

Not Every Call Needs a Frontier Model: Per-Call-Site Evaluation of Small Language Models in a Deployed Agentic Home-Automation System

Panagiotis Kasnesis^{1,2,3} Christos Chatzigeorgiou^{1,2}
Lazaros Toumanidis^{1,2} Amalia Contiero Syropoulou^{1,3}

¹University of West Attica, Athens, Greece

²Waldiez PC ³ThinGenious PC

{pkasnesis, chrihatz, laztoun, acontiero}@uniwa.gr

Abstract

An agentic system issues several structurally different kinds of LLM calls. It routes intent, classifies actions, grounds language in a device registry, plans multi-agent pipelines and writes the Python code those pipelines run. The difficulty of these call sites varies by an order of magnitude, yet in practice a single model, chosen for the hardest site, serves all of them. In this work, we evaluate 9 models from 0.8B to a frontier hosted model across the five call sites of a deployed open-source home-automation framework (Wactorz), using its unmodified production prompts and two real Home Assistant installations (280 cases, 2520 scored calls). We find that capability is not ordered the same way at every site, and that larger models are not uniformly better: one 4B model is worse than its 2B sibling at grounded actuation. Paired testing shows the best local model to be statistically indistinguishable from both hosted models at four of five sites. Only code generation separates them, against a small hosted model ($p = 0.039$) as well as a frontier one ($p = 0.002$). Aggregate accuracy also *hides a safety failure specific to actuation*, where small models resolve the accuracy/refusal trade-off in degenerate ways: one model (Gemma4 E2B) actuates on 87.2% of requests for devices the site does not own, while another refuses every request it receives. Routing each site to its best local model reaches 91.8% against 95.4% at no per-call cost. In a live deployment judged by a user, hosting only the two generative sites matches hosting everything (39/43 against 39/43) for 28% of the spend, and the actuation gap the benchmark predicted appears as exactly one case in twenty-six. Benchmark, harness and all records are released at <https://github.com/waldiez/slm-callsite-eval>.

1 Introduction

The case for small language models in agentic systems is usually made at the level of a whole model: can a 4B model serve as the agent? This framing obscures where the practical leverage lies. A running framework issues several structurally different LLM calls per turn, and their difficulty varies widely. Deciding whether “turn off the tv” is an actuation or an automation request is a four-way classification with a ten-token budget, while writing an asynchronous agent that subscribes to an Pub/Sub (e.g., MQTT) topic with a correctly-shaped callback is program synthesis against an unfamiliar API. Systems typically serve both from one model sized for the second task, and as a result pay hosted prices for the first, which is also the call made most often.

We treat model selection as a *per-call-site* decision and measure it in a system (i.e., Wactorz [23], an open-source multi-agent home-automation runtime in which long-lived agents are supervised over MQTT) that is deployed rather than simulated. The first three of its five call sites are evaluated against the framework’s production system prompts, imported from the agent code, so the numbers describe the deployed system rather than a proxy; all five prompts are reproduced verbatim in Appendix A.

Contributions. We contribute (i) a 280-case benchmark over five call sites of a real agentic system, grounded in two installations with different inventories, including cross-site cases answerable at one site and impossible at the other; (ii) a 9-model measurement with paired testing, showing per-site orderings that differ, are non-monotonic in size, and leave only one call site separating local from hosted; (iii) an actuation-safety metric that separates models usable to drive hardware from models with the same aggregate score that are not; and (iv) routing policies tested end to end on the live deployment, where the offline prediction held and three failures appeared that no per-call benchmark can see. Code, benchmark and all 2520 records are released.¹

2 Related work

Small models in agentic systems. Belcak et al. [1] argue that most agentic calls are repetitive and narrow, that small models therefore suffice, and sketch a conversion procedure: instrument a running system, find the calls a smaller model can serve, and migrate them. This paper carries that procedure out on a system that is already deployed, and reports what the position paper could only project. Capability becomes a per-site table whose ordering differs by site and is not monotone in size; reliability becomes the finding that at actuation the failure is one of calibration rather than accuracy, in both directions; and the economic illustration becomes a measured 28% of hosted spend at no loss in task success. Surveys and on-device work provide the capability picture the position assumes [11, 12].

Routing per query, or per call site. Cost-aware serving is otherwise studied *per query*, through cascades that escalate until an answer is good enough [2] and learned routers that pick a strong or weak model [4, 7, 15, 20]. All of these need a judge or router at run time, and all of them can mis-route. Routing in this paper is instead *per call site*: the framework knows statically which call it is about to make, so the assignment is a configuration value that needs no extra model and cannot send a call to the wrong site, at the cost of not adapting to a hard instance of an easy site. The two are complementary rather than competing: a per-query router could still choose among models *within* a site. Agent evaluation asks how good a model is *at being* an agent [10, 17, 25], building on the agent-construction and API-grounding lines [16, 18, 21, 24]; we ask instead which calls of an agent system that already exists can move to a smaller model without the system getting worse.

Substitution, not hallucination. Closest to our actuator site, Li et al. [9] score valid and invalid smart-home instructions. Our invalid requests name devices that *the other installation really owns*, so a wrong answer is substitution rather than hallucination, and it actuates real hardware. King et al. [8] map under-specified goals onto device plans. Neither work decomposes a deployed system by call site, and neither reports cost.

3 Call sites, benchmark and setup

Call sites. Figure 1 shows the five call sites. The call sites are those of Wactorz’s deployed architecture, which we evaluate as it stands rather than propose as a design. *Intent* is a four-way label under a ten-token budget. *HA action* classifies a Home Assistant request into one of the framework’s action labels. *Actuator* maps language to a JSON array of service calls against the site’s live entity registry. *Planner* emits a pipeline specification that must carry concrete entity ids, topics and schedules rather than describe them, and *codegen* writes the Python body of each agent that specification names, against the framework’s agent API. Only the last two are generative; the first three emit a label or a short structured object, which is the asymmetry the rest of this paper turns on.

¹<https://github.com/waldiez/slm-callsite-eval>

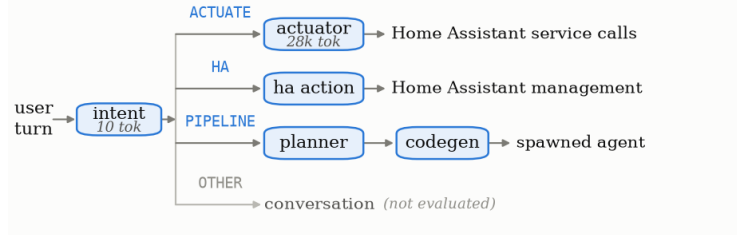


Figure 1: The five call sites of one user turn. *Intent* is a static gate: it runs on every turn under a ten-token budget and fixes which downstream site executes, so a per-call-site model assignment is a configuration value and cannot mis-route between sites. *Actuator* is a leaf but the expensive one; its prompt carries the site’s whole entity registry, 28 122 input tokens on average, which is where 81% of hosted spend goes. *Planner* and *codegen* form the only chain, which is what makes “hosted for generation only” a clean cut. The *OTHER* branch is ordinary conversation and is not evaluated here.

Table 1: Accuracy (%) per call site, one pass over the frozen benchmark at temperature 0. Local models are served over an OpenAI-compatible endpoint and cost nothing per call; the hosted rows are measured API spend for the whole 280-case sweep. Paired significance against the best local model is in Table 2.

Model	Intent <i>n</i> =36	HA action <i>n</i> =28	Actuator <i>n</i> =116	Planner <i>n</i> =50	Codegen <i>n</i> =50	All <i>n</i> =280	Cost (\$)
Qwen3.5 0.8B	50.0	71.4	33.6	8.0	24.0	33.2	0
Qwen3.5 2B	61.1	78.6	62.1	30.0	36.0	53.2	0
Qwen3.5 4B	91.7	92.9	50.9	52.0	38.0	58.2	0
Gemma4 E2B	83.3	89.3	56.0	46.0	34.0	57.1	0
Gemma4 E4B	91.7	96.4	81.9	50.0	80.0	78.6	0
Gemma4 12B	86.1	89.3	95.7	76.0	84.0	88.2	0
Gemma4 26B	91.7	96.4	94.8	88.0	80.0	90.7	0
Claude Haiku 4.5	80.6	92.9	93.1	82.0	94.0	89.6	3.06
Claude Sonnet 5	94.4	92.9	99.1	84.0	100.0	95.4	12.13

Scoring. Intent and HA action are exact label matches. Actuator requires an *exact action-set* match: an unrequested extra actuation counts as a failure, because in a house an unwanted action is worse than none. Cases whose expected output is empty require refusal, since the device does not exist at that site. Planner requires a valid plan of allowed agent types that resolves the entity ids and topics it was given rather than describing them. Codegen requires code that compiles and defines the required async functions; §5 tightens this to code that *runs*.

Benchmark. 280 cases: 36 intent, 28 HA action, 116 actuator, 50 planner, 50 codegen. The actuator set spans two live installations with different inventories, a household and an office, and includes *flip* cases (a device only the other site owns), *nearmiss* cases (a plausible name for an absent device) and refusals; sub-groups are encoded in case ids and broken out in Appendix C.

Models and hardware. 9 models: Qwen3.5 at 0.8B/2B/4B [22], Gemma4 at E2B/E4B/12B/26B [5], and Claude Sonnet 5 and Haiku 4.5 as hosted references. Local models run on one NVIDIA DGX Spark (GB10 Blackwell, 128 GB unified memory), served by llama.cpp [6] behind a LiteLLM proxy, with every model at 4-bit GGUF and a 128k context, so no prompt in the suite is truncated. Two properties of these builds bear on how Table 1 should be read. First, Gemma4 26B is a mixture of experts with roughly 4B parameters active per token, so the column order is not an order of inference cost. Second, the Gemma4 builds are quantization-aware trained while the Qwen3.5 builds are quantized after training, a confound we cannot separate from the architectures. Exact builds are in Appendix E. All calls run at temperature 0, one pass per case (see §6).

Table 2: Paired McNemar tests on the same cases, exact two-sided binomial. Each block compares the best all-round local model (Gemma4 26B) with one hosted model; *disc.* gives the discordant split as (local only / hosted only). Four of the five call sites are statistically indistinguishable from either hosted model. Codegen is the exception, and it separates Gemma4 26B from *both* — including from the small hosted model, which is why we read the gap as a property of the call site rather than of frontier scale.

Call site	Gemma4 26B	vs. Haiku 4.5		vs. Sonnet 5	
		disc.	p	disc.	p
Intent	33/36	4/0	0.125	1/2	1.000
HA action	27/28	1/0	1.000	2/1	1.000
Actuator	110/116	6/4	0.754	0/5	0.062
Planner	44/50	5/2	0.453	7/5	0.774
Codegen	40/50	1/8	0.039	0/10	0.002

4 Results

Per-site orderings differ, and are not monotonic in size. No model in Table 1 wins every column: three different local models take the five sites. Size does not settle the ordering either. Qwen3.5 4B scores below Qwen3.5 2B on actuation (50.9% vs. 62.1%), and Gemma4 26B falls below Gemma4 12B on code generation (80.0% vs. 84.0%). Nor is the hosted model uniformly best: Gemma4 26B beats it at planning (84% vs. 88.0%), a gap driven by the *infeasible* sub-group, where the hosted model plans a pipeline for a request the site cannot support 20% of the time (Appendix Table 7). Scale evidently has not removed this kind of over-eagerness.

Only code generation separates local from hosted. Because column-wise gaps invite over-reading, Table 2 pairs the best all-round local model against each hosted model case by case (McNemar exact, 13). On four of five sites it is statistically indistinguishable from both. Code generation separates it from *both* hosted models ($p = 0.002$ against Sonnet 5, $p = 0.039$ against Haiku 4.5), with every discordant case favouring the hosted model. The fact that a small, cheap hosted model reproduces the gap is our strongest evidence that the codegen deficit is a property of the call site itself rather than of frontier scale. The actuator column remains open: the comparison misses significance against Sonnet 5 ($p = 0.062$), but all five discordant cases fall in the same direction, which points to insufficient statistical power rather than a true null result. More sampling passes cannot resolve this, since the test is over a fixed set of 116 cases.

The codegen gap is concentrated in a handful of API shapes. Appendix Table 8 splits code generation by the framework API a case requires, and the deficit turns out to be narrow rather than broad. Gemma4 26B already matches the hosted models on five of twelve shapes (blocking work, topic contracts, one-shot agents, publication and state), and loses almost all of its ground on the callback and lifecycle shapes: `window` 50, `poll` 67, `cleanup` 67 and `subscribe` 71, against 100 for both hosted models. These are the same primitives that failed on real hardware in §5, where generated agents called `agent.window()` and silently read zero forever. The offline benchmark therefore localised the weakness before the runtime confirmed it, which supports splitting a call site by the API it exercises instead of scoring it as a single number. One shape resists both small models: `11m` defeats Gemma4 26B and Haiku 4.5 equally (33 each), where Sonnet 5 scores 100.

What separates codegen from planning. Both sites are generative, but they ask for different things. The planner emits a declarative specification over a fixed set of allowed agent types and is scored on whether it resolves the entity ids and topics it was given, all of which are in its prompt. Codegen must follow a framework contract that runs against common asynchronous idioms: `subscribe` and `window` are synchronous calls inside `async` code, callbacks take exactly one argument, and `process` is invoked on a `poll` rather than looping. The prompt documents these rules with explicit correct and incorrect examples (Appendix A), yet the shapes on which local models lose ground are exactly these, while shapes that reduce to ordinary Python (blocking work, publication, state) are already matched. We therefore read the gap as difficulty following an unfamiliar, framework-specific API rather than a general limit on program synthesis at this scale.

Table 3: Whole-system accuracy and cost under four routing policies over the same 280 cases. Accuracy is weighted by benchmark case counts, not by a deployment’s call mix. Generation-only policies keep intent, HA action and actuation on the best local model per site and send planner and codegen to the hosted model named.

Routing configuration	Acc. (%)	\$ total	\$/call	Saving (%)
All hosted (Sonnet 5)	95.4	12.13	0.0433	0
All hosted (Haiku 4.5)	89.6	3.06	0.0109	75
Hosted for generation only (Sonnet 5)	93.9	2.15	0.0077	82
Hosted for generation only (Haiku 4.5)	92.5	0.56	0.0020	95
All local, best per site	91.8	0.00	0.0000	100
All local, single model (Gemma4 26B)	90.7	0.00	0.0000	100

Small hosted is not uniformly better than local. Haiku 4.5 scores 80.6% at intent, below five of the seven local models, and 89.6% over all five sites, which is *below* Gemma4 26B’s 90.7%. The two models reach near-identical totals with opposite profiles: the local model is stronger at routing, and the hosted one is stronger at code generation.

Aggregate accuracy hides the actuation trade-off. Appendix Figure 2 splits the actuator site into requests that should produce service calls and requests that should produce none. Small models fail in two opposite ways. Both are invisible in a single accuracy number, and both correspond to the calibration failure that over-refusal benchmarks study in the safety setting [3, 19], here with a physical consequence. Qwen3.5 0.8B does not refuse so much as emit a constant: across all 116 actuator cases it returns the same two-token empty array, so its perfect 0.0% over-actuation rate and its 0.0% execution accuracy are the same fact, and the 39 refusals it scores correct carry no evidence that it recognised anything. It generates normally at every other call site (i.e., 50 distinct outputs at codegen), so the collapse belongs to this one site; whether that is the 20k-token registry in the prompt or the demand to resolve an exact entity id we cannot separate, since the two vary together in our design. Scale within one family does not settle the direction either: Qwen3.5 2B over-actuates on 66.7% of impossible requests, while Qwen3.5 4B returns an empty action list on 72.7% of actionable ones (Appendix Figure 2). Calibration at this site therefore has to be measured per model rather than inferred from size.

Constrained decoding would not close the gap. A natural mitigation is to restrict the actuator’s output to the site’s valid entity ids. That guarantees a well-formed id but not a correct decision to act. Of the 70 service calls that models emitted on impossible requests, 53 (76%) already named an entity that exists in the site’s registry: the model chose a real but wrong device, so a grammar over valid ids would not have blocked them, and for the remainder it would force a choice among real devices unless the empty action list stays available. Because our impossible requests name devices the *other* installation owns, the failure is substitution, and preventing it needs calibration of *whether* to act, not only constraints on *what* may be emitted. Such guardrails remain worth deploying, but alongside a calibrated model rather than in place of one.

Routing beats model choice. All-local per-site routing costs nothing and gives up 3.6 points against an all-hosted system; keeping only the generative sites hosted gives up 1.4 points and still cuts spend by 82%, with the three non-generative sites served by the best local model for each (i.e., Qwen3.5 4B for intent, Gemma4 E4B for HA action and Gemma4 12B for actuation; a tie at intent goes to the smallest model). Appendix Table 11 shows why: 81% of hosted spend goes to the actuator site, whose prompt carries the entire entity registry, and this is precisely the site that a mid-sized local model already serves at 95.7%. In other words, the call that dominates spend is not the call that needs the most capable model. Which hosted model serves the generative sites is a further lever: routing them to Haiku 4.5 instead of Sonnet 5 costs 92.5% against 93.9%, a difference of 1.4 points, for \$0.56 against \$2.15, a quarter of the spend. These accuracies are weighted by benchmark case counts rather than by a deployment’s call mix; a traffic-weighted view would favour local routing further.

Table 4: End-to-end outcomes on the live deployment, judged by a person: the requested effect happened, nothing else was actuated, the reply claimed nothing untrue. Only the two generative call sites differ between the middle and right columns. $n=46$ cases scored in every configuration, on one installation. 3 cases hit limitations outside the model — a framework bug in the sliding-window API, and an LLM client a spawned agent could not reach — and the same 3 are excluded from the second total in every column, so that row does not simply subtract failures: one of the three passed under all-hosted.

	all-hosted	gen-hosted	all-local
Direct (actuate, refuse)	25/26	24/26	24/26
Pipeline (scheduled, reactive, spawn)	15/20	15/20	12/20
All cases	40/46	39/46	36/46
Excluding system-limited cases	39/43	39/43	35/43
Hosted spend (\$)	2.47	0.70	0.00

5 End-to-end validation on the live deployment

Per-site accuracy indicates which calls a small model can serve, but not whether a system wired that way works as a whole. We therefore drove the running deployment through its chat endpoint under three routing configurations on one installation, judging every case as a user would: the requested effect happened, nothing else was actuated, and the reply claimed nothing untrue. Judgement is by a person rather than a model [26] or a side-effect check, because neither of those would catch an agent that registers, runs and does the wrong thing, a failure that is common here. 46 cases were scored in all 3 configurations.

Hosting only the generative call sites matches hosting everything. Sending planner and codegen to the frontier model while keeping everything else local scores 15/20 on pipeline cases, the same as hosting every call site, for 28% of the spend (\$0.70 against \$2.47). Excluding the 3 system-limited cases, the two configurations are indistinguishable at 39/43. Running everything locally costs three pipeline cases (12/20).

The offline benchmark predicted the deployed outcome. Only the generative sites differ between the middle and bottom configurations, so the direct cases act as a control, and they behave like one: 24/26 and 24/26, identical across all 26. Moving actuation from the frontier model to a local one costs exactly one case in twenty-six (25/26 against 24/26), or four points, and Table 1 predicted four (99.1% against 94.8%). An offline measurement on a frozen benchmark anticipated what the deployed system did to real hardware.

Scale is not monotone here either. The all-hosted configuration was the only one to fail *in two minutes turn off the livingroom lights*, classifying it as conversation and actuating nothing, while both configurations with a local intent classifier handled it. It also answered *when the livingroom door opens turn on the lights there* by switching on every light in the house, where the local configurations replied that the site has no door sensor. This is the over-actuation of §4 appearing live, and it came from the model that never over-actuated offline. Both are cases the cheaper system got right.

What the harness could not have told us. Three findings came only from running the system. Two pipeline cases failed because `agent.window()` returns a fresh empty buffer on every call, so an agent that counts or averages reads zero forever; here, correct generated code was defeated by a framework API. A third case degraded silently to a statistical summary because the spawned agent could not reach an LLM client. Most consequentially, several requests never reached the codegen site at all: the framework matches them to a pre-built catalogue recipe and spawns it directly, at no cost and independent of the routing configuration. This bounds §4: the codegen site matters less in deployment than a benchmark that weights categories equally would imply.

6 Limitations

One pass per case at temperature 0: the generative sites vary between samples and we report no variance, though one end-to-end case passed in two configurations and failed in the third, so that variance is real. Two installations, a household and an office, with ground truth annotated by the authors; diversity across further homes and buildings (layouts, device mixes, registry sizes) is therefore untested, and synthetic suites such as HomeBench [9] offer a breadth that real installations cannot, complementing our grounding in live hardware. The end-to-end run covers one of them and its verdicts come from a single unblinded rater. Its criteria are observable (the effect happened, nothing else moved, the reply claimed nothing untrue) and the annotations are released for re-judging, while a blinded, multi-rater protocol is future work. Three intent keys and the planner entity block were revised after inspecting outputs. Two of the affected cases routed to a Home Assistant path the framework has disabled, one asked for a hardware recommendation that the route’s own definition excludes, and the planner prompt omitted the area labels the deployed system sends, leaving location cases unanswerable. The revisions moved the hosted model $+0.08$ and three local models -0.03 , against the direction of our conclusion, and both key sets are released. Mislabelled answers are common in benchmarks [14], and we report our revisions openly so they can be checked. Subgroup counts are small (2–10), so those percentages are indicative rather than conclusive. Offline accuracy counts every benchmark case equally, while a real deployment makes some calls far more often than others; §5 shows that this gives the codegen site more weight than it has in practice. Offline accuracy is weighted by benchmark composition rather than by a real call mix, which §5 shows overstates the codegen site. Latency is single-request, with no batching and no energy measurement; only wall-clock time per call was recorded, so time to first token cannot be separated from decoding, and hosted latency includes the network round trip. We did not vary the size of the entity registry, which grows with the home and dominates the actuator prompt, so how small models degrade as it grows is open. Compound requests that mix an action with conversation follow the framework’s single-label intent contract and are labelled OTHER, so their actionable part is not evaluated; decomposing them is future work. Static assignments must be revisited when prompts or models change, and the released harness turns that into a scripted re-run rather than a manual judgement; we did not compare them against a learned per-query router. All local models ran on a DGX Spark, a high-end edge device with 128 GB of unified memory. The 4-bit builds of the mid-sized models have modest memory footprints, but we have not measured them on less powerful edge hardware, where latency and throughput may differ. Finally, the prompts are one framework’s: the call-site *taxonomy* generalises to agentic systems with a routing/grounding/planning/codegen split, but the absolute numbers do not.

7 Conclusion

Model selection in an agentic system is best treated as a per-call-site routing decision rather than a single procurement choice. Once the expensive calls are separated from the difficult ones, most of a deployed framework’s traffic can run locally at no per-call cost. For safety, what matters is not aggregate accuracy but the joint rate of acting when asked and refusing when the device does not exist; aggregate scores hide this pair, and it decides whether a model can be trusted with a lock. The specific assignments belong to one framework and two installations; what we expect to carry over is the procedure of measuring each call site separately and checking the result on the live system. Harness, benchmark, the 2520 offline records and the end-to-end annotations are available at <https://github.com/waldiez/slm-callsite-eval>, and every table and percentage here is generated from them. All experiments use the production prompts of Wactorz [23] v0.5.3, and the live run used a v0.5.3-based deployment.

Acknowledgments and Disclosure of Funding

Research supported by the NVIDIA Academic Grant Program using two NVIDIA DGX Spark systems.

References

- [1] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic AI. *arXiv preprint arXiv:2506.02153*, 2025.
- [2] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.
- [3] Justin Cui, Wei-Lin Chiang, Ion Stoica, and Cho-Jui Hsieh. OR-Bench: An over-refusal benchmark for large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 11515–11542, 2025.
- [4] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024.
- [5] Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.
- [6] Georgi Gerganov et al. llama.cpp: LLM inference in C/C++. <https://github.com/ggml-org/llama.cpp>, 2023.
- [7] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. RouterBench: A benchmark for multi-LLM routing system. *arXiv preprint arXiv:2403.12031*, 2024.
- [8] Evan King, Haoxiang Yu, Sangsu Lee, and Christine Julien. Sasha: Creative goal-oriented reasoning in smart homes with large language models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 8(1):1–28, 2024. doi: 10.1145/3643505.
- [9] Silin Li, Yuhang Guo, Jiashu Yao, Zeming Liu, and Haifeng Wang. HomeBench: Evaluating LLMs in smart homes with valid and invalid instructions across single and multiple devices. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vienna, Austria, 2025.
- [10] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shuntian Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Wei Chen, Yuxuan Shen, Xiaohan Zhang, Yu Su, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024.
- [11] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In *International Conference on Machine Learning (ICML)*, 2024.
- [12] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D. Lane, and Mengwei Xu. Small language models: Survey, measurements, and insights. *arXiv preprint arXiv:2409.15790*, 2024.
- [13] Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.
- [14] Curtis G. Northcutt, Anish Athalye, and Jonas Mueller. Pervasive label errors in test sets destabilize machine learning benchmarks. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2021.
- [15] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024.
- [16] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

- [17] Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning (ICML)*, 2025.
- [18] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024.
- [19] Paul Röttger, Hannah Rose Kirk, Bertie Vidgen, Giuseppe Attanasio, Federico Bianchi, and Dirk Hovy. XSTest: A test suite for identifying exaggerated safety behaviours in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024.
- [20] Marija Šakota, Maxime Peyrard, and Robert West. Fly-swat or cannon? cost-effective language model choice via meta-modeling. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining (WSDM)*, 2024.
- [21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [22] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL <https://qwen.ai/blog?id=qwen3.5>.
- [23] Wactorz Contributors. Wactorz: Actor-model multi-agent framework. <https://github.com/waldiez/wactorz>, 2026. Version 0.5.3.
- [24] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- [25] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ -bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.
- [26] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

A Prompts

Every prompt below is generated from the constant the running system uses, not transcribed. The first three are the deployed prompts unchanged. The last two are condensed for the harness: the deployed planner and codegen prompts are assembled at call time from the live entity registry and the running agent inventory and reach several hundred lines, so the variants here keep the output contract that scoring depends on and drop the situational material. Where a prompt is completed at call time with site-specific content, that content is named rather than reproduced.

A.1 Intent routing

agents/prompts/main_actor_prompts.py

INTENT_CLASSIFIER_PROMPT, 28 lines. Used unmodified by the deployed system.

Appended at call time: The user's turn, and a 10-token output budget.

```
You are a routing classifier for a smart home AI assistant.
Respond with exactly one token: ACTUATE, HA, PIPELINE, or OTHER.

ACTUATE = immediate one-shot device control in Home Assistant:
- Turn on/off a device right now
- Set temperature, dim lights, lock/unlock door
- Open/close covers or blinds right now
- Any direct command whose whole purpose is immediate device control

HA = Home Assistant management, listing, or automation CRUD:
- List devices, areas, entities, automations
- Create/edit/delete a HA automation
- Query what devices or automations exist

PIPELINE = a reactive rule that should run continuously:
- 'if X happens then do Y' -- any conditional/reactive logic
- 'when X send me a message/notification'
- 'whenever X turns on/off do Y'
- Any rule involving a sensor state change triggering an action or notification
- Any webcam/camera detection triggering anything
- Anything involving Discord/Telegram notifications triggered by an event

OTHER = general conversation, coding, questions, or mixed requests.anything not HA or pipeli \
ne related.

Important:
- Choose ACTUATE only when the entire request is immediate device control.
- If the request mixes device control with non-HA tasks, return OTHER.
- If the request is about automations, listing, discovery, or CRUD, return HA.
```

A.2 Home Assistant action classification

agents/prompts/home_assistant_prompts.py

HA_ACTION_CLASSIFICATION_PROMPT, 35 lines. Used unmodified by the deployed system.

Appended at call time: The user's turn.

```
Classify a Home Assistant user request.
Output exactly one of these strings -- nothing else, no punctuation:

recommend_hardware
create_automation
delete_automation
edit_automation
list_automations
list_areas
list_devices
list_entities
other
```

unknown

Guidelines:

- recommend_hardware → user only wants hardware/device suggestions, compatibility info, or \ to know what the existing hardware can do
- create_automation → user wants to create/add/build/make a new automation, even if they a \ also mention choosing between existing sensors, lights, or devices
- delete_automation → user wants to delete/remove/disable an existing automation
- edit_automation → user wants to update/change/rename/modify an existing automation
- list_automations → user explicitly asks to list/show/enumerate existing automations
- list_areas → user explicitly asks to list/show/enumerate areas
- list_devices → user explicitly asks to list/show/enumerate devices as an inventory
- list_entities → user explicitly asks to list/show/enumerate entities or entity IDs a \ s an inventory
- other → Home Assistant related, but not one of the supported operations above
- unknown → request is unclear or not Home Assistant related

Decision rule:

- If the user asks you to create/build/add/set up an automation, classify as create_automati \ on.
- Use recommend_hardware only when the user is asking about hardware feasibility or hardware \ choices and is not asking you to actually create the automation.
- Use other for Home Assistant status/context questions that need current HA data.
- Use other for existence, count, lookup, or state questions about specific HA devices, sens \ ors, entities, rooms, or device types.
- "Do I have any thermometers?" is other, not list_devices.
- "What is the state of my thermometer?" is other, not list_entities.
- Use other for camera/snapshot/stream questions, e.g. "show me the kitchen camera", "take a \ snapshot of the front door", "give me the stream for camera.backyard", "do you have any c \ ameras?", "can you show a camera snapshot?".
- Use unknown for non-Home-Assistant requests.

A.3 Grounded actuation

agents/one_off_actuator_agent.py

RESOLVER_PROMPT, 71 lines. Used unmodified by the deployed system.

Appended at call time: The user's turn followed by the site's entire entity registry, one entity id and friendly name per line. That list is what makes this the longest prompt in the system and the one that dominates hosted spend; it is site-specific and is not reproduced here.

You are a Home Assistant service-call resolver.

Your task:

- Convert the user's natural-language device control request into one or more Home Assistant \ service calls.
- Use only entities that exist in the provided Home Assistant device discovery payload.
- Return strict JSON only: an array of action objects.

Action schema:

```
[
  {
    "domain": "light",
    "service": "turn_on",
    "entity_id": "light.living_room_lamp",
    "service_data": {"brightness_pct": 50}
  }
]
```

Worked examples:

Request: "dim the living room lamp to 50%" with light.living_room_lamp in the payload:

```
[
  {
    "domain": "light",
    "service": "turn_on",
    "entity_id": "light.living_room_lamp",
```

```

    "service_data": {"brightness_pct": 50}
  }
]

Request: "turn off the TV" with media_player.living_room_tv in the payload:
[
  {
    "domain": "media_player",
    "service": "turn_off",
    "entity_id": "media_player.living_room_tv"
  }
]

Request: "turn off the TV" when NO tv/media_player entity exists in the payload
(only lights, switches, etc.):
[]

Rules:
- Return an array, never an object.
- Use the most specific matching entity_id available.
- The chosen entity MUST be the device the user named. NEVER substitute a
  different device: if the user says "TV" and no TV-like entity exists in the
  payload, return [] -- do NOT act on a light, switch, or any other device
  instead. Acting on the wrong device is far worse than doing nothing.
- Return ONLY actions for the devices the user asked about in THIS request.
  Do NOT add extra actions for other devices the user did not mention.
  The worked examples above illustrate the output FORMAT only -- never copy
  their actions into your answer.
- If the request is ambiguous or no device matches, return [].
- For multiple commands in one request, return multiple actions.
- Only include service_data keys that are needed.
- Common examples:
  - turn on/off light or switch -> light.turn_on / light.turn_off or switch.turn_on / switch \
    .turn_off
  - set a light color -> light.turn_on with service_data containing rgb_color
  - set heating/thermostat temperature -> climate.set_temperature with {"temperature": number}
  - lock/unlock door -> lock.lock / lock.unlock
  - open/close cover/blinds -> cover.open_cover / cover.close_cover
  - brightness percent -> use {"brightness_pct": number}
- Color requests:
  - For "blue", use {"rgb_color": [0, 0, 255]}.
  - For "pink", use {"rgb_color": [255, 105, 180]}.
  - For "red", use {"rgb_color": [255, 0, 0]}; "green" -> [0, 255, 0]; "purple" -> [128, 0, \
    128]; "white" -> [255, 255, 255].
  - If the user says just "my light" / "the light" and a color is requested, prefer a color- \
    capable light entity over a color-temperature-only light. Color-capable entities usually \
    have state attributes such as supported_color_modes containing hs, rgb, rgbw, rgbww, or \
    xy, or current_color_mode hs/rgb/xy.
  - Do not send color service_data to a light that only supports color_temp.
- Do not invent entity IDs.
- Do not return markdown or explanation.

```

A.4 Pipeline planning

evalharness.py

PLANNER_SYSTEM, 30 lines. Condensed for the harness; see below.

Appended at call time: The user's turn, the entity registry, and the topics and names of agents already running. The deployed prompt is assembled at call time and runs to several hundred lines; this variant preserves the output contract that scoring depends on.

```

You are designing reactive automation pipelines for a multi-agent IoT system.
Output ONLY a valid JSON array - no explanation, no markdown, no code fences.
Each array item is a COMPLETE spawn config: it must contain the concrete
entity ids, MQTT topics and schedules needed to run, not just a description.

```

```

TYPE 1 - "ha_actuator": call a Home Assistant service when an MQTT message arrives

```

```

{"name": "<kebab-case>", "type": "ha_actuator", "description": "<what it does>",
 "mqtt_topics": [<trigger-topic>],
 "actions": [{<domain>: "<ha-domain>", "service": "<ha-service>",
   "entity_id": "<entity_id from the entity list>", "service_data": {}}],
 "detection_filter": {<payload-key>: <value>} or null,
 "cooldown_seconds": <number>}

TYPE 2 - "scheduled": fire at a time or interval
{"name": "<kebab-case>", "type": "scheduled", "description": "<what it fires>",
 "schedule": {<type>: "daily", "at": "17:00"} | {<type>: "weekly", "at": "07:30",
   "days": ["mon"]} | {<type>: "interval", "seconds": 1800},
 "publish_topic": "schedule/<name>/fired"}

TYPE 3 - "dynamic": custom logic in Python
{"name": "<kebab-case>", "type": "dynamic", "description": "<what it does>",
 "poll_interval": <seconds>, "code": "<async setup/process/handle_task>"}

Rules:
- Use entity ids EXACTLY as they appear in the AVAILABLE HA ENTITIES list.
- Reuse a topic already published by an agent in AVAILABLE AGENTS rather than
  inventing a new one, and use that topic's exact payload field names.
- Delegate to an existing agent by name when one already does the job.
- HA state changes arrive on homeassistant/state_changes/#; filter by entity_id
  inside the payload (state is nested: payload['new_state']['state']).

```

A.5 Agent code generation

evalharness.py

CODEGEN_SYSTEM, 34 lines. Condensed for the harness; see below.

Appended at call time: The task description from the plan step being executed. As above, the deployed prompt is long and assembled at call time.

```

You write Python code for a dynamic agent in a multi-agent IoT framework.
Output ONLY Python code - no explanation, no markdown fences.
Define async functions among: setup(agent), process(agent),
handle_task(agent, payload), cleanup(agent).

AGENT API
agent.state                dict, persists across process() calls
await agent.publish(topic, data)  publish to MQTT
await agent.mqtt_get(topic)    ONE-SHOT read, returns one payload
await agent.log(msg)          dashboard log
await agent.alert(msg, severity) dashboard alert
agent.persist(key, value)     save to disk (NOT awaitable)
agent.recall(key)             load from disk (NOT awaitable)
agent.subscribe(topic, callback) continuous subscription (NOT awaitable)
agent.window(topic, seconds=N) sliding window object (NOT awaitable)
agent.declare_contract(publishes=[...], subscribes=[...])
await agent.llm.chat(prompt)   the framework's LLM - never import your own

CRITICAL RULES
- agent.subscribe() is NOT awaitable and REQUIRES a callback. It runs as a
  background task; the callback must be an async function taking exactly one
  argument (the payload).
  CORRECT: async def on_msg(payload): ...
            agent.subscribe('some/topic', on_msg)
  WRONG:   await agent.subscribe('some/topic', on_msg) # not awaitable
  WRONG:   agent.subscribe('some/topic')               # no callback
  WRONG:   async def on_msg(topic, payload): ...       # too many args
- agent.window() is synchronous: 'w = agent.window(t, seconds=300)', never await.
- agent.persist() / agent.recall() are synchronous - do not await them.
- process() is called repeatedly on a poll interval. Do NOT write 'while True'
  loops inside it; just do one iteration of work and return.
- Import libraries INSIDE functions, never at module level.
- Wrap blocking calls (cv2, torch, psutil-heavy work) in
  'await asyncio.get_event_loop().run_in_executor(None, fn)'.

```

B Acting versus refusing

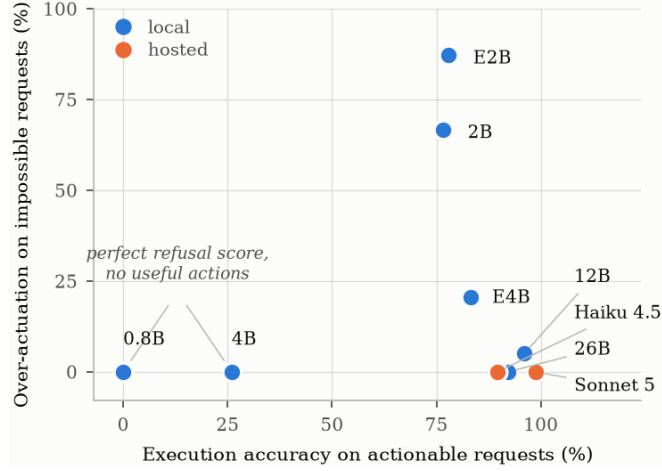


Figure 2: Acting versus refusing on the actuator call site ($n=77$ actionable, $n=39$ impossible). Each point is one model: execution accuracy on requests that should produce service calls, against the share of impossible requests that produced at least one call on real hardware. Deployable is the bottom right; the bottom left is a model that never actuates and so scores a perfect over-actuation rate, which is why neither axis is meaningful alone. Qwen3.5 2B and 4B fail in opposite directions and land at almost the same joint score: 2B never abstains but actuates on two thirds of the impossible requests, while 4B returns the empty action list on 56 of the 77 actionable requests and is correct on 20 of the 21 it answers.

C Sub-group breakdowns

Table 5: Actuator accuracy by cross-site sub-group. Prefixes $a_b_$ are the two installations; *control* is a device the site owns, *flip* one only the *other* site owns, *nearmiss* a plausible name for an absent device, *same* a device both own, *refusal* a device neither owns.

Model n	a_control 28	a_flip 7	a_nearmiss 6	a_refusal 6	a_same 3	b_control 8	b_flip 7	b_nearmiss 6	b_refusal 6	b_same 3
Qwen3.5 0.8B	0	29	100	100	0	0	71	100	100	0
Qwen3.5 2B	79	43	0	0	100	88	86	67	83	100
Qwen3.5 4B	36	29	100	100	33	12	71	100	100	33
Gemma4 E2B	86	57	0	17	100	88	14	0	33	100
Gemma4 E4B	93	71	83	100	100	100	71	67	83	100
Gemma4 12B	100	100	100	100	100	100	86	100	83	100
Gemma4 26B	100	86	100	100	100	100	100	100	100	100
Claude Haiku 4.5	96	100	100	100	100	100	100	100	100	100
Claude Sonnet 5	100	86	100	100	100	100	100	100	100	100

Table 6: Actuator accuracy by how the device was named: by *name*, by *location*, by both, by an *attribute* (“the lamp on the small table”), by the *switch* it hangs off, or as a *multi-device* request. *refusal* cases name a device the site does not own.

Model <i>n</i>	attribute 8	location 5	multi 2	name 6	name+location 4	refusal 8	switch 3
Qwen3.5 0.8B	0	0	0	0	0	100	0
Qwen3.5 2B	62	80	100	67	100	0	0
Qwen3.5 4B	38	0	50	17	25	100	33
Gemma4 E2B	50	40	50	100	100	25	33
Gemma4 E4B	62	60	0	83	75	75	100
Gemma4 12B	88	80	50	100	100	100	100
Gemma4 26B	75	100	50	83	100	100	67
Claude Haiku 4.5	62	40	50	100	100	100	100
Claude Sonnet 5	100	100	100	100	100	100	100

Table 7: Planner accuracy by sub-group. *infeasible* asks for a pipeline the site cannot support and is scored on refusing to plan one.

Model <i>n</i>	chain 4	create 6	entity 8	infeasible 5	reuseagent 7	reusetopic 10	schedule 5	topic 5
Qwen3.5 0.8B	0	0	12	20	14	10	0	0
Qwen3.5 2B	0	83	38	60	0	30	20	0
Qwen3.5 4B	0	67	62	80	14	80	20	60
Gemma4 E2B	25	100	25	80	43	60	20	0
Gemma4 E4B	50	100	12	100	43	60	20	20
Gemma4 12B	25	67	100	40	86	90	100	60
Gemma4 26B	50	100	100	80	86	100	100	60
Claude Haiku 4.5	75	100	100	80	71	100	80	20
Claude Sonnet 5	100	83	100	20	71	90	100	100

Table 8: Code generation accuracy by required API shape.

Model <i>n</i>	blocking 4	cleanup 3	contract 3	llm 3	oneshot 3	poll 3	publish 5	schema 4	state 5	subscribe 7	task 6	window 4
Qwen3.5 0.8B	25	67	33	0	0	33	40	25	40	0	33	0
Qwen3.5 2B	0	0	33	33	0	67	40	25	80	29	83	0
Qwen3.5 4B	0	0	0	33	0	33	60	75	60	43	83	0
Gemma4 E2B	50	33	33	67	67	0	20	0	20	43	67	0
Gemma4 E4B	75	67	100	33	100	33	60	100	100	100	100	50
Gemma4 12B	50	67	100	100	100	33	80	100	100	86	83	100
Gemma4 26B	100	67	100	33	100	67	100	75	100	71	83	50
Claude Haiku 4.5	100	100	100	33	100	100	100	100	80	100	100	100
Claude Sonnet 5	100	100	100	100	100	100	100	100	100	100	100	100

Table 9: Intent routing accuracy by sub-group.

Model <i>n</i>	crud 6	delegate 6	lexical 4	mixed 4	multi 2	query 4	sched 6	verbose 4
Qwen3.5 0.8B	17	67	75	50	100	25	33	75
Qwen3.5 2B	33	67	75	25	100	100	33	100
Qwen3.5 4B	83	83	100	100	100	100	83	100
Gemma4 E2B	67	67	100	100	100	75	83	100
Gemma4 E4B	67	83	100	100	100	100	100	100
Gemma4 12B	67	67	75	100	100	100	100	100
Gemma4 26B	83	67	100	100	100	100	100	100
Claude Haiku 4.5	50	67	75	100	100	75	100	100
Claude Sonnet 5	100	83	75	100	100	100	100	100

Table 10: Home Assistant action classification accuracy by sub-group.

Model <i>n</i>	camera 3	create 3	disable 3	edit 3	existence 4	hardware 3	list 4	state 3	unknown 2
Qwen3.5 0.8B	100	67	100	33	50	100	100	67	0
Qwen3.5 2B	100	33	100	67	100	100	75	100	0
Qwen3.5 4B	100	67	67	100	100	100	100	100	100
Gemma4 E2B	100	100	100	67	50	100	100	100	100
Gemma4 E4B	100	100	100	100	100	100	100	100	50
Gemma4 12B	100	67	33	100	100	100	100	100	100
Gemma4 26B	100	67	100	100	100	100	100	100	100
Claude Haiku 4.5	100	67	67	100	100	100	100	100	100
Claude Sonnet 5	100	100	67	100	100	100	100	100	50

D Where hosted spend goes

Table 11: Where hosted spend goes. The actuator prompt carries the site’s full entity list, so one call site dominates the bill for both hosted models — and it is the site a mid-sized local model already handles. Token counts are provider-reported and are not comparable across model families: the two tokenise identical prompts differently.

Call site	Claude Sonnet 5			Claude Haiku 4.5		
	Mean in tok.	\$	Share (%)	Mean in tok.	\$	Share (%)
Intent	499	0.06	0.5	348	0.01	0.5
HA action	721	0.06	0.5	509	0.02	0.5
Actuator	28 122	9.86	81.3	20 921	2.47	80.8
Planner	9 703	1.73	14.3	7 140	0.45	14.6
Codegen	814	0.42	3.5	576	0.11	3.7
Total		12.13			3.06	

E Model builds and serving

All seven local models run on the same host and the same runtime, so latency differences in Appendix F reflect the model, not the stack.

Table 12: Local model builds. *MTP* is multi-token prediction; *QAT* is quantization-aware training. Gemma4 26B is A4B: about 4B parameters active per token.

Model	Build	Quant.	Notes
Qwen3.5 0.8B	Qwen3.5-0.8B-UD-Q4_K_XL	Q4	MTP
Qwen3.5 2B	Qwen3.5-2B-UD-Q4_K_XL	Q4	MTP
Qwen3.5 4B	Qwen3.5-4B-UD-Q4_K_XL	Q4	MTP
Gemma4 E2B	Gemma-4-E2B-it-qat-GGUF	Q4	MTP, QAT
Gemma4 E4B	Gemma-4-E4B-it-qat-GGUF	Q4	MTP, QAT
Gemma4 12B	Gemma-4-12B-QAT-MTP	Q4	MTP, QAT
Gemma4 26B	Gemma-4-26B-A4B-it-qat-GGUF	Q4	MTP, QAT, MoE

Host: NVIDIA DGX Spark, GB10 (Blackwell), 128 GB unified memory. Runtime: llama.cpp build 9592, commit ac4cddeb0, fronted by LiteLLM v1.82.6. All builds are the Unsloth GGUF releases.

F Latency and throughput

Table 13: Mean wall-clock seconds per call, and aggregate generation throughput. Local latency is measured on the hardware in §3 with no batching; hosted latency includes network round-trip. Throughput aggregates tokens over time rather than averaging per-call rates. All seven local models share one host and one runtime (Appendix E), so these differences are the models. Gemma4 26B beats Gemma4 12B because it is a mixture of experts with about 4B parameters active per token, against 12B dense: a larger model that does less work per token.

Model	Intent	HA action	Actuator	Planner	Codegen	tok/s
Qwen3.5 0.8B	0.3	0.2	2.1	2.9	3.4	122
Qwen3.5 2B	0.3	0.1	3.0	4.7	6.2	48
Qwen3.5 4B	0.5	0.3	6.0	4.3	4.9	25
Gemma4 E2B	0.4	0.2	3.7	2.7	4.3	94
Gemma4 E4B	0.4	0.2	7.4	3.6	2.7	28
Gemma4 12B	0.4	0.3	14.0	9.0	3.2	14
Gemma4 26B	0.4	0.2	10.1	5.9	2.7	21
Claude Haiku 4.5	0.9	0.9	2.5	3.9	2.7	64
Claude Sonnet 5	2.1	2.5	3.2	5.3	4.6	43