

Too Many Tools Break Mid-Size Models: A Two-Stage Method for Reliable Tool Use

Frits Lyneborg

FRITS AI ApS

ABSTRACT

Model cards imply broad tool support. Operating a Mistral-based assistant with more than twenty tools, we found reliable behaviour only up to roughly four to ten tools per call, with reproducible failures past that: the model answers from memory in prose before deciding to call a tool, so the user watches two contradictory answers appear; it invents tool names borrowed from other vendors' ecosystems; and on smaller models it fails silently — no call, no text, no error — in 86% of calls during burn-in. The worst of these can be eliminated by construction rather than by prompting. Force a tool call on every turn and the model cannot answer before it decides, so the contradiction becomes impossible. We build that into a two-stage architecture: a four-tool router that must end every turn in a tool call, then sixteen specialist categories capped at three tools each. Invented tool names are repaired deterministically instead of fought with prompt rules. A pre-registered go/no-go experiment passed all five criteria, and tool-bound requests came out 24% faster.

1. Mid-size models handle far fewer tools than their model cards imply

The natural architecture for tool use is to hand the model every tool and let it choose [1 (#ref-1)]. On a frontier-scale model this mostly works. On the mid-size open models a European-sovereignty deployment must build on, it does not.

We run a Mistral-based assistant with more than twenty tools. Reliable behaviour stops at roughly four to ten tools per call — far below what the model card implies, and on no public benchmark we could find. Past that ceiling, four failures recur.

Answering before deciding: the two-bubble bug. Under `tool_choice: "auto"`, the model sometimes commits to a prose answer from memory *and then* calls the relevant tool. The user watches one answer appear, then a second that contradicts it. This is the worst class — user-visible, trust-destroying, and beyond the reach of prompting. “Always check the knowledge base first” competes with the model’s fluency at answering directly, and loses often enough to matter.

Tools from other people’s ecosystems. The model reaches for names it saw in training rather than the ones in its schema [3 (#ref-3)]: `codeInterpreter` and `executePython` for our `runPython`, format-suffixed inventions like `createDocx`, lowercase near-misses like `weather`. These are coherent memories of other APIs, which is exactly why they are frequent.

Schema pressure on smaller models. We tried routing on a small model of the same family, for cost. Its time-to-first-token spiked past client timeouts, and burn-in produced silent failures — no call, no text, no error — in 12 of 14 attempts. The cliff between “supports tool calling” and “supports tool calling under production schema load” is wide and unbenchmarked [5 (#ref-5)].

Machinery in the answer. Tool-call JSON, speaker labels and echoes of prior tool metadata leak into visible prose around malformed calls. Occasionally the model enters a repetition loop.

2. Two stages, a four-tool router, a forced call every turn

Two rules follow. Never show the model many tools. Never let it answer without first choosing one.

Stage 1 is a router with exactly four tools [2 (#ref-2)]: `webSearch`, `webContext` (fetch and ground a named source), `routeToSpecialist` [8 (#ref-8)], and `directAnswer`. It runs with `tool_choice: "required"`, so the model cannot answer in prose — it must end the turn

in exactly one call. The two-bubble bug is not discouraged here. It is impossible, because no code path exists in which prose precedes the decision.

`directAnswer` is the “no lookup needed” path, and most turns in a chat product need no tool [7 (#ref-7)]. Its schema is empty: it carries no answer, only the routing signal, and the answer comes from a second, tool-free call that streams token by token. That extra trip exists for a Mistral-specific reason — tool-call arguments arrive as one block rather than streaming, so an answer carried inside the call would mean a long silence followed by a wall of text.

Stage 2 is sixteen specialist categories, at most three tools each. `routeToSpecialist` names one — code, documents, images, charts, weather, knowledge base and so on. Each maps to a fixed tool set with a hard ceiling of three, enforced by a runtime assertion that fails loudly in development. The specialist sees only its own tools and owns the answer. When a category wants a fourth tool, we split the category.

Both stages run on the family’s flagship model. After the small-model burn-in above, we treat routing as flagship work [9 (#ref-9)]: it is one cheap call, and everything downstream depends on it.

3. Repair the model’s mistakes instead of prompting against them

Invented tool names we do not prevent. We repair them.

The rule that beat instruction-stacking: accept what the model produces, and shape the environment so that what it produces works. We call these guide wheels rather than guardrails. Guardrails try to stop the model straying, and grow the prompt — and the latency, and the strangeness of what still gets through — with every rule added.

- **Name repair.** A hook maps unknown tool names and category values onto the intended real ones. The other-vendor imports, the format suffixes and the case variants all resolve instead of erroring. A genuinely unknown name still fails.
- **Fallback chains.** Empty or failed results retry along fixed routes: image generation falls back to image search, the two web tools substitute for each other, knowledge-

base misses fall back to grounded web context. Capped at three attempts. Transient emptiness stops being user-visible.

- **Stream correction.** A transformer strips leaked JSON, speaker labels and metadata echoes as they stream, and cuts repetition loops. It matches only patterns that cannot occur in legitimate output.

One boundary keeps this safe: corrections must be deterministic and certain. Where we know what the model meant, the environment fixes the difference silently. Anything fuzzy stays a prompt problem. Repair is not licence to guess.

4. Validation

Before making forced calling the default we ran a controlled comparison. Five representative prompts — greeting, trivial arithmetic, a knowledge-base question, a current-events question, an image request — each under the incumbent `auto` configuration and the forced two-stage design, two runs each. Small by benchmark standards [6 (#ref-6)], and built as a go/no-go gate on five pre-registered criteria. All five passed.

Criterion	Result
Pre-tool prose hallucination	Eliminated: 0/10 forced-mode sessions (reproduced in auto mode)
Tool selection accuracy	5/5; forced routing chose knowledge-base grounding <i>more</i> often, and one answer improved as a result
Time-to-first-token cost	Median +200 ms (range +188 to +301 ms)
Answer correctness	Equal or better on every prompt
Model breakage under <code>required</code>	None: 0 malformed calls, 0 schema errors

One result we did not predict: image requests finished 24% faster, 10.4 s against 13.6 s. Under forced routing, generation halts the moment the routing call is emitted; in auto

mode the model habitually produced post-decision filler tokens that taxed every tool-bound request. Forcing the call removed a waste we had not noticed we were paying.

That battery is now permanent infrastructure — 100+ prompts across all stage-1 tools and specialist categories in nine languages, re-run whenever routing text changes, and reused as the routing-fidelity phase of the exam any new model must pass (see FRITS-TR-2026-01).

5. Costs and limitations

The two-trip `directAnswer` design roughly doubles LLM calls on no-tool turns and adds its round trip to their latency — the standing cost of both streaming and the structural guarantee. The +200 ms first-token overhead applies to every turn, and the sixteen-category taxonomy is maintained by hand.

Every number here measures one model family on one day. The ten-tool ceiling, the 86% small-model failure rate and the latency deltas will move with model generations, so re-measure rather than inherit them. The validation was a small pre-registered gate, not a benchmark; it earned the rollout, and production logs have done the real validating since.

What transfers is the shape rather than the constants. Cap what any single call can see. Where a failure class is intolerable, make it structurally impossible rather than instructionally discouraged. Route hierarchically, so tool breadth lives in a taxonomy instead of one schema [4 (#ref-4)]. And fix the model's vocabulary in your own code rather than in its prompt.

Corrections and prior art pointers are welcome: [contact \(/contact/\)](/contact/).

References

1. Schick, T., et al. — *Toolformer: Language Models Can Teach Themselves to Use Tools*. Meta AI, 2023. arXiv:2302.04761.

2. Yao, S., et al. — *ReAct: Synergizing Reasoning and Acting in Language Models*. Princeton / Google, ICLR 2023. arXiv:2210.03629.
3. Patil, S. G., Zhang, T., Wang, X., Gonzalez, J. E. — *Gorilla: Large Language Model Connected with Massive APIs*. UC Berkeley, 2023. arXiv:2305.15334.
4. Qin, Y., et al. — *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs*. Tsinghua University (THUNLP) et al., ICLR 2024. arXiv:2307.16789.
5. Li, M., et al. — *API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs*. Alibaba DAMO Academy, EMNLP 2023. arXiv:2304.08244.
6. Patil, S. G., Mao, H., Yan, F., Ji, C. C.-J., Suresh, V., Stoica, I., Gonzalez, J. E. — *The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models*. ICML 2025, PMLR v267, pp. 48371–48392.
7. Chen, L., Zaharia, M., Zou, J. — *FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance*. Stanford University, 2023. arXiv:2305.05176.
8. Ong, I., et al. — *RouteLLM: Learning to Route LLMs with Preference Data*. UC Berkeley / LMSYS, ICLR 2025. arXiv:2406.18665.
9. Ding, D., et al. — *Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing*. Microsoft Research, ICLR 2024. arXiv:2404.14618.

How to cite

Frits Lyneborg (2026). Too Many Tools Break Mid-Size Models: A Two-Stage Method for Reliable Tool Use. FRITS AI ApS, Technical Report FRITS-TR-2026-03. <https://frits.ai/research/two-stage-tool-routing/> doi:10.5281/zenodo.21274521.

```
@techreport{lyneborg2026many,
  title      = {Too Many Tools Break Mid-Size Models: A Two-Stage Method for Reliable Tool Use},
  author     = {Lyneborg, Frits},
  institution = {FRITS AI ApS},
  number     = {FRITS-TR-2026-03},
  year       = {2026},
  month      = {jul},
  doi        = {10.5281/zenodo.21274521},
  url        = {https://frits.ai/research/two-stage-tool-routing/}
}
```