Should SLMs Be in the Driver's Seat?

A Study of Local Models as Interpreters in Cloud-Written Agent Programs

Boris Radovič[1,2,*] **Marco Canini**[1] **Veljko Pejović**[2]
[1]King Abdullah University of Science and Technology
[2]University of Ljubljana, Faculty of Computer and Information Science
[*]boris.radovic@kaust.edu.sa

Abstract

The most accurate agents executing user-provided tasks are driven by large, cloud-hosted models, which solve them by reading the private data of the user. Agents powered by models small enough to run on edge devices keep that data local, but yield far worse results. A third option is to have cloud models write a program that devices execute: the data stays local, but as the program is written before any data is seen, it fails whenever a step depends on interpreting content that appears only at run time. We explore an intermediate architecture, in which the cloud-written program defers bounded questions over user data to a small local model through an ordinary function call. On AppWorld this improves the success rate of a code-only agent in all six combinations of backbone model and test partition we evaluate, by up to 14.9pp, recovering a substantial part of the gap to a cloud-hosted ReAct loop while keeping user data on the device.

1 Introduction

Users increasingly treat their smartphones and other edge devices as personal assistants [3, 9, 31], asking them to carry out a range of tasks and expecting these tasks to be carried out correctly, quickly, and efficiently. State-of-the-art methods address such requests with agents [32, 40], i.e., systems in which a language model (LM) interacts with the environment on behalf of the user. The LM acts as the "engine" of the agent, and choosing it involves a fundamental trade-off, since *which* LM is used also determines *where* such a model can be run. Large LMs (LLMs) obtain by far the best task success rates, but they are typically too large to fit in the memory of an edge device and/or too compute-intensive for such devices, and are thus served on the cloud. Small LMs (SLMs) can instead run locally on the device of the user, but, as is generally expected from the scaling behavior of LMs [18], are considerably weaker, especially in multi-step tasks [14].

Figure 1 quantifies the gap between SLMs and LLMs in agentic settings by running ReAct [40] agents on AppWorld [31] with two LM families (Gemma 4, Qwen 3.5). Consistent with results reported by Rainone et al. [26], the success of the agents degrades as the model size shrinks. No model below five billion parameters exceeds 5.9% success rate on the Test Normal and 3.8% on the Test Challenge partition, and the sub-two-billion LMs we evaluate solve no task at all. The sizes that make a local ReAct agent work reasonably well are also the ones that do not fit on edge devices: the 12B Gemma model reaches 73% success rate, but its weights alone require $\sim$6 GB in 4-bit precision, before the KV cache and before the memory the

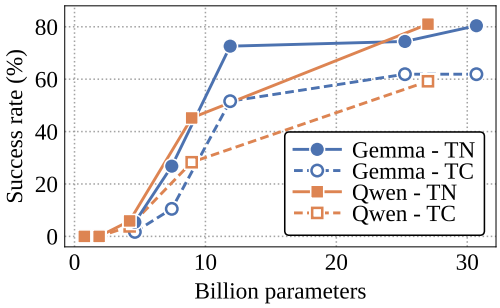


Figure 1: Success rate of the ReAct agent on the AppWorld benchmark on the Test Normal (TN) and Test Challenge (TC) partitions, w.r.t. the number of LM parameters.

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: SLMs for Agentic Systems.

rest of the system needs. Even if memory were not an issue, running the ReAct loop locally would incur a substantial energy cost [35], which on battery-powered devices is undesirable. Based on these observations, we argue that at present SLMs are not ready to "drive" agentic tasks on their own.

At first glance this settles the matter in favor of the cloud, and indeed most deployed assistants are built this way. Such a choice, however, is problematic because of *how* these agents consume data. In a ReAct-style loop, which is the dominant paradigm, the agent does not merely issue commands: rather, it inspects the outputs of the commands it issues and decides its next action based on what it has seen. For instance, when the agent issues a command to list the emails of the user, those emails are appended to the context and are processed in the next iteration. As ReAct agents issue short commands and constantly analyze their output, the result is a continuous flow of personal content leaving the device. Such a flow is problematic on several counts: it may violate the privacy of the user [9, 14], it consumes bandwidth, and it is expensive, since providers bill by the number of input and output tokens [19, 22]. Lastly, relying entirely on the cloud leaves the local compute untapped, whereas exploiting this resource can reduce cost by up to $30\times$ [22].

A less explored family of agents avoids this flow altogether. Instead of interleaving reasoning and observation, in a code-first agent the LLM writes a program for the user request, which then runs end-to-end on the edge device where the data resides [31, 35]. The cloud receives the user prompt and generates the script, but it never receives the data the program operates on. Thus, whenever a step towards solving the problem depends on properly interpreting content retrieved at run time, such an approach will likely fail: a program written before the data is read can only match patterns fixed in advance, and in numerous situations reliable patterns are difficult or impossible to pre-engineer.

The limitations of the two paradigms are thus of a rather different nature. Cloud-based ReAct agents achieve high success rates, but sacrifice data locality, while a code-first approach keeps the data local but fails whenever a task requires data interpretation. This difference suggests dividing the work along a boundary that conveniently coincides with the privacy one. The structure of a task, i.e., which APIs to call, in which order, and under which conditions, typically contains no private data [42] and is the part that genuinely benefits from a large model [16, 25]. The content the task operates on is private, but the reasoning it demands is often narrow [1, 22].

In this paper we explore this division: a cloud LLM writes a program for the user request and, wherever the program has to interpret user data, defers an "atomic" reasoning step, i.e., a single, stateless question over program-supplied data, to an SLM running locally. Similarly to Debenedetti et al. [7], we instantiate the design as a single Python function exposed to the LLM, which turns a semantic step into an ordinary call returning a typed value, so that the SLM is only asked isolated questions and never asked to plan. On AppWorld [31], this improves the success rate of a code-first agent in all six combinations of backbone LLM and test partition that we evaluate by between 2.2pp and 14.9pp, recovering a substantial part of the gap that separates code-first agents from ReAct.

2 Related Work

We summarize here the closest lines of work and defer the full discussion to Appendix A.

Programs instead of loops. PAL [11] and Program-of-Thoughts [6] showed that offloading the deterministic part of a task to an interpreter is more reliable than carrying it out in tokens. Trivedi et al. [31] bring the idea to agentic tasks, replacing the ReAct loop [40] with a single Python program. Such programs need not be pure code: ViperGPT and VisProg give the code-generating LLM API access to a set of vision models and let it compose their usage [13, 30], and CaMeL lets the program call a "quarantined" LLM to parse untrusted data as a defense against prompt injection [7]. All of them, however, run these models in the cloud, so execution never becomes local; ViperGPT and VisProg are moreover tied to a single domain.

SLM capabilities. SLM accuracy degrades as the context grows. This degradation is already pronounced when the SLM is only asked to select a tool for a single step of an agentic task [24, 45], let alone when it has to chain multiple such steps to reach a goal [14]. Consequently, systems that do run complete agentic tasks with SLMs make restrictive assumptions – TinyAgent, for instance, has a fixed set of only 16 tools [9]. A single question over a single bounded piece of data does not enter this regime. We thus confine the SLM to answering exactly such questions over the context the program hands it, which is precisely where small models have been shown to be reliable [1, 22].

Local-remote collaboration. Systems that split agentic tasks between device and cloud differ in the role they assign to the device: SLMs may extract information from documents for a remote aggregator [22], execute and verify the steps of a cloud-generated plan [41], or run code the cloud synthesizes from periodically uploaded sensor readings [8]; a separate line keeps the loop in the cloud and has a local model sanitize or re-write what is uploaded [20, 42, 43]. All of them either target a narrow domain or keep the remote model involved while the task runs, and none quantifies how often a code-first agent fails simply because its program cannot interpret the data it retrieves.

3 Cloud Planner with On-Device Interpreter for Edge Agentic Tasks

We consider two actors: a compute-powerful cloud node running an LLM and a compute-constrained edge device, which can only run an SLM – by SLM we indicate any LM that fits within the memory budget of the device, and by LLM any that does not. As there is a one-to-one mapping between models and nodes, we drop qualifiers and write LLM for the cloud model and SLM for the client model. The client receives agentic requests from the user and is tasked with carrying them out. In doing so, other than maximizing the accuracy, its desiderata are:

(1) minimize local compute, and thus energy consumption,
(2) keep the data used during execution local, and
(3) minimize LLM involvement.

The server has no direct interest in the first two points. The third, however, can be seen as shared by the two actors, though for different reasons: the client wants to minimize the number of LLM tokens to reduce cost, while the server may be interested in minimizing the effort spent on a client so as to serve a larger number of clients concurrently. That is, with the same infrastructure, by minimizing the resources (time, tokens, compute, memory) allocated to a client, the server can serve more clients.

With respect to the second desideratum, while the data the task operates on should always remain confined to the device, we assume the prompt does not contain private information and can be shared with the server. Otherwise, our method is compatible with the prompt sanitization mechanisms [20].

3.1 Execution loop

We study whether multi-step agentic tasks can be carried out by a script that runs end-to-end on the device: the LLM writes the program, and the SLM is queried only where a step requires interpreting data. Akin to the Full-Code (FC) agent [31], our framework proceeds as follows:

(1) The edge device sends the user request to the LLM, which selects the APIs the program will use.
(2) The LLM, given the request, the documentation of the selected APIs, and instructions, generates a Python script. The instructions expose the framework-provided `slm` function and, since running the SLM is by far the dominant cost on the device, advise using it only where a code-only approach would be unreliable or impossible. The LLM ships the resulting program to the device.
(3) The device runs the program by involving the SLM at the points in which the LLM determined it to be necessary. If an exception is raised, the stack trace is sent back to the LLM, which reasons over it and generates another version, returning to step 2 with the error added to its context.

The loop is capped at five attempts, after which the task is considered failed. If the initial API selection turns out to be incomplete, the LLM can extend it at run time without involving the device.

Table 1 positions our design: it interprets run-time data, which FC cannot; it invokes the local model only where needed, unlike local ReAct; and it uploads only the request and, on failure, the traceback.

Table 1: Qualitative comparison of the execution strategies.

Agent	Interprets run-time data	Local LM cost	User data to cloud	Cloud tokens	Cloud-device round trips
Cloud ReAct	✓	none	every observation	high	one per step
Edge ReAct	✓	high	none	none	none
Full-Code (FC)	✗	none	request, traceback	moderate	≤ 5
Ours	✓	low	request, traceback	moderate	≤ 5

3.2 Programmatic interface to the SLM

A challenge lies in how to make code interact with an SLM. Inspired by Surís et al. [30], who provide a programmatic interface to models for solving visual question answering tasks, we define an ordinary Python function named `slm` and make it available by describing it in the instructions we give to the LLM, much as CaMeL exposes its quarantined model [7]. Our goal is that, from the perspective of the program, the call *behaves as closely as possible to an ordinary Python function*, by returning a value of a declared type or format.

```python
def slm(
    query: str,
    context: Any,
    context_description: str | None = None,
    keys: list[str] | None = None,
    choices: list[str] | None = None,
    output_type: str = "str",
    optional: bool = False,
) -> Any:
```

A call passes a question (`query`) and the data it ranges over (`context`). The remaining arguments shape the input and constrain the output: `choices` restricts the answer to a fixed vocabulary, `output_type` declares the type into which the response is parsed and cast, `keys` projects away the fields of a JSON context that the question does not need, `context_description` supplies format or ordering information that cannot be recovered from the serialized data alone, and `optional` makes the function return `None` instead of raising when the answer is absent. The function then performs all the overhead steps before and after the SLM invocation – serialization, prompt construction, parsing, verification – leaving the program with a plain function call, usable wherever the LLM deems some data interpretation necessary.

If the context does not contain the answer, the SLM is required to report its absence rather than fabricate a value. As a result, unless the `optional` argument is set, the call raises a *not-found* exception [7], which is propagated to the LLM like any other exception.

We present qualitative examples showcasing the usage of the `slm` function in Appendix B.

4 Evaluation

We evaluate the SLM agent on the AppWorld benchmark and its Test Normal (TN) and Test Challenge (TC) partitions, where the latter contains more complex tasks. We consider three LLMs, used for code generation by the SLM and FC agents and for driving the loop in the ReAct agent, specifically the Qwen 3.6 27B, the Qwen 3.5 122B-A10B, and the Qwen3-coder-next LLMs. For compactness, we refer to these models as Qwen-27B, Qwen-122B, and Qwen-coder, respectively. For the SLMs, we consider Qwen 3.5 models with sizes of up to 9B parameters. We report the success rate (SR), which indicates the percentage of correctly executed tasks. Further details about the experimental setup are provided in Appendix C.

4.1 How does the methodology compare to ReAct and FC agents?

We compare the agent we propose against the FC and ReAct agents and report the SR of all three in Table 2. The results indicate that the SLM agent improves the performance of the FC agent on the TN partition by up to 14.9pp (in the case of Qwen-27B, from 45.8% to 60.7%). In the case of the TC partition, on the other hand, the improvements are much more modest.

However, not every task requires an SLM during its execution, as evidenced by the non-zero SR of the FC agent. When analyzing the performance only on TN tasks in which the planner decided to use the `slm` function, the SLM agent improves the SR of the FC agent from $0.0 \rightarrow 28.6$, $2.4 \rightarrow 38.1$, and $13.8 \rightarrow 48.3$ for the Qwen-coder, Qwen-122B, and Qwen-27B models, respectively.

One of the driving motivations of the SLM agent is that having the local SLM answer only atomic questions is computationally lighter for the client than running an entire ReAct loop on-device. Figure 2 shows this directly: with the same 9B model on the client, the SLM agent has a higher SR than a purely local ReAct agent (52.4% vs 45.2%) while the client processes on average between one and two orders of magnitude fewer input tokens – the lower value under full prefix reuse – and generates $1.8\times$ fewer output tokens.[1] The saving is concentrated in input tokens because the ReAct agent must read the output of every command it issues, whereas the SLM only ever sees the values

[1] All Qwen SLMs run in thinking mode; token counts include reasoning tokens. Since `slm` returns only a scalar or short string, its output tokens are likely reducible via prompting or a non-reasoning model, though we have not evaluated this.

Table 2: SR for the considered agents and backbone LLMs and corresponding standard error, estimated with bootstrapping.

LLM	Agent	TN	TC
Qwen-coder	ReAct	59.5 ± 3.7	29.5 ± 2.2
	FC	33.3 ± 3.6	13.4 ± 1.7
	SLM	38.7 ± 3.7	15.6 ± 1.8
Qwen-122B	ReAct	57.1 ± 3.9	38.6 ± 2.4
	FC	42.3 ± 3.8	25.7 ± 2.1
	SLM	52.4 ± 3.9	28.8 ± 2.2
Qwen-27B	ReAct	81.0 ± 3.0	59.2 ± 2.4
	FC	45.8 ± 3.9	36.2 ± 2.4
	SLM	60.7 ± 3.8	41.5 ± 2.4

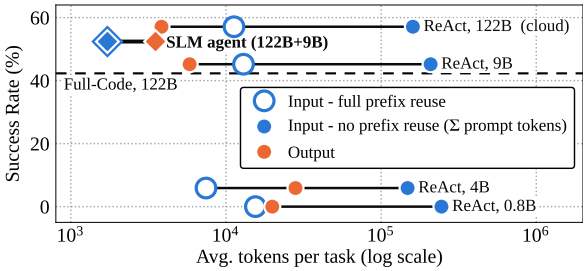


Figure 2: SR vs. avg. tokens per task processed by the local model for Qwen-122B on TN. Each row represents one agent. For the SLM agent, only tokens of `slm` calls are counted. Cloud ReAct (122B, cloud-side tokens) and FC (dashed line, no local tokens) are shown for reference.

the cloud-generated program passes to it. A fully cloud-based ReAct execution reaches a somewhat higher success rate (57.1%), but requires every token to be processed by the 122B model.

Because the SLM agent reduces the number of input and output tokens, its peak context is about 3× smaller than that of an all-local ReAct agent using the same 9B model (median 3.7k vs 12.0k tokens). As a consequence, the SLM also requires less memory to keep the context.

By construction, the FC agent uses no tokens on the client. On the cloud, our instructions are a superset of the FC ones, adding, per program, 3.7k tokens describing `slm` and its best practices: our agent thus requires ∼32% more input tokens than FC, and slightly more output tokens, though never by more than 10%. These extra instructions are also a confound, as the SLM agent improves on tasks that never invoke `slm` (e.g., +1.5pp for Qwen-122B), a gain the mechanism cannot explain. This accounts for at most a fraction of the gains we report, leaving the majority attributable to the SLM.

4.2 How large does the SLM need to be?

In the previous section we considered a 9B client LM, which is already too large for many devices. Figure 3 shows that the SLM need not be this large. We plot how often the value returned by `slm` agrees with the one the 9B model produced on the identical request, replayed offline so that only the SLM varies. The 9B is a reasonable reference: an LLM-as-a-judge evaluation finds its answers correct about 95% of the time on the TC partition. Decoding is not bit-reproducible across GPUs, so we also replay the 9B against its own trace; this self-comparison is the noise floor of the measurement.

For constrained output formats – booleans, selections among candidate items, and numeric or short free-form strings – a 2B model returns the identical value in 91.6% of calls, 5.4pp below the 9B model's own reproducibility (97.0%). Only at 0.8B is there a marked drop. Excluded from those numbers are the calls where the planner asks for a JSON object or list despite the `slm` instructions forbidding it; agreement there is far lower, but so is the 9B's with itself (69.9%).

Table 3: SR on TN w.r.t. SLM size.

SLM size	Qwen-coder	Qwen-122B	Qwen-27B
None (FC)	33.3 ±3.6	42.3 ±3.8	45.8 ±3.9
0.8B	33.9 ±3.8	48.8 ±3.9	52.4 ±3.9
2B	36.9 ±3.7	51.2 ±3.9	61.3 ±3.7
9B	38.7 ±3.7	52.4 ±3.9	60.7 ±3.8

The above is an agreement study, not an end-to-end one, as the values `slm` returns re-enter the program and change what happens next. Table 3 reports the SR when repeating the evaluation end-to-end. The 2B and 9B LMs are within one standard error for all three planners. The 0.8B model, on the other hand, loses more, but it still offers benefits over FC for Qwen-122B and Qwen-27B. Note that the sub-2B models solve no task at all when driving a ReAct loop (cf. Figure 1), yet with the Qwen-27B planner they improve FC by 6.6pp (0.8B) and 15.5pp (2B), suggesting that *answering questions over program-supplied data is better aligned with the current capabilities of SLMs than full planning*.

4.3 Where does the gap to ReAct come from?

Several factors contribute to the lower SR of the SLM agent relative to ReAct. We discuss the two we can measure. The first is that generated programs are not idempotent: attempts are not isolated, so a

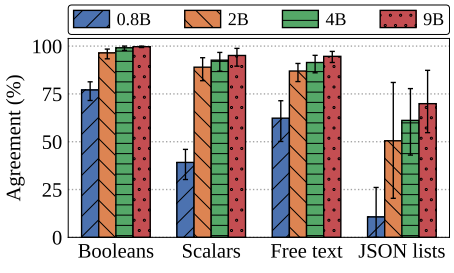


Figure 3: Agreement of Qwen SLMs with Qwen-9B; the 9B bar is self-agreement. Scalar types are `float`, `int`, and `item`.

Table 4: Share of failing programs aggregated over TN, TC, and all LLMs, whose exception recurs in the agent's next program. n counts failures that had a successor; each task's final program is excluded.

	FC		SLM	
Failure type	n	rep. (%)	n	rep. (%)
API error, 422	1070	35.6	966	34.3
Other API error	205	16.1	174	15.5
API misuse	310	8.1	299	10.7
Shape / attribute	386	13.7	371	14.0
Safety guard	76	10.5	56	10.7
SLM: info. not found	–	–	448	55.6

program that would be correct on the initial state can fail because an earlier attempt already modified that state, e.g., by sending a message. Re-executing the final program of each affected task in a fresh environment bounds this effect at 1.8pp in any single configuration and under 0.5pp on average, so cross-attempt state accumulation does not materially affect our results.

The second is that the planner is not told enough when an exception occurs. On failure, the stack trace is returned to the LLM, which generates the next program; for a not-found error the SLM agent additionally labels the cause as ambiguous question, wrong context, insufficient context, or unclassified. To test whether this is enough, we group failing programs by exception type and measure how often the next program the planner generates raises an exception of the same type; a recurrence indicates that the planner could not work out the cause from what it was told. Table 4 shows that a not-found error recurs 55.6% of the time, against at most 35.6% for any other type. A recurrence does not lower SR by itself, but it consumes one of the five generation attempts and grows the context with a trace carrying nothing the planner did not already have. The two code agents agree within 3pp on every exception type they share, so delegating to the SLM does not degrade ordinary recovery.

5 Conclusion and Future Work

We studied a division of labor in which a cloud LLM writes the program for an agentic request and a local SLM is invoked from within it, only where a step requires interpreting user data. This recovers much of what code-first agents lose by writing programs before observing any data, and, given a capable planner, is both more accurate than a local ReAct agent using the same model and substantially cheaper on the client. This paper reports an early instantiation of the design; below we discuss the open problems it surfaces and how we plan to address them.

Privacy. By construction, the only data that leaves the device is the user request and, on failure, the traceback, which the runtime emits and which may therefore embed user content. Analyzing the 2,917 exceptions raised across all our runs on the TC partition, we find such content – messages, notes, or file contents – in 12 of them (0.4%). The majority of exceptions carry coarser information: timestamps echoed back when parsing fails, or negative facts such as "file X does not exist". Still, what leaves the device should be guaranteed rather than incidental. We thus plan to have the SLM sanitize the traceback locally, so that every upward payload is declared.

Closing the gap. The design does not yet match a cloud-hosted ReAct loop in terms of success rate. Part of the remaining gap comes from how little the planner learns when a program fails. All types of exceptions recur, but the effect is most pronounced when the SLM cannot find the requested information in the context (Table 4). We thus plan to have the SLM inspect the variables at the point when the exception happens and let it explain what went wrong, so that the planner can more easily generate a corrected program. Doing so safely is hard: such an explanation is derived from private content, so it must convey why the call failed without reproducing the data itself. Alternatively, the SLM could attempt to recover from the failure on the device, where it has full access to the runtime state, avoiding the problem altogether, and escalate to the cloud only if local recovery fails. LMs can handle runtime errors [29], and weaker models can reuse programs written by stronger ones [2]; we thus also plan to study whether an SLM can recover from failures in the programs our planner writes.

Performance on edge devices. Our resource claims rest on token and context measurements, which are only proxies for what matters on devices, e.g., latency and energy. Fewer tokens mean less compute and a smaller KV cache, so we expect the reductions we reported to carry over to those metrics. The actual savings, however, depend on hardware, which notoriously varies widely across edge devices. We plan to measure them once the method is developed into a full framework.

Acknowledgments and Disclosure of Funding

This work was in part supported by the Slovenian Research Agency through the "approXimation for adaptable distributed artificial intelligence" (N2-0393) project.

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, 2025. arXiv:2506.02153.

[2] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *ICLR*, 2024.

[3] Wei Chen and Zhiyuan Li. Octopus v2: On-device Language Model for Super Agent. 2024. arXiv:2404.01744.

[4] Wei Chen, Zhiyuan Li, Zhen Guo, and Yikang Shen. Octo-Planner: On-Device Language Model for Planner-Action Agents. In *EMAS*, 2025. doi:10.1007/978-3-032-18011-7_9.

[5] Wei Chen, Zhiyuan Li, and Mingyuan Ma. Octopus: On-device language model for function calling of software APIs. In *NAACL*, 2025. doi:10.18653/V1/2025.NAACL-INDUSTRY.27.

[6] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *Trans. Mach. Learn. Res.*, 2023.

[7] Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. In *SaTML*, 2026. doi:10.1109/SaTML68715.2026.00040.

[8] Chinmaya Kumar Dehury, Siddharth Singh Kushwaha, Qiyang Zhang, Alaa Saleh, and Praveen Kumar Donta. LLM-assisted Agentic Edge Intelligence Framework. 2026. arXiv:2604.09607.

[9] Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Richard Charles Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. TinyAgent: Function Calling at the Edge. In *EMNLP*, 2024. doi:10.18653/V1/2024.EMNLP-DEMO.9.

[10] Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-Act: Improving Planning of Agents for Long-Horizon Tasks. In *ICML*, 2025.

[11] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided Language Models. In *ICML*, 2023.

[12] Lorenzo Giusti, Ole Anton Werner, Riccardo Taiello, Matilde Carvalho Costa, Emre Tosun, Andrea Protani, Marc Molina, Rodrigo Lopes de Almeida, Paolo Cacace, Diogo Reis Santos, and Luigi Serio. Federation of Agents: A Semantics-Aware Communication Fabric for Large-Scale Agentic AI. In *ER (NeurIPS Workshop)*, 2025.

[13] Tanmay Gupta and Aniruddha Kembhavi. Visual Programming: Compositional Visual Reasoning Without Training. In *CVPR*, 2023. doi:10.1109/CVPR52729.2023.01436.

[14] Mohd Ariful Haque, Fahad Rahman, Kishor Datta Gupta, Khalil Shujaee, and Roy George. TinyLLM: Evaluation and Optimization of Small Language Models for Agentic Tasks on Edge Devices. 2025. arXiv:2511.22138.

[15] Pranav Harikumar. PlanCompiler: A Deterministic Compilation Architecture for Structured Multi-Step LLM Pipelines. 2026. doi:10.21203/rs.3.rs-9186427/v1.

[16] Xinming Hou, Mingming Yang, Wenxiang Jiao, Xing Wang, Zhaopeng Tu, and Wayne Xin Zhao. CoAct: A Global-Local Hierarchy for Autonomous Agent Collaboration. 2024. arXiv:2406.13381.

[17] Rohan Kadekodi, Zhan Jin, Keisuke Kamahori, Yile Gu, Sean Khatiri, Noah H Bayindirli, Sergey Gorbunov, and Baris Kasikci. AgentFlux: Decoupled fine-tuning & inference for on-device agentic systems. 2025. arXiv:2510.00229.

[18] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. 2020. arXiv:2001.08361.

[19] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM Compiler for Parallel Function Calling. In *ICML*, 2024.

[20] Siyan Li, Vethavikashini Chithrra Raghuram, Omar Khattab, Julia Hirschberg, and Zhou Yu. PAPILLON: Privacy Preservation from Internet-based and Local Language Model Ensembles. In *NAACL*, 2025. doi:10.18653/V1/2025.NAACL-LONG.173.

[21] Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, Jun Wang, and Weinan Zhang. Hammer: Robust Function-Calling for On-Device Language Models via Function Masking. 2024. arXiv:2410.04587.

[22] Avanika Narayan, Dan Biderman, Sabri Eyuboglu, Avner May, Scott W. Linderman, James Zou, and Christopher Ré. Minions: Cost-efficient Collaboration Between On-device and Cloud Language Models. In *ICML*, 2025.

[23] Dhruvi Paprunia, Vansh Kharidia, and Pankti Doshi. Advancing SLM Tool-Use Capability using Reinforcement Learning. In *AIC*, 2025. doi:10.1109/AIC66080.2025.11211997.

[24] Varatheepan Paramanayakam, Andreas Karatzas, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. Less is More: Optimizing Function Calling for LLM Execution on Edge Devices. In *DATE*, 2025. doi:10.23919/DATE64628.2025.10992798.

[25] Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-Needed Decomposition and Planning with Language Models. In *Findings of NAACL*, 2024. doi:10.18653/V1/2024.FINDINGS-NAACL.264.

[26] Corrado Rainone, Davide Belli, Bence Major, and Arash Behboodi. When Cloud Agents Meet Device Agents: Lessons from Hybrid Multi-Agent Systems. 2026. arXiv:2605.30102.

[27] 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 *NeurIPS*, 2023.

[28] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. In *NeurIPS*, 2023.

[29] Zhensu Sun, Haotian Zhu, Bowen Xu, Xiaoning Du, Li Li, and David Lo. Toward agentic runtime healing. *Commun. ACM*, 2026. doi:0.1145/3800691.

[30] Dídac Surís, Sachit Menon, and Carl Vondrick. ViperGPT: Visual Inference via Python Execution for Reasoning. In *ICCV*, 2023. doi:10.1109/ICCV51070.2023.01092.

[31] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. In *ACL*, 2024. doi:10.18653/V1/2024.ACL-LONG.850.

[32] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable Code Actions Elicit Better LLM Agents. In *ICML*, 2024.

[33] Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents. In *NeurIPS*, 2023.

[34] Xiaolong Wei, Yuehu Dong, Xingliang Wang, Xingyu Zhang, Zhejun Zhao, Dongdong Shen, Long Xia, and Dawei Yin. Beyond ReAct: A Planner-Centric Framework for Complex Tool-Augmented LLM Reasoning. In *AAAI*, 2026. doi:10.1609/AAAI.V40I40.40676.

[35] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. AutoDroid-V2: Boosting SLM-based GUI Agents via Code Generation. In *MobiSys*, 2025. doi:10.1145/3711875.3729134.

[36] Yihan Wen and Xin Chen. PseudoAct: Leveraging Pseudocode Synthesis for Flexible Planning and Action Control in Large Language Model Agents. 2026. arXiv:2602.23668.

[37] Simon Willison. The Dual LLM pattern for building AI assistants that can resist prompt injection. `https://simonwillison.net/2023/Apr/25/dual-llm-pattern/`, 2023.

[38] Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. ReWOO: Decoupling Reasoning from Observations for Efficient Augmented Language Models. 2023. arXiv:2305.18323.

[39] Qingwen Yang, Zhi Li, Jiawei Tang, Yanyi Liu, Tiezheng Guo, and Yingyou Wen. LEAF: A lightweight edge agent framework with expert SLMs for the industrial internet of things. *Comput. Mater. Contin.*, 2026. doi:10.32604/cmc.2025.074384.

[40] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*, 2023.

[41] Biao Yi, Xueyu Hu, Yurun Chen, Shengyu Zhang, Hongxia Yang, and Fan Wu. EcoAgent: An Efficient Device-Cloud Collaborative Multi-Agent Framework for Mobile Automation. In *AAAI*, 2026. doi:10.1609/AAAI.V40I35.40230.

[42] Guangsheng Yu, Qin Wang, Rui Lang, Shuai Su, and Xu Wang. PlanTwin: Privacy-Preserving Planning Abstractions for Cloud-Assisted LLM Agents. *IEEE Trans. Serv. Comput.*, 2026. doi:10.1109/TSC.2026.3695494.

[43] Liangqi Yuan, Wenzhi Fang, Shiqiang Wang, and Christopher G. Brinton. PAAC: Privacy-Aware Agentic Device-Cloud Collaboration. 2026. arXiv:2605.08646.

[44] Qizheng Zhang, Michael Wornow, and Kunle Olukotun. Agentic Plan Caching: Test-Time Memory for Fast and Cost-Efficient LLM Agents. In *NeurIPS*, 2025.

[45] Xiaoyu Zhang, Yi Wang, Along He, Haobin Wang, and Tao Li. TAPE: A Multi-Agent Framework for Task-Adaptive Planning and Execution in Resource-Constrained Environments. *Expert Syst. Appl.*, 2026. doi:10.1016/J.ESWA.2025.129423.

A Extended Related Work

ReAct, planning, programming. The predominant way of building agents is the ReAct loop, in which the agent reasons, emits an action – expressed as text, JSON, or Python – and observes its result, repeating until the task is completed [32, 40]. Since larger models yield better agents, especially in multi-step tasks [14], such a loop is typically driven by an LLM served in the cloud, with the cost and privacy consequences discussed in Section 1. Running the loop on edge devices is impractical for the

opposite reasons: accuracy is lower, the context grows with every observation and may overwhelm the client memory [45]. Also, involving the model at every action is compute-demanding [35].

A separate line of research decomposes the task into smaller steps, either during execution when a step fails [25] or, more commonly, in an initial planning stage carried out before any data is observed, as in ReWOO [38]. The plan may be a linear sequence, so that each step is handled by a "more focused" agent [4, 10, 16, 33, 36] or so that thinking can be cached and reused [44], or a directed acyclic graph whose nodes are steps and whose edges are data dependencies [9, 12, 19, 28, 34]. Since nodes may be traditional functions, the LM need not be involved in every step, which makes these methods more efficient than ReAct in terms of tokens [19, 38] and in some cases even more accurate [38]. The relief is however partial: every node that involves language understanding is still a separate *cloud* invocation over the data that node retrieves.

Such plans are, at the end of the day, no more than programs, i.e., defined sequences of steps to achieve a goal. PlanCompiler makes this explicit, as it "compiles" an LLM-generated plan into a Python script [15], while PAL and Program-of-Thoughts had already shown that offloading the deterministic part of a task to an interpreter is more reliable than carrying it out in tokens [6, 11]. Closest to our approach, ViperGPT and VisProg give the code-generating LLM API access to a set of vision models and let it compose their usage in code [13, 30], CaMeL lets the program delegate the parsing of untrusted data to a quarantined LLM [7, 37], while Trivedi et al. [31] explore carrying out agentic tasks using Python code alone. These methods, however, are either tied to a single domain [13, 30], keep the cloud involved whenever data must be interpreted [7], or run programs written without observing any data, which fail whenever a step depends on content available only at run time [31]. CaMeL, in particular, is methodologically similar to our approach: a "privileged" LLM writes code that may delegate the parsing of data to a "quarantined" LLM [7]. The goal of CaMeL, however, is security, specifically preventing prompt injection, rather than data locality. While the authors note that the quarantined LLM could run locally, they evaluate the method only with cloud models; whether models small enough for edge devices can fill this role, and at what cost to the device, is the question we study.

SLMs for tool calling and agentic tasks. A natural candidate to make raw data visible at run time without exposing it to the cloud is a local SLM. Turning an LM into an agent requires selecting a function and generating the arguments that carry out a task in the environment [5, 27], and several works study this task specifically on edge devices [3, 17, 21, 23]. Accuracy, however, degrades as the number of tools grows, since every tool description adds context [24, 45]; most methods therefore either add a procedure that selects which descriptions to present, or start from a small set of tools. Long contexts of this kind are also expensive in energy and compute on the device [4].

In any case, these methods are concerned with a single step, for which they need an external planner [4]. Some approaches instead target complete agentic tasks. TinyAgent brings the DAG-based planning approach above to edge devices, showing that finetuned SLMs between 1 and 7B parameters can compete with vanilla GPT-4-Turbo [9]; the price is a fixed and very limited set of tools (16 in the paper), a curated finetuning dataset which is expensive to build and must be regenerated when new tools appear, an additional procedure to select which function definitions to provide in the prompt, and no recovery mechanism when the initial plan is wrong. Yang et al. [39] instead combine three SLMs with different roles to carry out agentic tasks in IIoT settings.

What these results show is not that SLMs are unusable, but that they suffer as the context grows, for instance when selecting a function among a few dozen descriptions. Given issues in such a narrow regime, and given that the methods carrying out complete agentic tasks with SLMs succeed only under heavy scaffolding – a small fixed tool set, a curated finetuning dataset, or an explicit tool-filtering stage – a natural conclusion is that *at present* SLMs are not appropriate for driving general agentic tasks on their own. A single question asked over a single, bounded piece of data does not enter this regime, and this is precisely the one in which small models are known to be reliable [1, 22] – a property we exploit.

Local-remote agent collaboration. Since neither placing the whole loop in the cloud nor running it on the device is satisfactory, some systems split the work between the two. In MinionS a local SLM extracts information from documents as instructed by a remote LLM, which aggregates the results [22]. In EcoAgent the LLM generates a sequential plan specifying input and output conditions for every step, and a local SLM carries the steps out and verifies that the progression adheres to the

expectation [41]. In LEI the edge device periodically uploads sensor readings to the cloud, where the LLM synthesizes the code that is then used to analyze the data directly on the device [8]. In other settings the loop stays in the cloud and a local model sits in front of it, sanitizing the data by replacing sensitive information with placeholders [42, 43] or by re-writing the prompt [20] before uploading.

What is missing is the intersection of these lines: a division that keeps the cloud LM out of the loop during execution, as code-first agents do, while still allowing the program to reason over data that only exists at run time. The design we evaluate differs from the above along several axes at once: it studies an explicitly asymmetric placement, with a large planner in the cloud and a small executor on the device; it recovers from errors by regenerating the program from the traceback rather than by re-planning blindly; it makes no domain-specific assumptions, unlike collaborative systems studied so far in narrow domains such as data analysis or sensing; and it employs the SLM for the kind of question small models already answer reliably, rather than introducing finetuning or tool-filtering machinery to make them fit a role they are not suited for. To our knowledge, no existing method combines these properties, and no prior work has quantified how often code-first agents fail simply because the program cannot interpret the data it retrieves.

B Examples of the `slm` function usage

To illustrate the kind of task the proposed architecture targets, we take two examples from the AppWorld benchmark. Both are cases in which a task requiring tens of steps can be carried out almost entirely by pure Python, with an LM being strictly necessary at one or two points only. This is not meant as evidence that the approach works in general, which we quantify in Section 4, but as an existence proof of the pattern the method exploits.[2] In both examples, API keys are necessary to access applications, read local files, etc., all of which are by nature private [42].

Example 1. *Kristin paid for my grocery recently as my payment cards were not working at the time. Send them the owed money with a description note "Groceries" as per my phone text conversation, and then send them a phone text message, "It is done."*

```
amount_owed = slm(
    "What is the dollar amount that Kristin said I owe her for groceries?",
    context=messages, # fetched from the messages API
    context_description="all text messages between me and Kristin",
    keys=["message"], # extract bodies, remove metadata
    output_type="float",
)
```

Of the 17 steps the emitted program performs, all but one are deterministic: resolving the contact, fetching the conversation, issuing the payment, and sending the confirmation message are all plain API calls with known arguments. The exception is recovering how much is owed, which is stated somewhere in the conversation in natural language. Extracting it with a regular expression is unreliable, since a regex cannot tell the amount apart from a look-alike distractor elsewhere in the thread. An LM is thus needed – but note *which* one: answering this question requires reading the entire recent message history, so delegating it to the cloud means uploading the conversation in order to obtain a single number. With our design the program instead calls `slm` at that one point.

Example 2. *I paid for our last month's electricity bill. Its amount is supposed to be shared equally among my roommates and me. Make venmo requests to my roommates, with a description note, "For electricity bill.". The bill receipt is in my file system.*

```
electricity_bill_path = slm(
    "Which of these files is the electricity bill for last month?",
    context=files, # survivors of a keyword filter
    output_type="item", # return one element of files
)
```

In the second example, the user never says which file contains the receipt, and the name is unknown at generation time – it may be `May23.txt`, `May_23.txt`, `05_2023.txt`, or anything else, anywhere in the file system. Left to guess, the LLM hallucinates a plausible path, which is what we observe in practice. The division of labor we advocate resolves this cheaply: pure Python walks the file system and keeps the files whose name contains "electric", "bill", or a few related terms, and the SLM is asked a single question over the handful of survivors. Guessing a keyword set is recoverable in a way that guessing a path is not, since the filter

[2] These are not made-up examples, but actual programs we obtained by running the proposed method. We only introduced comments in the listings for clarity.

is deliberately high-recall and its failures surface as a wrong or empty candidate set rather than as a non-existent path.

```
total_amount = slm(
    "What is the total amount of the electricity bill?",
    context=bill["content"],
    output_type="float",
)
```

Having identified the file containing the receipt, the program must extract the amount that was paid. As in Example 1, a regular expression is not a reliable option, and for the same underlying reason: the program is written before any file is read, so its layout is unknown at generation time. A receipt may list several currency values, such as an amount before taxes, a tax line, and a total, and there is no guarantee that the figure of interest carries a dollar sign at all, as in "Total ($): 122.30". Distinguishing the total from the other values requires reading the labels around it, which is exactly the kind of interpretation a pattern fixed in advance cannot perform. The generated program therefore delegates this step to the SLM as well.

C Reproducibility details

We serve every model locally with `llama.cpp`, compiled from source (build `b10162`, commit `6ba5ef247`) separately for the A100 and V100 GPUs. Table 5 lists the checkpoints and the hardware. Each model runs on a single GPU, without tensor parallelism.

Table 5: Employed models. All weights are GGUF files (UD = Unsloth Dynamic quantization).

Model	Quantization	GPU
Qwen-27B (Qwen3.6-27B)	UD-Q8_K_XL	A100-80GB
Qwen-122B (Qwen3.5-122B-A10B)	UD-Q4_K_XL	A100-80GB
Qwen-coder (Qwen3-Coder-Next)	Q6_K	A100-80GB
Qwen3.5-0.8B/2B/4B/9B	UD-Q8_K_XL	V100
Gemma-4 E2B–31B	UD-Q8_K_XL	A100-80GB

Generation. Decoding is greedy: every request uses temperature 0, and we set neither top-p nor top-k. Qwen-27B and Qwen-122B models produce a thinking trace before answering. Qwen-coder produces none, and Gemma-4 runs with thinking disabled, its chat-template default. To stop runaway generations, planner and ReAct outputs are capped at 16,384 tokens, thinking included. For the SLM we cap thinking itself at 4,096 tokens; the server then appends a short wind-down message and closes the thinking block, so the model still answers.

Protocol. We evaluate on the 168 Test Normal and 417 Test Challenge tasks of AppWorld (data v0.2.0) and report success rate (referred to as "task goal completion" in the original repository). The evaluator is unchanged from upstream commit a072b7a, which our fork is based on. We run each configuration once. Standard errors come from bootstrapping over tasks: 10,000 resamples, with the SE taken as the standard deviation of the resampled means.

Baseline Agents. FC is AppWorld's full-code agent. Besides minor fixes, we extend it in two ways, and our agent inherits both. First, a single planner call, prompted with three training tasks as demonstrations, shortlists up to 20 APIs. The list is completed with the login and ID-provider APIs it depends on, and it is cached, so FC and our agent receive the same shortlist for a given planner. Second, the planner may look up API documentation at run time through documentation-only programs, which run on the server, as the documentation contains no user data. The first four such programs of a task do not count as attempts, although each one is a planner call. Each agent then makes at most five attempts, i.e., generates at most five programs that are executed on the device; the shortlisting call above runs entirely in the cloud and does not count as an attempt. After every attempt that raises an error, the planner receives the untruncated traceback, together with AppWorld's reflexion prompt, and the whole conversation is kept. The ReAct baseline is AppWorld's ReAct agent, unmodified. It runs for at most 50 steps and never truncates observations or history.

SLM answer quality study. For the agreement study, we take every 9B round-trip recorded in the three TN runs: 1,584 in total (676 `bool`, 663 `str`, 209 `item` and 36 numeric). We replay each one on

every candidate SLM with the same prompt and decoding settings. Two answers agree when their outcomes match and, if both succeed, their cast values are equal. Floats must match exactly. Strings are compared after removing the thinking trace, canonicalizing JSON and collapsing whitespace. Error bars are 95% percentile intervals over 1,000 bootstrap resamples of tasks.