answering in Thai.

1. The system

The assistant runs as a small fleet rather than one program, and the split is deliberate. A native shell owns the window, the global summon hotkey and the tray icon. An orchestrator runs the agent loop, serves the interface, and supervises two hidden children: the language model server and a privileged actuator daemon. The daemon is the only process permitted to touch the operating system.

```
oracle-of-delphi.exe        the window, the hotkey, the tray
      \__ oracle-core.exe        agent loop, serves the HUD
            |__ llama-server.exe  the model, port 8080
            |__ oracle-actd.exe   OS control, over a named pipe
```

That isolation exists for a specific reason. The model is treated as an untrusted planner: it proposes actions, it does not perform them. Anything irreversible, killing a process or running a shell as the user, is gated behind an explicit confirmation before the daemon will carry it out. A planner that can be talked into proposing something destructive is a nuisance. A planner wired directly to the operating system is an incident.

Four processes with a supervision tree, a named pipe, a local HTTP server and a WebSocket is a distributed system living on one machine. It fails like one too, which is most of what follows.

2. The interface that loaded blank

Symptom. The window opened onto nothing. No error, no partial render, just a blank page where the interface should have been.

What it looked like. A front end problem. That is the natural reading when a page comes up empty, and I spent time in the browser console and the bundler config on that assumption. Neither had anything to say.

Root cause. The orchestrator serves the interface bundle over a raw TCP socket it manages itself. It was writing the response and then closing the socket while the client still had unread request bytes in flight. On Windows, closing a socket with unconsumed data in the receive buffer sends a TCP reset rather than a

graceful shutdown, and a reset truncates whatever the peer had not yet read. The 500KB JavaScript bundle was arriving cut off. A truncated script does not execute, and nothing renders.

Fix. Drain the incoming request fully before responding, then shut the write half down gracefully instead of dropping the socket. Verified by byte count: 497,862 of 497,862 arriving intact.

What it changed. The bug was invisible from the layer where the symptom appeared. Nothing in the browser could tell me the response was short, because as far as the browser was concerned the connection ended normally. The lesson I took is to check the boundary rather than the endpoints: when two components disagree about reality, the interesting thing is usually the transport between them, not either side.

3. The daemon that would not connect

Symptom. Every operating-system feature reported that the actuator daemon was offline, even though the orchestrator was supposed to launch it automatically.

The tell. It worked when I launched the daemon myself, and failed only when the shell auto-launched it. Identical binary, identical arguments, different outcome. That is the shape of an environment difference, not a logic error, and it is the single most useful clue in this document.

First cause, and a red herring. An orphaned daemon from a previous run was still holding the named pipe, so the newly launched one failed to bind and gave up. I fixed that: the orchestrator now reaps strays before launching, and the daemon retries binding for a few seconds while a just-killed orphan releases its handle. The symptom persisted, which meant the real bug had been hiding behind this one.

Actual root cause. The first thing the daemon does on startup is open its audit log. It tried to place that log next to its pipe path, but a Windows pipe path has no real parent directory, so it fell back to the current working directory. Launch it from my project folder and that is writable, so it worked. Let the shell launch it from a shortcut and the inherited working directory was not writable. Opening the log threw, and the error propagated all the way out of startup and killed the process before it ever bound the pipe.

```
daemon start
  |__ open audit log  ->  cwd not writable  ->  Err
  |__ (never reached)  bind named pipe

orchestrator's view: pipe absent  ->  "daemon offline"
```

Fix. Two changes, and the second matters more than the first. The audit log can no longer take down the daemon: if it cannot open, the daemon warns and serves anyway, logging to a null sink in the worst case. Separately, the orchestrator now tells the daemon exactly where to keep its log, in a runtime directory that is always writable, so it normally never reaches the fallback at all. And when the daemon does fail to start, the orchestrator reads the tail of its log and reports the real reason instead of a generic offline message.

What it changed. This is the one I think about. A logging failure took down an entire service, which is a category error: observability is supposed to describe the system, not be load-bearing inside it. Anything that exists to tell you what happened should degrade rather than abort.

- Inherited environment is real state. Working directory, environment variables and handles come from whoever launched you, and "works when I run it" is a statement about my shell, not about the program.
- Never let a diagnostic path be fatal. If the log cannot open, warn and continue.
- A supervisor that reports "child is not there" without saying why is hiding the answer it already has. Read the child log and surface the real error.
- Fixing a real bug that does not fix the symptom means there were two. That is information, not failure.

4. The tools that never worked

Symptom. The model could not reliably call its tools. It emitted garbage, or half-formed calls, or a pile of nameless fragments that surfaced in the interface as rows of errors.

What it looked like. A model that was not good enough at tool use. This is the reading I want to flag, because it is comfortable and it is wrong, and it was wrong twice in the same bug.

Root cause, first layer. The model server needs an explicit flag to apply the model's own chat template. Without it the template is never applied, and the template is precisely what teaches the model the tool-call format. The model was not failing at tool calling. It had never been shown the format.

Root cause, second layer. With the template applied, the model streams a tool call in pieces: the function name arrives first, then the arguments dribble in as string fragments across many messages. My parser treated each fragment as a separate, nameless call. One correct tool call was being rendered as a dozen broken ones.

Fix. Accumulate fragments by index, keeping the name and concatenating the argument string, and emit exactly one whole call when the turn finishes. Pinned with a test that replays a real captured fragmented stream and asserts it collapses to a single correct call.

What it changed. Both layers were mine, and both presented as model quality. There is a strong pull, when a component is probabilistic, to attribute any bad output to the probability rather than to the deterministic code around it. That attribution is almost always the lazier of the two available explanations, and it is unfalsifiable in a way that quietly ends the investigation. The test replaying a captured stream matters for the same reason: it turns the part I control back into something with a right answer.

5. The assistant that started speaking Thai

Symptom. Ask a plain English question and the reply would occasionally come back as a fluent sentence in Thai. Not garbage: a coherent, correct translation of what it meant to say, in the wrong language. Often paired with a stall, an offer to check my inbox and then no result.

What it looked like. The clearest possible case of the model being broken. Nothing in a deterministic program produces fluent Thai by accident.

Root cause. Two layers again, and the smaller one first. I was sending the model server almost no sampling parameters beyond temperature, so it ran on wide-open defaults and the long tail of the token

distribution stayed open. Tightening that to the model's recommended window, along with an explicit instruction to answer in English, helped but did not close it.

The decisive cause was simpler: temperature was 0.7. At that heat, in the low-confidence moment right before a tool call, the sampler had enough room to pick a plausible-but-unusual continuation, and a language switch is exactly that. The model was not confused about what to say. It was being asked to be creative at the precise moment it should have been obedient.

Fix. Temperature to 0.3, one line in a config file, no rebuild. At that setting the distribution sharpens hard toward the obvious next token, and given English input and an English prompt the obvious next token is English. The language flips stopped.

What it changed. This is the best example I have of a symptom pointing at the wrong component with total confidence. Fluent Thai looks like a model problem in a way that nothing else in this document looks like anything. It was a number in a configuration file. Temperature is not a quality dial, it is a decision about how much of the distribution you are willing to sample from, and an assistant that calls tools needs a narrow one.

6. The pattern

Four failures at four layers: a transport that truncated a response, a process killed by its own logger, a parser that misread a stream, and a sampler set too wide. Every one of them presented as something else, and two of them presented specifically as the language model being inadequate.

That is the thing worth carrying out of this project. A system built around a probabilistic component gives you a permanent, always-available, always-plausible explanation for any bad behaviour, and reaching for it ends the investigation before it starts. Not one of these was a model problem. They were a missing flag, a wrong directory, an unread socket buffer, and a number in a config file. Specific, findable, fixable things.

- When the symptom points at the component you understand least, distrust the pointer. It is the easiest place for an explanation to hide from evidence.
- The difference between "works when I run it" and "fails when it is launched" is the environment, and the environment is state you did not write down.
- Diagnostics must never be able to kill the thing they observe.
- Confidence in a wrong hypothesis costs more than having no hypothesis. Fluent Thai felt like an obvious diagnosis, and that feeling was the expensive part.

The system underneath was sound the whole time. That is usually how it goes, and it is worth believing earlier than I did.