Well-formed and wrong

Why checking an AI's instructions against a schema is not enough

Munawar Kazmi • munawarkazmi.com

A plain-language guide to **toolcall-contract**, an open validator for the commands language models send to real systems.

Code, datasets, and every figure: github.com/munawarkazmi/toolcall-contract

---

When an AI presses the buttons

Modern AI assistants do not only write text. They operate things. When you ask an assistant to book a meeting, adjust a thermostat, or move a robot, the model does not touch the machinery directly. It fills in a form.

The form is called a tool call, and it looks like this:

```
{"tool": "navigate_to_waypoint", "arguments": {"waypoint": "dock"}}
```

A name for the thing to do, and the values to do it with. Some other piece of software reads that form and actually drives the robot.

So the interesting question is not whether the model writes good prose. It is whether the form it filled in is one you would be happy to act on.

Everyone checks the shape of the form

There is a standard way to check these forms, and essentially every AI system that uses tools does it. It is called schema validation, and it asks structural questions:

- Is this valid, parseable data at all?
- Is `navigate_to_waypoint` a tool that actually exists?
- Does it have the fields it is supposed to have, and are they the right types? A number where a number belongs, text where text belongs, nothing missing.

These checks are real and worth doing. They are also, it turns out, almost never the thing that goes wrong.

Consider this call, produced by a real model when asked to pick up a green cylinder:

```
{"tool": "pick_object", "arguments": {"object_id": "green_cylinder",
                  "gripper_force": 20}}
```

It parses. The tool exists. Every field is present and correctly typed. It passes schema validation completely.

There is no green cylinder. It is not in the robot's catalogue of objects and never was. The call is perfectly formed and refers to something that does not exist, and the check that most systems rely on has nothing to say about it, because "does this thing exist?" is not a question about shape.

The second layer

toolcall-contract is a validator with two layers. The first is the structural check just described. The second asks the questions a schema cannot:

- **Does the thing referred to exist?** Waypoints, objects, people and zones are checked against the actual catalogues of what is out there.
- **Do the numbers contradict each other?** A task that ends before it begins is well-typed and impossible.
- **Are conditional requirements met?** If you say the setting applies to a zone, you have to name the zone.
- **Are mutually exclusive options being combined?** Some settings cannot both be true.

The first layer is checked against an established, independent validation library across 25,000 randomly generated combinations of rules and data. The two agreed on all 25,000, with zero disagreements. That matters because the whole point of this project is the second layer, and a second layer built on a shaky first one would prove nothing.

What two models actually did

Forty tasks were put to two different language models, both with randomness switched off so the results reproduce exactly. One is small and runs on a laptop; the other is roughly ten times larger and runs in a data centre. Every response is committed to the repository.

Neither model invented a tool that did not exist. Neither model broke a single schema.

> The structural layer alone, the check most systems stop at, would have passed 39 of the 40 calls. For both models. It caught almost nothing, because there was almost nothing of that kind to catch.

The second layer caught seven bad calls from the smaller model and five from the larger one. Every one of them was a call that a schema check waves straight through.

Three ways to be well-formed and wrong

The failures fall into three families, and all of them are quoted here exactly as the model produced them.

Faithfully writing down something that does not exist. Asked to place a battery pack at the workbench, when no workbench exists:

```
{"tool": "place_object", "arguments": {"object_id": "battery_pack",
                  "waypoint": "workbench"}}
```

The model is not being careless. It is being obedient. It was told about a workbench, so it wrote down a workbench. The same pattern produced calls referring to a nonexistent person, a nonexistent zone and a nonexistent object.

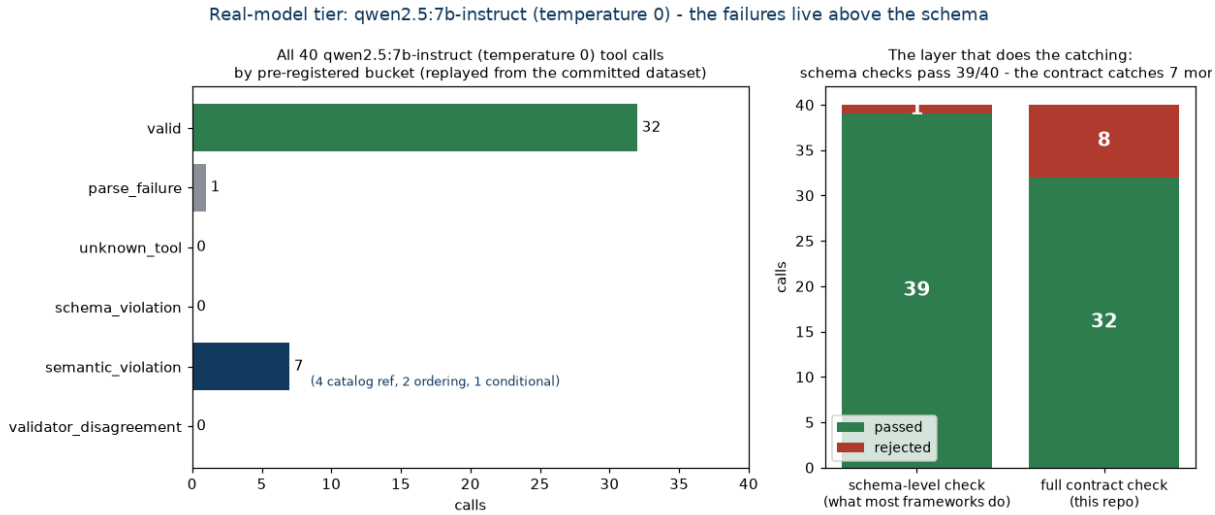

Figure 1: The smaller model's forty calls. On the left, where they land: no invented tools, no broken schemas, seven contract violations. On the right, the comparison that matters: a schema-level check passes 39 of 40, the full contract check rejects 8. The failures live above the schema.

Copying a contradiction without noticing it. Asked to schedule a delivery starting at second 900 and ending at second 300:

```
{"tool": "schedule_task", "arguments": {"task_type": "deliver",
                  "start_time": 900, "end_time": 300}}
```

Both numbers are numbers, so the schema is satisfied. The delivery finishes ten minutes before it starts. The request itself was impossible, and the model transcribed the impossibility rather than flagging it. Another call set a boundary running from 9.0 down to 1.0, the same failure in a different costume.

Leaving a requirement dangling. Asked to set a speed limit for a zone without being told which zone:

```
{"tool": "set_speed_limit", "arguments": {"limit": 1.5, "scope": "zone"}}
```

It says the limit applies to a zone, and never says which. Well-typed, and unactionable.

The most honest thing either model did

One task asked both models to rotate a camera. There is no camera tool.

The smaller model replied with an empty object, `{}`, effectively saying nothing at all. The larger one wrote a short refusal in plain prose naming the missing capability.

Neither invented a camera tool to look helpful. Both failures land in a bucket that exists specifically for this, and it is arguably the best behaviour in either dataset: when the right answer is "I cannot do that with what I have", saying nothing is far better than confidently pressing the wrong button.

The line this validator will not pretend to cross

Here is the part that matters most, and it is an admission rather than a result.

Several of the forty requests were impossible or wrong, and the models quietly turned them into something possible instead. Every one of these calls passes the contract check completely, and is counted as valid:

- Asked to go to *warehouse 3*, which does not exist, the model went to `storage_a` instead. It did not ask, and it did not object.
- Asked to set the speed limit to *5 metres per second*, above what the system permits, it set `0.5`. Ten times slower than requested, silently.
- Asked to *turn the robot off*, which no tool can do, it sent back a battery status report.

Every one of those is a legal, well-formed, contract-satisfying call. The validator passes them, deliberately, and says so in public.

> Whether a call does what the person actually meant is a different question from whether it is valid, and it needs different machinery. Merging the two into a single "valid" score would hide exactly these cases. This project declares that boundary up front rather than quietly scoring across it.

That is an unusual thing for a piece of software to say about itself, and it is the reason the numbers above can be trusted: the tool reports what it can check, and names what it cannot.

How it is kept honest

The categories every call falls into were written down and committed before any results were published, so no flattering category could be invented after seeing the data. Two of those categories are load-bearing: the number of times the two structural checkers disagreed, and the number of times something invalid was marked valid. Both are zero, and the automated tests reject any change that makes them anything else.

Beyond the two models, 262 hand-built cases with known correct answers exercise every rule, and the validator must match every one exactly.

Reproducing all of it needs no AI model and no internet connection. Every response is committed, so the evaluation replays deterministically on any computer, and the tests do exactly that on every change.

What this does not show

Two models, at one setting each, on forty tasks apiece. That is a small sample and the counts are reported per model, never averaged into a claim about "language models". The tool catalogue is a robotics one, so the specific failures are robot-flavoured, though nothing about the three failure families is specific to robots.

The strongest available next step is more models and more tasks, to find out whether "the failures live above the schema" holds as broadly as these two suggest.

Why it matters

AI systems are increasingly wired to things that act: robots, databases, payment systems, calendars. The industry standard for checking those commands answers whether the message is shaped correctly.

On this evidence, shape was never the problem. Both models filled in the form perfectly every single time, and still produced commands that referred to objects that do not exist, scheduled deliveries that end before they begin, and set limits on unnamed zones. Those calls are what reaches the machinery, and a checker that only reads the shape will forward every one of them.

Every task, every raw model response, every figure, and the validator itself are public. This guide is part of a wider line of work on machines that know their own limits: a benchmark measuring *how* language model planners fail, a safety layer that detects unsafe robot routes and recovers or halts, and this validator, which checks the commands before they are sent.
Repository: github.com/munawarkazmi/toolcall-contract