

SchemaWeaver: Accelerating On-Device Mobile Agentic AI via Schema-Guided Parallel LLM Decoding

Boyuan Yang
University of Pittsburgh
Pittsburgh, PA, USA
by.yang@pitt.edu

Wei Gao
University of Pittsburgh
Pittsburgh, PA, USA
weigao@pitt.edu

ABSTRACT

Large language models (LLMs) on mobile devices can act as agents that call external software tools to complete complex tasks, but the major challenge of supporting on-device mobile agentic AI with LLMs is the long latency of LLM tool calling on mobile devices. In particular, the decode phase is the main contributor of such long latency, but is hard to speed up on mobile processors as it is bound by memory bandwidth rather than compute power. To address this limitation, in this paper we present SchemaWeaver, a new on-device mobile agentic AI technique that enforces schema-guided parallel decoding in the decode phase of LLM tool calling, to reduce its latency by producing multiple tokens in each decode pass. SchemaWeaver can be generically applied to the LLM inference engine on mobile devices without involving any training efforts or changes on LLMs, and our experiment results on major mobile SoCs and LLM families show that SchemaWeaver can reduce the LLM tool calling latency by up to 3.9 times, compared to the competitive baselines.

1 INTRODUCTION

In recent years, Large Language Models (LLMs) have been used as the backbone of *agentic AI* systems. When given proper context, LLM agents can interpret user instructions, select the proper external software tools, and call these tools to complete complex tasks [27, 52, 53, 77]. For example in Figure 1, when being asked for driving directions, the LLM agent selects the `get_route` software tool from the available tools for this task. It then calls this tool for route planning and returns the results back to the user in natural language. In this process, the LLM agent perceives each tool from its *schema*, which is a machine-readable description that gives the tool’s description and the names and types of its arguments [27]. The LLM agent, then, calls the tool by generating the text command with values of the tool’s arguments.

Currently, LLM agents on mobile devices (e.g., smartphones) have become increasingly popular, allowing intelligent and automated operations on the user’s personal data [11, 47, 62]. Agentic AI functionalities on mobile devices have also been supported by mainstream mobile OSes, via Android AppFunctions [2, 3] and iOS App Intents [4]. In particular, running on-device LLM agents on local devices well

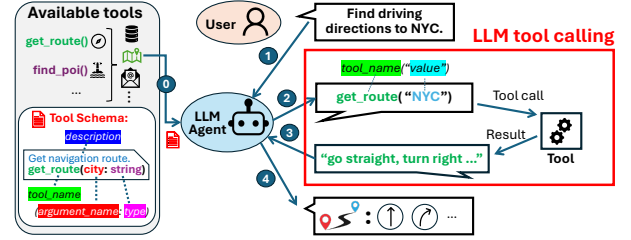


Figure 1: Agentic AI systems based on LLM agents that call external software tools

preserves the user’s data privacy [37, 73, 78], and quantized LLMs with 1B-3B parameters, which fit in a smartphone’s memory budget [19, 22, 39, 42], have been capable of launching tool calls with reasonable accuracy [25, 49, 80].

The major challenge of supporting on-device mobile agentic AI, however, is the long latency of LLM tool calling on mobile devices. In inference, an LLM produces its output through repeated forward passes that constitute a *prefill phase* and a *decode phase*. The prefill phase uses the first pass to read the input prompt, and the decode phase then generates the response using the other passes, each of which produces one output token [51, 72]. As shown in Figure 2(a), in traditional conversation-based LLM uses, one prefill is usually followed by a large quantity of decoded text tokens that the user reads as they arrive, and most of the decode latency is hence “hidden” in user reading [40, 68]. Agentic LLM operations, in contrast, first decode a structured tool call command for tool execution, and text tokens in response can only be produced after the completion of tool execution. The decode latency experienced by the user, hence, becomes significantly longer, especially when executing complicated agentic tasks such as on-device content search that involves multiple tool calls of different types. In these tasks, our experiments show that the decode latency constitutes up to 81% of the wall-clock time being taken (Sec 2.2).

Unfortunately, it is difficult to accelerate this decode phase on mobile devices, which is bound by the device’s limited memory bandwidth rather than by compute power [28, 51]. More specifically, on mobile devices, the LLM’s weights have to be repeatedly loaded from memory to mobile processors by every pass, due to the processors’ limited cache capacities

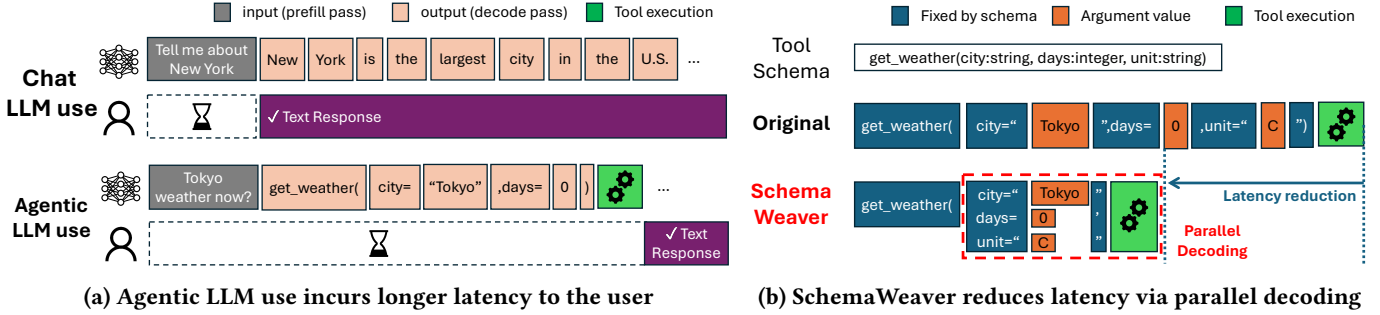


Figure 2: The latency of LLM tool calling in on-device mobile agentic AI

[1, 51]. Since each decode pass only produces one output token, the decode latency is determined by the speed of memory bus, which is usually much slower than the mobile processor speed and hence causes processor starvation. Our experiment results on the OnePlus 12R phone with Snapdragon 8 Gen 2 SoC show that the decode speed on mobile CPU is $> 5\times$ slower than the prefill speed (Sec 2.3). Using mobile GPU or NPU cannot help much, due to the shared memory architecture in mobile SoCs [8, 11, 34, 72].

The key to accelerating the decode phase in mobile agentic AI, hence, is how to allow producing multiple output tokens in one decode pass, to fully utilize the on-device compute power. One straightforward approach is speculative decoding [9, 20, 32], which first uses a much smaller draft model to guess the next few tokens and then lets the target model check the correctness of these guesses in one pass. Since a tool call is highly structured, most output tokens are highly predictable. For example, with a SmoLLM2-1.7B as the target model and SmoLLM2-135M as the draft model, 74% of guesses are correct. However, these correct guesses only result in 31% speedup, because the draft model still decodes one token per pass and is bounded by memory bandwidth as well.

Another option is constrained decoding [16, 21, 33, 64, 82], which builds a set of grammatical rules from the tool’s schema and forbids any output token that makes the LLM tool call ill-formed. These rules can be further used by jump-forward decoding [82] in each decode pass, to skip the current decode computation when only one legitimate output token exists. These skipped tokens can then be processed in batch by a single prefill pass. However, since tokens can only be skipped when they are being computed, the overall decode latency is still bound by the number of decode passes, which can only be sequentially computed.

To fundamentally overcome this limitation of sequential decode computations, we instead aim to allow decoding the argument values of an LLM tool call in parallel, by leveraging the pre-known and fixed information contained in the tool schema. More specifically, the schema describes the position of each argument’s value in the tool call command, and each position is always fixed. For example, in the call `get_weather()` shown in Figure 2(b), the value of argument

city is a string that begins after `city=` and ends at the closing quote, and the value of argument days is an integer after `days=`. Hence, instead of decoding these argument values sequentially, we can decode all of them as one batch in parallel to save unnecessary passes. On the other hand, the fixed names of arguments need no decoding, and can instead be loaded by the LLM inference engine via one prefill pass.

Based on this idea, in this paper we present SchemaWeaver, a new on-device mobile agentic AI technique that enforces schema-guided parallel decoding in the LLM inference engine. First, SchemaWeaver parses the fixed parts in the tool schema to build a *tool skeleton* that provides information about the position of each argument value (Sec 4.1). It then places argument names as parallel token sequences, so that arguments share the same prompt and tool call prefix to facilitate parallel decoding of argument values. To handle rugged value lengths, we pad decode sequences on the fly for the maximum compute efficiency (Sec 4.2). Since decode results of argument values may not share exactly the same sequential prefix, we add a write-back phase after parallel decoding to realign the decode results and rebuild the LLM agent’s KV cache accordingly. To avoid this write-back phase from blocking tool executions, we assemble a valid tool call at the same time and emit the tool call early to further reduce the end-to-end latency (Sec 4.3).

Many tools called by the LLM agent may include optional arguments. For example, when calling the weather query tool, the LLM agent may opt to specify the temperature unit or not. The values of optional arguments have to be decided after all required argument values are generated, and cannot adapt to parallel decoding. Instead, we have to decide and generate values of optional arguments in a sequential manner, after parallel decoding of required argument values (Sec 5). To minimize the extra latency caused by this sequential decoding, we design provisional calls for read-only tools to emit early only with required arguments, which can be later replaced by a corrected call once the LLM agent adds the optional argument values (Sec 5).

To our best knowledge, SchemaWeaver is the first system design that accelerates on-device LLM tool calling on mobile devices, without involving any training efforts or changes

on LLMs. We implemented SchemaWeaver in llama.cpp [22] inference engine by using multiple tokens as a matrix in decoding when performing GEMM (general matrix-matrix multiply)¹, and evaluated it on multiple tool calling benchmarks, mobile SoCs and LLM families including Qwen [74, 75], Llama 3.2 [24] and SmolLM2 [6]. Based on experiment results, we have the following conclusions:

- SchemaWeaver reduces the number of forward passes for a tool call from 44.3 to 13.1, and the end-to-end tool calling latency by 2.13× over the competitive baselines.
- The gain of SchemaWeaver grows as the memory bandwidth falls. When the memory bus is slowed by 4.7×, serial decoding slows by 3.2× but SchemaWeaver only slows by 2.4×, and its speedup rises from 3.0× to 3.9×.
- Applying SchemaWeaver costs <2% of tool calling correctness across different LLM settings.
- SchemaWeaver is lightweight and efficient. With 8.9 MB of extra peak memory being used compared to baselines, it reduces the energy consumption of a tool call by 2.1×.

2 BACKGROUND & MOTIVATION

2.1 Tool Calling Interfaces of LLM Agents

Nowadays, different tool calling interfaces are available and they differ in how a tool is presented to and executed by the LLM agent. Model Context Protocol (MCP) [27, 45] is the most commonly used and provides tool schemas in JSON format with key-value pairs, such as `{"name": "get_weather", "arguments": {"city": "Tokyo"}}`. Other formats also exist, such as the Python-like tool calls in Llama 3.2 [24] and the special tokens around each argument value in FunctionGemma [7]. In each format, the tool schema fixes every character of the tool call except the argument values, and SchemaWeaver hence generally applies to all formats.

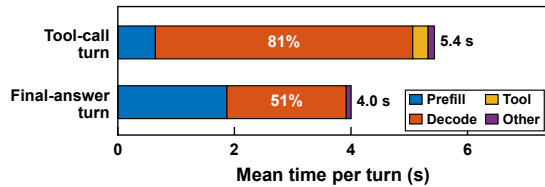


Figure 3: Wall-clock time breakdown of LLM tool calling latency on an OnePlus 12R smartphone

Some desktop coding agents allow the LLM agent to act in free text instead of following the tool schema, by generating a shell command or a program [59, 76]. Such an action is passed through a tool with a single text argument, e.g., `{"command": "ls -l"}`, or executed directly as code. These agents, however, cannot apply to mobile agentic AI, because

¹Our implementation achieves parallel decoding by increasing the data parallelism, and can hence be generically applied to different mobile processors including CPU, GPU and NPU.

a mobile OS often exposes the app functionality through typed and permission-controlled interfaces such as Android AppFunctions [2, 3] and iOS App Intents [4]. Hence, in the rest of this paper, we consider an LLM tool call to always include a tool name with typed argument names and values.

2.2 Tool Call Latency Breakdown

The breakdown of LLM tool calling latency, on a OnePlus 12R smartphone with Qwen3-1.7B-Q8 model, is shown in Figure 3, where the tool-call turn emits the tool call command from the user input (e.g., ① and ② in Figure 1) and the final-answer turn executes the tool and generates response to the user based on results of tool execution (e.g., ③ and ④ in Figure 1). In both cases, decode contributes the most (up to 81%) to the end-to-end tool calling latency. In particular, a large amount of decode computation in the tool-call turn is redundant: our analysis of 2,351 tool calls in the Berkeley Function Calling Leaderboard (BFCL-v4) [50] dataset shows that, for any selected tool, 54-63% of a tool call’s tokens in the decode phase of the tool-call turn are fixed. Hence, SchemaWeaver mainly targets the tool-call turn and reduces its latency by eliminating redundant decode computations.

Mobile Processor	Prefill	Decode
Hexagon NPU	685.4	16.4
Adreno GPU	93.1	11.3
CPU	145.7	22.6

Table 1: Prefill and decode throughputs (tokens/s) on mobile processors with Qwen3-1.7B-Q8 model.

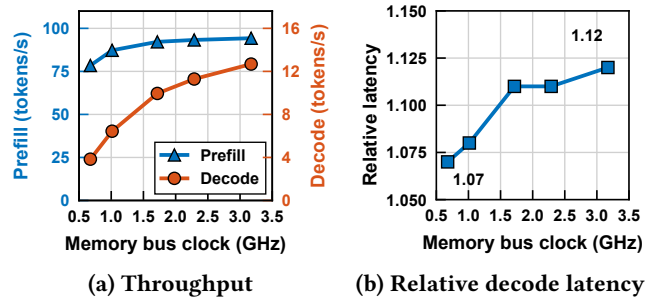


Figure 4: Prefill and decode performance with different memory bus frequencies on mobile devices

2.3 Memory-Bound Tool Call Decoding on Mobile Processors

Table 1 shows the prefill and decode throughput on a OnePlus 12R smartphone with Snapdragon 8 Gen 2 SoC. Among all on-device processors, CPU is the fastest for decoding, and the decode throughput is much slower than the prefill throughput on all mobile processors. These results match the findings in the existing work [11, 72], and are caused

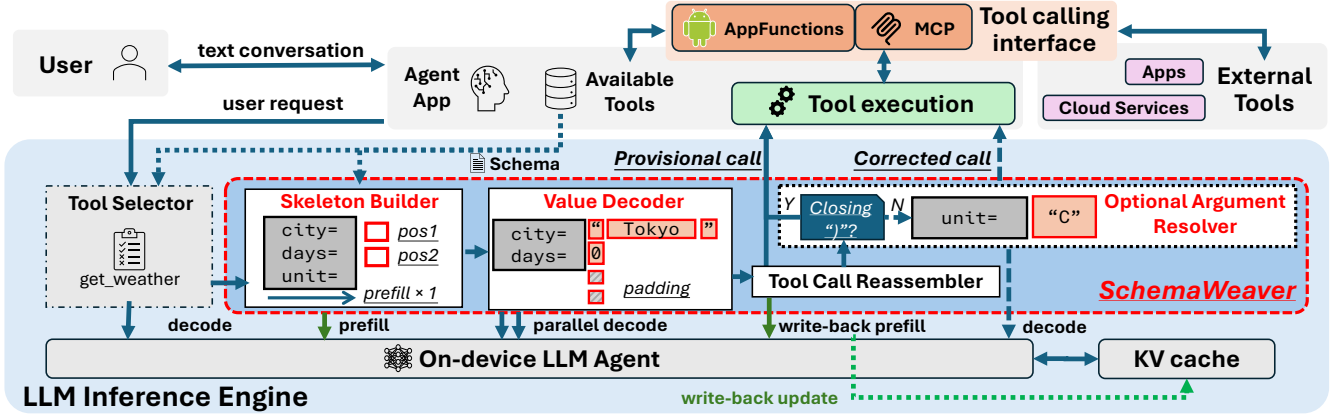


Figure 5: Overview of SchemaWeaver design

by the mismatch between memory bus bandwidth and processor workloads. While prefill produces multiple tokens in one pass, one decode pass only produces one token and results in low utilization of mobile processors awaiting data from memory between passes. Our experiment results in Figure 4(a) further support this observation: when we slow down the memory bus frequency, the decode throughput falls sharply by 70% but prefill throughput falls by only 17%.

Such a bound on memory bandwidth, however, creates unique opportunities for parallel decoding: if a decode pass already reads all the necessary data (LLM model weights, input prompt and tool schema) from memory, the extra latency incurred by decode computations of producing multiple tokens in this pass would be only marginal. As shown in Table 2, when a decode pass outputs 4 tokens² instead of 1, the decode latency only increases by up to 20% and the average per-token decode latency hence drops by 70%. Figure 4(b) further shows that this saving is insensitive to changes of memory bandwidth. These results motivate the parallel decoding design in SchemaWeaver, to fully utilize the available compute power in decode passes by producing multiple tokens in each decode pass.

# tokens in one pass	1	2	3	4	5	6	8
OnePlus 12R	1.00	1.61	2.15	1.20	1.91	2.49	2.26
Google Pixel 7	1.00	1.75	2.34	1.11	1.88	2.47	2.06

Table 2: Relative increase of the parallel decoding cost in one decode pass on different smartphone models

3 OVERVIEW

As shown in Figure 5, SchemaWeaver decodes the argument values of an on-device agentic AI tool call in parallel, and leaves everything fixed in the tool schema as prefill. To that end, we assume that every available tool comes with

²The optimal sizes of token batch are multiples of 4, because LLM inference engine’s CPU matrix routines handle 4 sequences at a time in a pass [22].

a schema that lists its arguments, types of arguments, and whether each argument is required or optional. This availability of tool schemas is guaranteed in most existing agentic frameworks, such as LangChain and AutoGen [31, 65].

SchemaWeaver sits inside the on-device LLM inference engine and does not alter other components outside of tool calling. Without loss of generality, in SchemaWeaver design we assume that the software tool being called is decided by a specialized Tool Selector given the user request, and SchemaWeaver hence focuses on accelerating the corresponding call of the selected tool. In practice, such tool selection has been recognized as an easy NLP task, on which a small LLM (e.g., Octopus-v2-2B [13] or TinyAgent-1.1B [17]) can succeed.

3.1 Tool Skeleton Builder

As stated in Section 1, an argument value can only be decoded in parallel with others if the inference engine knows its position, but traditional serial decoding learns the position of an argument value only when decoding of the previous value has ended. SchemaWeaver, instead, makes the positions of all argument values pre-known before decoding, by building a *tool skeleton* that contains all the fixed parts of tool call, namely the tool name and names of required arguments, from the tool schema. Argument values remain unfilled in the tool skeleton, as exemplified in Figure 5 for the tool `get_weather` with two arguments `city=` and `days=`. Positions of argument values can then be derived by tokenizing the texts in tool skeleton with the argument names laid out in parallel and letting the LLM agent read these tokens in one prefill pass: the first token of each argument value can be located immediately as after the last prefilled token of the corresponding argument name.

Special handling, however, is needed at the boundary between the tool skeleton and an argument value, because the model may write the last character in the skeleton and the value’s first character as one fused token, which distorts the

parallel decoding of argument values. More details about the tool skeleton and such special handling are in Section 4.1.

3.2 Parallel Value Decoder

In parallel decoding of argument values, each argument value corresponds to one token sequence that starts from the token corresponding to the argument’s name in the tool skeleton. All these token sequences can be computed as one batch, and a token sequence stops when the corresponding argument value is completely decoded, depending on the argument’s type. For example, a string ends with the closing quote, and a number ends at the first token that cannot continue as a number. Decoded argument values hence differ in length, and the decoder applies padding to token sequences when necessary. More details are in Section 4.2.

The decode results cannot be directly used as input to the LLM agent, because each sequence contains only part of the tool call command (i.e., the corresponding argument and its value). The LLM agent cannot continue its inference on any sequence, as it cannot read the full text of tool call command. To address this issue, SchemaWeaver writes the decoded argument values back to a sequential space in device memory after the parallel decoding completes, to reassemble the complete call text and enter it into the LLM’s KV cache as one new sequence. The write-back costs a single prefill pass instead of decoding the call again token by token, since the assembled call can be handed to the tool for execution before write-back, such reassembling does not incur extra latency on tool execution. More details about such write-back are in Section 4.3.

3.3 Optional Argument Resolver

Optional arguments involved in the specific tool call need to be separately handled. Only after the write-back of required argument values, SchemaWeaver decides the optional arguments one at a time with the Optional Argument Resolver, to ensure the correctness of generated tool call command. To minimize the latency of such serial decisions, SchemaWeaver starts the tool early as a *provisional call* that carries only the required arguments, while the Optional Argument Resolver decides the optional arguments. If the tool call indeed involves optional arguments, these are later added towards a corrected call that replaces the provisional call.

Early execution of the provisional call is feasible only when the tool is read-only and its execution does not apply any changes to the mobile system. For example, a provisional call to the weather lookup tool illustrated in Figure 2 is feasible, but other tools such as sending a message cannot be executed by a provisional call. More details are in Section 5.

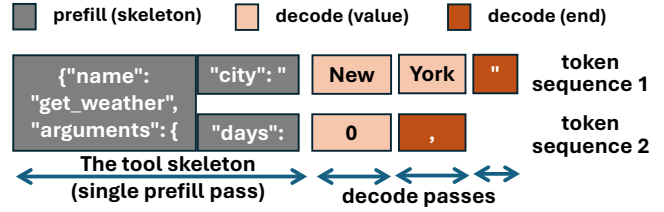


Figure 6: The tool skeleton in one prefill pass, with every argument value starting at the same position

4 PARALLEL DECODING OF ARGUMENT VALUES

In this section, we will use an end-to-end example of the `get_weather` tool call with two required arguments, `city` and `days`, to present details about parallel decoding of the tool call’s argument values.

4.1 Building the Tool Call Skeleton

The tool skeleton is determined jointly by the tool schema and the LLM’s tool call format being used, which fixes the tokens that open and close a call and the keys around the tool name and list of arguments. Different LLM families use different tool calling formats, and SchemaWeaver’s Tool Skeleton Builder adapts to them, by recognizing the format being used from the special tokens in the model’s vocabulary. For example, for Qwen3 models [74], the shared head of tool skeleton uses the opening `<tool_call>` tag and the key arguments. The closing brace and the closing tag come after the last argument value, and so they are added at the write-back (Sec 4.3). For Llama 3.2 [24] models, the head has no tag and uses the key parameters.

Figure 6 shows how the tool skeleton is laid out. The shared head of the call lists the tool name and the beginning part of the argument list. They are shared and visible to all token sequences in decoding, each of which corresponds to one argument and starts at the same position after the shared head. A single prefill pass processes the head and all argument names together; afterwards, the starting position of every argument value (in decoding) is known, as decoding follows from the skeleton’s next token in the sequence.

Model	Serial decode (ms)	Jump-forward prefills (ms)	SchemaWeaver’s prefill (ms)
Qwen3-1.7B	1,201	577	314
Qwen3-1.7B 4-bit	999	615	335
Qwen3-0.6B	545	299	172
Llama-3.2-1B	743	407	222
Llama-3.2-3B	1,886	997	551
SmolLM2-1.7B	1,352	945	395

Table 3: Compute cost of tool skeleton decode per call

Implementing the skeleton decode as prefill is beneficial. Table 3 shows the compute cost of building the skeleton

on 100 tool calls picked from the BFCL dataset [50] using a OnePlus 12R smartphone, where SchemaWeaver achieves up to 3.8× speedup compared to other approaches.

In addition, most tool call formats insert a space in front of argument values, and the LLM tokenizers [54] fuse the prefix space and the first character of the argument value, such as the minus sign (-), into the same token. Placing the space into the tool skeleton hence confuses the model into emitting wrong values. SchemaWeaver builds the skeleton with the LLM tokenizer as the reference, to avoid splitting a fused token between the skeleton and argument values.

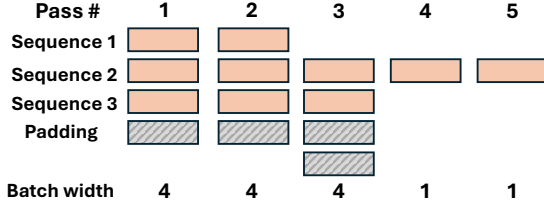


Figure 7: Parallel decoding of three argument values with rugged lengths being padded

4.2 Parallel Token Sequences

As shown in Figure 6, each token sequence is separated from others, and restricted to the type of its argument and ends where its type ends: at the closing quote of a string, at the separator after a number or a boolean, or at the matching bracket of an array or an object.

Decoding pass reduction. As shown in Figure 7, the decoding for all token sequences advances together. For a call with k required argument values $v_1, \dots, v_k$ of $|v_i|$ tokens each, serial decoding spends $\sum_i |v_i|$ decode passes, whereas SchemaWeaver takes $s_{\text{name}} + \max_i |v_i|$ decode passes plus two prefill passes, where s_{name} corresponds to the one or two passes that decide the tool name, and the two prefill passes are for the tool skeleton and the write-back (Sec 4.3). When being tested on the BFCL dataset [50], SchemaWeaver reduces the number of decode passes from 13.1 to 6.8 on average, and reduces the decoding latency by 39% accordingly. Especially for complicated tool calls with 2 or more required arguments, it takes advantage of argument parallelism and further reaches 47% latency reduction.

Padding rugged value lengths. A pass with n unfinished sequences computes n tokens in one batch. However, decoding with the minimum number of sequences (i.e., only the unfinished sequences) is not always proper on mobile devices. As shown in Table 2, a batch of 4 minimizes the per-sequence decode cost as a result of mobile LLM inference engine (e.g., llama.cpp), and the Parallel Value Decoder in SchemaWeaver hence pads a pass with two or three unfinished sequences using placeholder tokens. Our statistics show that, over tool calls in the BFCL dataset [50], such padding is applied in 27%

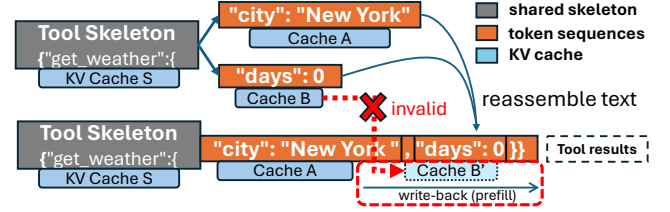


Figure 8: Parallel decoding invalidates the KV cache entries of token sequences. The write-back reassembles the tool text and rebuilds memory of the whole call. of decode passes and further improves the decode speedup of SchemaWeaver by another 17%.

Limited impact on correctness. Parallel decoding may impact the LLM agent’s correctness of generating the tool call command, because of the mutual separation between parallel token sequences. For example in a flight search task from Berlin to New York, serial decoding writes the origin as the airport code BER and then the destination as JFK, but parallel decoding may not see the origin and incorrectly uses NYC. Nevertheless, the impact of such incomplete tool call information on the correctness is minor, because most argument values are taken directly from the user’s request and rarely depend on each other. On the BFCL dataset [50] with Qwen3-1.7B, this impact is as low as 0.9%, and more detailed results over different LLM families are in Section 7.

4.3 Assembling the Tool Call

When decoding of all token sequences has finished, the required argument values are all known. However, since different argument values are simultaneously generated in parallel sequences, such parallel decoding invalidates the token sequences’ corresponding entries in the LLM agent’s KV cache, when these entries are integrated into a single sequence to complete the tool call³. For example in Figure 8, one token sequence ends with "city": "New York" and the other ends with "days": 0, but neither of them completes the tool call.

SchemaWeaver tackles this issue with the Tool Call Reassembler, which discards KV caches in diverged token sequences, reassembles the decoding results from parallel token sequences into a single sequence, and recomputes the affected KV cache for the complete tool call. The assembled argument texts are then written back to the KV cache in a single prefill pass. For tool calls in the BFCL dataset [50], the write-back prefills 34.5 tokens on average and takes 522 ms on the OnePlus 12R, about the time of 9 decode passes.

However, note that this write-back may not increase the end-to-end tool call latency, because the recomputed KV cache is only needed in the next conversation of LLM agent (e.g., when the LLM agent reads the tool’s result and writes its reply to the user). Hence, if the tool call has no optional

³Since KV cache is a sequential data structure [30, 51], altering certain tokens would invalidate all cache entries at later positions in the KV cache.

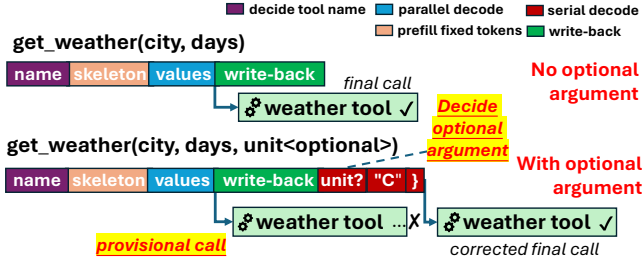


Figure 9: Handling optional arguments with a provisional call that only contains the required arguments

argument, the assembled tool call texts can be handed to the tool for execution before write-back. Calling a tool whose schema contains optional arguments, on the other hand, will be discussed in Section 5.

5 HANDLING OPTIONAL ARGUMENTS

Many tool calls also contain optional arguments (e.g., 66% in the BFCL dataset [50]). Figure 9 illustrates the two cases that SchemaWeaver handles, namely tools without and with optional arguments. We first explain when the optional arguments should be decided, and then how provisional calls are designed to minimize the tool call latency with optional arguments.

Deciding optional arguments. For a specific tool call, while its required arguments are pre-defined by tool schema and can be decoded in parallel based on such information, the optional arguments being involved depend on values of required arguments and cannot be decided in advance.

Table 4 shows why the time of deciding the involved optional arguments matters. If the optional arguments are decided before any value of required arguments is generated, the correctness of deciding the optional arguments is 17% lower than that with serial decoding, as the LLM agent has to make guesses in many cases. The correctness of the tool call, correspondingly, also drops by 13.5%. Another choice is to include all the available optional arguments into the tool skeleton but usually involves unused arguments, further reducing the argument correctness by another 5.2%.

Hence, SchemaWeaver opts to decide the optional arguments after values of required arguments are decoded, using serial decoding, and such late decision of optional arguments improves the argument correctness to 79%. On the BFCL dataset [50], such serial decoding adds 13.5 decode passes on average.

Provisional calls. To tackle such extra latency caused by serial decoding of optional argument values, our basic approach is to let the tool start on a provisional call that carries only the required arguments, and the optional arguments

When the optional arguments are decided	Argument correctness	Tool call correctness	Latency (ms)
End of the call (serial decoding)	80%	76.0%	2,586
Before required argument values (in the skeleton)	63%	62.5%	1,105
Never (all put into the skeleton)	57%	70.8%	1,103
After required argument values (SchemaWeaver)	79%	74.0%	1,211

Table 4: Different choices of times when the optional arguments are decided, with the BFCL dataset [50] and Qwen3-1.7B model

are decided and decoded during this provisional tool’s execution. As shown in Figure 9, the write-back described in Section 4.3 comes after the provisional call and before the optional argument decision. In this way, both the write-back and the decision of optional arguments can be conducted while the tool executes, hence avoiding the tool execution being delayed.

Afterwards, if no optional argument is involved, the provisional call is considered as correct and its execution results are used to generate the response to the user. Otherwise, the provisional call’s execution results will be replaced by the corrected call with the optional arguments added, and the provisional call and its execution results are discarded.

Read-only tools. Since a provisional call may be discarded, it only applies to tools whose execution can be repeated without effect. For example, the tool annotations of the MCP specification mark such a tool as *read-only*, defined as a tool that “does not modify its environment” [45]. SchemaWeaver sends provisional calls only to these read-only tools. In other cases, a tool without read-only annotations (e.g., non-MCP tools) is classified by its name and description. For example, a tool that modifies the local system’s configurations is not read-only and cannot be started with a provisional call.

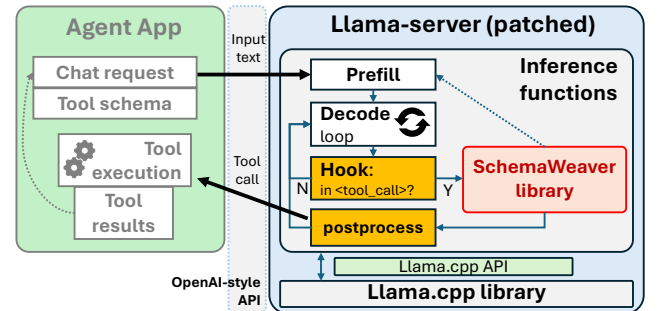


Figure 10: Implementation of SchemaWeaver inside the llama-server

	Qwen3-1.7B		Qwen3-1.7B 4-bit		Qwen3-0.6B		Qwen3-4B		Llama-3.2-1B		Llama-3.2-3B		SmolLM2-1.7B	
Latency (ms)	12R	Pixel 7	12R	Pixel 7	12R	Pixel 7	12R	Pixel 7	12R	Pixel 7	12R	Pixel 7	12R	Pixel 7
No Grammar	2,602	3,565	2,242	2,482	1,271	1,569	5,605	24,004	1,854	2,707	4,137	6,145	3,749	5,664
Constrained decoding	2,586	3,561	2,173	2,456	1,183	1,569	5,574	23,871	1,636	2,415	4,009	5,852	3,541	5,619
Jump-forward decoding	1,921	2,723	1,807	1,794	1,071	1,346	3,867	18,563	1,304	1,842	2,719	4,365	2,396	3,536
Speculative, EAGLE-3	9,610	12,031	8,141	9,102	3,709	4,812	14,714	OOM	8,703	12,847	14,585	21,290	/	/
Speculative, DFlash-2	/	/	/	/	/	/	13,433	OOM	/	/	/	/	/	/
SchemaWeaver	1,211	1,600	1,236	1,272	577	705	2,685	10,334	773	1,101	1,840	2,707	1,498	2,243
# of Forward passes														
No Grammar	44.2		43.4		42.8		45.0		45.5		40.8		60.1	
Constrained decoding	44.3		41.6		42.4		44.7		42.9		40.0		57.8	
Jump-forward decoding	21.9		19.2		19.7		22.4		20.1		18.2		21.0	
Speculative, EAGLE-3	18.1		17.2		13.9		16.9		17.4		16.9		/	
Speculative, DFlash-2	/		/		/		9.2		/		/		/	
SchemaWeaver	13.1		13.2		12.3		12.7		12.6		11.6		14.5	
Correctness														
No Grammar	76.0%		74.2%		62.9%		78.1%		8.0%		29.0%		20.0%	
Constrained decoding	76.0%		73.2%		65.2%		78.1%		60.6%		78.0%		65.3%	
Jump-forward decoding	76.0%		73.2%		65.2%		79.2%		61.6%		78.0%		65.3%	
Speculative, EAGLE-3	76.0%		73.2%		65.2%		78.1%		60.6%		78.0%		/	
Speculative, DFlash-2	/		/		/		78.1%		/		/		/	
SchemaWeaver	74.0%		72.1%		63.6%		81.2%		58.0%		76.0%		64.0%	

/: no draft model or head exists for the model. OOM: out of memory on the phone.

Table 5: The end-to-end latency and correctness of tool calls over different LLMs. BFCL-v4 dataset is used.

6 IMPLEMENTATION

We implemented SchemaWeaver on llama.cpp [22] (commit d59d455f, August 2026), the most commonly used open-source LLM inference engine for Android smartphones. Our implementation consists of a library of 930 lines of C++ code on the public API of llama.cpp, together with a 350-line patch to its server as a hook on tool calls to its decode loop. When the LLM agent chooses to emit a tool call, the hook lets SchemaWeaver produce the call, and returns the provisional call to the agent app as soon as it is assembled, before the optional arguments are decided. When the call is complete, the hook feeds its tokens back to the server exactly as if they had been decoded in serial. The engine itself is otherwise unchanged. Figure 10 shows how the pieces fit together.

SchemaWeaver is deployed on two smartphone models with different mobile SoCs, i.e., a OnePlus 12R with Qualcomm Snapdragon 8 Gen 2 SoC and 16 GB of memory running Android 14, and a Google Pixel 7 with Google Tensor G2 SoC and 8 GB of memory running Android 16. The LLM inference engine and SchemaWeaver are built from source with clang 21.1 in Termux, and we use the publicly released GGUF files of LLMs in 8-bit (Q8_0) and 4-bit (Q4_0) quantization. Both smartphones are controlled from a desktop over the Android Debug Bridge (adb), which launches SchemaWeaver and the baselines to collect experiment data.

7 EVALUATION

In our evaluation, we use open-sourced LLMs including Qwen3 [74], Llama-3.2 [24] and SmolLM2 [6]. The model size varies from 0.6B to 4B to fit the memory and compute

limits of smartphones. We use the following tool call datasets to evaluate the on-device performance of tool calling:

- **BFCL-v4** [50] consists of 2,501 single-turn tool call samples across 991 tools, with up to 9 arguments per call and 2.4 on average. Our evaluation uses a subset of 100 calls, uniformly sampled over the number of arguments per call.
- **DroidCall** [69] is a dataset for Android system intents, i.e., tool calls that open apps, set alarms or send messages on the phone, from which we take its 154 single-call requests over 23 tools.

Experiment protocol. To ensure fair comparisons, the inference engine process is pinned to 4 performance cores of mobile CPU, at a fixed CPU clock under a cooling fan, since thermal throttling would otherwise change CPU frequency in the middle of a run. The choice of the core set and CPU clock is given in Appendix A. Peak resident memory is read from the mobile processor’s VmHWM counter polled once per second. Time consumption is measured via the phone’s builtin monotonic clock.

Baselines. We compare the performance of SchemaWeaver with the following four decoding schemes, and every baseline runs in the same engine software build as SchemaWeaver.

- **Unconstrained decoding**, shown as *No Grammar* in tables and figures, lets LLM write the call freely.
- **Constrained decoding** [16, 21, 22, 64] applies the schema constraint built into llama.cpp on top of unconstrained decoding.

- **Jump-forward decoding** [82] skips the tokens that the schema fixes when the decoder meets them, as described in Section 4.1.
- **Speculative decoding** [32] guesses the next tokens using a smaller draft model for the target model to check its correctness. Due to the limitation in available draft and target models in LLM families we use, we test two different schemes with different models: 1) EAGLE-3 [36] with a draft layer of 59M to 247M parameters that reads the target model’s hidden states, available for every model in our set except SmolLM2; 2) DFlash-2 [10] with Qwen3-4B as target model and a 1.0 GB block-diffusion draft model.

Metrics. Our primary evaluation metric is the *tool call latency*. For SchemaWeaver, we compute the latency as the elapsed time between the start of generating tool call texts and the start of tool execution. For the baselines, the latency ends at the last token of the tool call. Latency is reported as the geometric mean [18, 26] over the repetitions of each call and then over the calls, since it is the mean that summarizes ratios consistently. We report a speedup as the ratio of two such means.

Besides, we also evaluate the whole-call *correctness*, i.e., the percentage of the tool calls made whose tool and arguments match the accepted answers in the dataset. The number of forward passes per call, the peak resident memory and energy cost per call are also evaluated and reported as more detailed information about the compute costs of tool calls.

7.1 Latency and Correctness of Tool Calling

We first compare SchemaWeaver with the four baselines on the BFCL dataset. As shown in Table 5, across the seven model settings and on both phones, SchemaWeaver keeps the correctness of tool calls at the same level as constrained decoding: on Qwen3-1.7B, the whole-call correctness is 76.0% under constrained decoding and 74.0% with SchemaWeaver. We find that SchemaWeaver chooses the same arguments as constrained decoding does, and the correctness loss comes from parallel decoding, where the values cannot see each other. Jump-forward decoding produces the same tokens as constrained decoding and keeps its correctness. The same applies to the speculative rows: on the Qwen3 models, decoding with No Grammar stays within about 2% difference from that of constrained decoding, so the constraint costs no correctness. The Llama and SmolLM2 models need the grammar for the call format itself. Without it they easily write the call in the wrong format, and their correctness falls to 8–29%.

Given such correctness of tool calling, SchemaWeaver significantly reduces the end-to-end latency of tool calling. On Qwen3-1.7B, its end-to-end latency is 1,211 ms on the

	Qwen3-1.7B		Llama-3.2-1B		SmolLM2-1.7B	
Latency (ms)	12R	Pixel 7	12R	Pixel 7	12R	Pixel 7
Constrained	1,771	2,523	1,035	1,567	2,682	4,136
Jump-forward	804	1,080	461	693	1,115	1,630
SchemaWeaver	677	944	328	490	779	1,166
Correctness						
Constrained	67.2%		58.4%		69.3%	
Jump-forward	67.2%		58.4%		68.6%	
SchemaWeaver	67.2%		61.7%		67.4%	

Table 6: The end-to-end latency and correctness of tool calls over different LLMs. DroidCall dataset is used.

OnePlus 12R smartphone, a 2.13× speedup over constrained decoding. The number of forward passes is correspondingly reduced from 44.3 to 13.1. Jump-forward decoding reaches 1.82×, since it skips the fixed tokens but still decodes every argument value one token per pass. Decoding with no grammar is as slow as constrained decoding, so the grammar constraint is not the bottleneck. SchemaWeaver reaches a higher speedup of 2.36× on SmolLM2-1.7B, since its call format consists of more fixed tokens.

The speculative rows gain little speedup on both phone models and all LLM families. EAGLE-3 for Qwen3-1.7B and DFlash-2 for Qwen3-4B run slower than constrained decoding at 0.27× and 0.42×, respectively. Two costs on the phone turn the gain into a loss. The verification pass in speculative decoding checks the drafted tokens as one batched pass, which on the phone costs more than a serial step according to Table 2. With 4.5 of the 16 drafted tokens accepted per block, the verification pass of DFlash-2 alone costs more than the serial steps it replaces. The draft head also reads its own weights from the same DRAM at every drafted token, so the drafting competes for the bandwidth that limits the decode phase. Besides, the EAGLE-3 head must process the whole prompt during prefill, from 11.4 to 50.8 ms per prompt token, and that accounts for three quarters of its loss.

Table 6 further shows the experiment results on the DroidCall dataset, whose tools are Android system functions with 1.5 arguments per call on average. SchemaWeaver reduces the tool call latency from 1,771 ms of constrained decoding to 677 ms on the OnePlus 12R with Qwen3-1.7B, reaching a speedup of 2.62×, and holds between 2.67× and 3.55× in the other settings, with correctness loss against constrained decoding controlled within 2%. The gain is larger than that on the BFCL dataset although the calls are shorter, because most of these calls have one required argument and five of the 23 tools are read-only, and so the provisional call handles the tool early as soon as the required value is decoded.

7.2 Different Model Sizes and Quantizations

Table 5 lists the measured latencies across six models, one of them (Qwen3-1.7B) in two quantizations. The speedup over constrained decoding holds between 1.76× and 2.36×

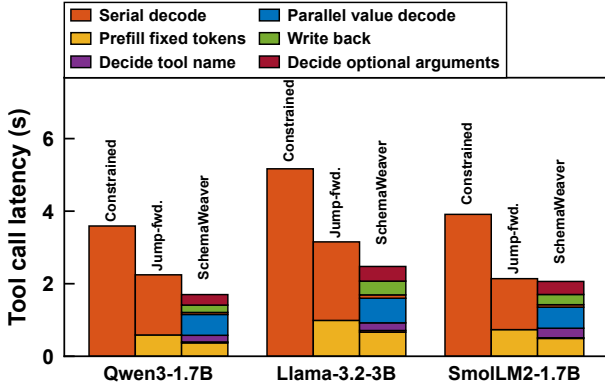


Figure 11: Tool call latency by phase on the OnePlus 12R

on the OnePlus 12R and between $1.93\times$ and $2.50\times$ on the Pixel 7. To see where the time of a call goes under each setup, Figure 11 splits the latency of the three schemes by phase on one model of each family. On Qwen3-1.7B and calls with 2+ required values, the tool name takes 170 ms, the skeleton pass 351 ms and the parallel values 572 ms on average, while the write-back (522 ms) and the optional arguments (247 ms) fall after the provisional call and overlap the tool execution for read-only tools. Llama 3.2 and SmoLLM2 spend more fixed tokens per call due to their call formats, so their skeletons replace more serial decoding and their speedups in Table 5 are larger.

The skeleton prefill costs about the same (5 to 6 serial decode steps) in every setting, so the differences in speedup come from the call format and the quantization. The 4-bit Qwen3-1.7B gains the least. Its weights are half the size, but its decode step is only 13% faster, because unpacking 4-bit weights costs extra arithmetic. Qwen3-4B does not fit in the Pixel 7’s 8 GB of memory. The engine keeps up to 1.5 GB of the 4.3 GB model in compressed swap in RAM, so a call under constrained decoding takes significantly longer ($23.9\times$) due to swapping. Even so, SchemaWeaver still achieves a speedup of $2.31\times$.

On the full BFCL dataset, SchemaWeaver loses 0.7 to 2.0 points of whole-call correctness across the seven settings, with the largest loss on Llama-3.2-1B. The loss comes from the required values decoded in parallel, which cannot see each other. On the 100-call subset in Table 5, the differences vary more ($+3.1$ to -5.6 points) since the subset is small, but they remain within a few points.

7.3 Impact of Memory Bandwidth

Section 2.3 shows that the decode phase is bound by memory bandwidth, so the gain of SchemaWeaver should grow on phones with a slower memory bus. To verify this, we pin the memory bus of the Pixel 7 at four of its own frequency levels, from the stock maximum of 3,172 MHz down to 676 MHz,

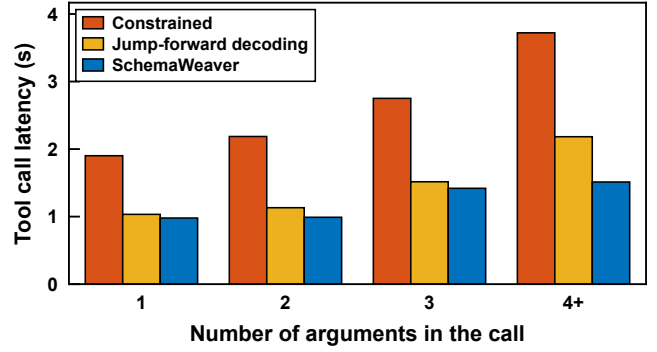


Figure 12: Tool call latency by the number of arguments of the call on the OnePlus 12R.

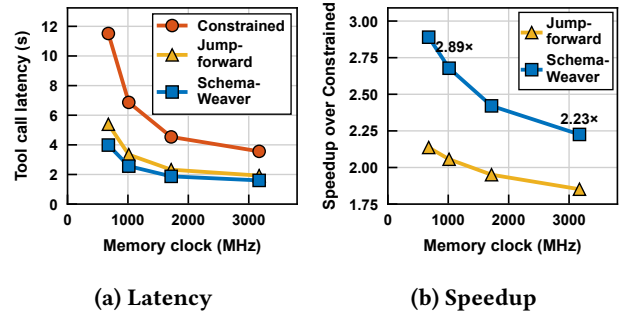


Figure 13: Tool call latency and speedup with different memory bus frequencies on the Pixel 7.

with the CPU clock pinned and the fan on. We repeat the BFCL calls of Table 5 with Qwen3-1.7B at every level. Figure 13 plots the latency of every scheme and the speedup of SchemaWeaver against the memory clock, i.e., the frequency of the memory bus.

The gain of SchemaWeaver indeed grows as memory bandwidth falls. As the bus slows $4.7\times$, constrained decoding slows $3.23\times$ while SchemaWeaver slows $2.49\times$, so its speedup rises from $2.23\times$ at the stock clock to $2.89\times$ at 676 MHz. Jump-forward decoding also gains from the slower bus, since the tokens it skips would have cost decode steps as well, but it gains less ($1.85\times$ to $2.14\times$), so the gap between its speedup and that of SchemaWeaver grows from 0.38 to 0.75.

The per-phase records explain this difference. With the lowered bus frequency, a serial decode step slows $3.23\times$, from 90 to 291 ms, because every step reads the whole model from memory. The four-sequence value pass slows almost as much, from 107 to 332 ms, since it reads the weights once as well. The prefill passes behave differently: the skeleton prefill slows only $1.62\times$, from 417 to 674 ms, and a prompt token only $1.12\times$, because a prefill reads the weights once for many tokens and stays compute-bound. SchemaWeaver replaces most of the serial steps of a call with prefill work and a few batched passes, so a smaller part of its latency scales with the bus.

Setup	Latency (ms)	Correctness
Constrained decoding	2,586	76.0%
+ skip the fixed tokens (jump-forward)	1,421	76.0%
jump-forward with a provisional call	1,349	76.0%
+ skeleton and provisional call	1,369	75.0%
+ parallel value sequences	1,247	74.0%
+ padded value batch (= SchemaWeaver)	1,211	74.0%
Argument set given (upper bound)	1,036	77.1%

Table 7: Latency and correctness on SchemaWeaver design variants

7.4 Impact of the Number of Arguments

The gain of SchemaWeaver grows with the number of arguments of the call, since every argument adds to the decode parallelism. Figure 12 shows the latency by the number of arguments. A one-argument call has nothing to decode in parallel, and its $1.9\times$ speedup comes from the skipped fixed tokens and the provisional call alone, so the gap to jump-forward decoding is the smallest there. Calls with four or more arguments gain $2.5\times$, since the parallel pass batches the most values there, and the gap to jump-forward decoding is the largest. The three-argument calls gain less than the two-argument ones ($1.9\times$ against $2.2\times$), because the model adds an optional argument after the provisional call in 72% of them, against about half of the other calls, so their tool call is more often the corrected call that waits for the write-back and the serial decision.

7.5 Ablation Studies

Table 7 adds the design steps of SchemaWeaver one at a time on the OnePlus 12R with Qwen3-1.7B. The first four rows separate the gain of skipping the fixed tokens from the gain of the parallel values. Skipping the fixed tokens, by jump-forward decoding or by the tool skeleton, together with the provisional call reaches a speedup of up to $1.92\times$. Parallel decoding of the argument values and the padded batch then raise the speedup to $2.13\times$. The skeleton and the parallel values cost one point of whole-call correctness each, and the padding costs none. The last row is an upper bound. When the set of arguments is given in advance, the call starts 15% sooner and is 3.1 points more correct than with SchemaWeaver, so deciding the optional arguments late recovers most of that correctness.

Table 8 breaks down the latency of one of the 100 BFCL calls by phase. The user asks for the traffic from Boston to New York, and the tool `get_traffic_info` has 2 required arguments and 1 optional argument. Constrained decoding spends 35 decode passes on the call. SchemaWeaver decides the name in one pass, prefills the skeleton in one pass, decodes the two values in two passes, and emits the provisional call to the tool after 505 ms, $3.97\times$ sooner than constrained

Phase	Passes	Tokens	Time (ms)
Constrained decoding	35	35	2,004
Tool selector	1	6	109
Skeleton prefill	1	19	268
Parallel decode	2	3	128
Provisional tool call			505
Write-back	1	18	238
Optional argument	4	7	238
Final tool call			981

Table 8: Cost breakdown for `get_traffic_info` tool, on the OnePlus 12R with Qwen3-1.7B.

decoding. The write-back and the optional argument, decided as Section 5 describes, complete the final call after 981 ms, with 9 passes instead of 35 and the same call text as constrained decoding produces.

7.6 Memory Overhead

SchemaWeaver is a memory-efficient approach, especially compared with designs based on speculative decoding. Table 9 lists the peak memory of each scheme. The peak resident memory for SchemaWeaver on the OnePlus 12R is only 8.9 MB above constrained decoding on Qwen3-1.7B. This extra memory holds the KV cache entries of the parallel sequences and their type constraints. Decoding with no grammar and jump-forward decoding add less than 1 MB, since neither keeps any state beyond the serial decoder’s. The EAGLE-3 head adds about 2.2 GB on top of its own constrained-decoding run, for its weights and for the prompt it must process alongside the target model.

	Constrained decoding	No Grammar	Jump-forward	Speculative, EAGLE-3	SchemaWeaver
Peak memory	4.725 GB	4.725 GB	4.725 GB	6.926 GB	4.734 GB

Table 9: Peak resident memory of each scheme on the OnePlus 12R with Qwen3-1.7B.

7.7 Energy Consumption

We further evaluate the energy consumption of SchemaWeaver per tool call. Figure 14 shows the energy cost of a tool call on the Pixel 7, split into the CPU rails and the memory rails. SchemaWeaver reduces the energy of a tool call from 12.8 to 6.1 J, i.e., by $2.10\times$. Jump-forward decoding saves almost as much ($1.97\times$), but the savings come from different rails. Jump-forward decoding saves memory energy only on the tokens it skips ($2.3\times$). SchemaWeaver saves $2.8\times$ on the memory rails, since one parallel pass reads the weights once for up to four values, but spends part of that saving on the CPU rails for the write-back, the optional arguments and the padding. The energy of a call counts the write-back and the optional arguments even when they overlap the tool

execution. For nearly the same energy, SchemaWeaver starts the tool call $2.23\times$ sooner than constrained decoding, against $1.85\times$ for jump-forward decoding.

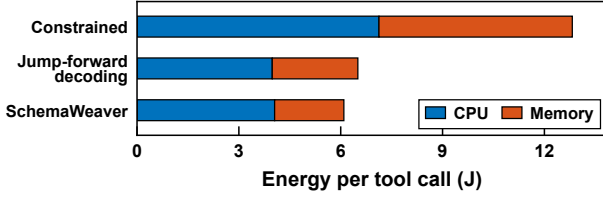


Figure 14: Energy per tool call on the Google Pixel 7

8 RELATED WORK

On-device LLM agents. Many mobile agents operate existing applications through the GUI [57, 62, 63, 79]. Mobile platforms are recently adding native, schema-described tools, such as Android AppFunctions [2, 3], iOS App Intents [4] and MCP [27]. Agent-X [15] accelerates the on-device agent pipeline by reusing the unchanged parts of the prompt across turns, while SchemaWeaver shortens the generation of the tool call within each turn, so the two can be used together. SPORK and PASTE [5, 56] guess the tool call while the model is still generating it, start the tool early, and discard the result if the guess was wrong, which partially inspires the optional argument handling in SchemaWeaver.

Parallel text generation. Another line of work generates several segments of text in parallel: Skeleton-of-Thought [46], APAR and PASTA [29, 41] target free-form texts only. SimpleTool focuses on tool calls and decodes a function’s name and arguments in parallel from the same independence observation [55]. However, it trains special tokens into the model and evaluates on a desktop GPU. HyFunc also shortens function calls, with a trained cascade that hands the call from a large model to a small one and fills the fixed syntax from a template [38]. SchemaWeaver is training-free, runs off-the-shelf quantized models on phones, and grounds its design in their memory-bandwidth profile.

Speculative decoding. Speculative decoding drafts several tokens with a cheaper mechanism and verifies them in one forward pass of the target model; the draft comes from a smaller model, from small heads trained onto the target model, or from the model’s own recent output [9, 20, 32, 35, 36]. Newer drafts come from the workload itself: SuffixDecoding and AgentSpec reuse the repetitive outputs of agent pipelines on a serving cluster [48, 60], and ToolSpec drafts a tool call from its schema and from retrieved past calls of the same tools [66]. These gains rely on a history of similar calls, which the first tool call on a phone does not have, and a wrong draft still costs the batched pass that rejects it. On phones, speculative systems keep the draft model in memory and load the target from flash only to verify a batch of drafted tokens as in LLMcad, EdgeLLM and

Lever [58, 70, 71], or schedule the draft and the verification on different processors [14, 43]. Self-speculative decoding drafts with the target model itself by skipping some of its layers, as in Draft & Verify [81] and SWIFT [67], so it needs no second model. However, they are designed for larger models (e.g., 13B) that cannot fit onto a phone. In every case, the draft model is a second set of weights that the memory bus must also move. SchemaWeaver needs no second model, and the passes it saves are fixed by the schema rather than by how well a draft guesses. The two can be combined: a draft model shortens each value sequence while SchemaWeaver decodes the sequences together.

9 DISCUSSIONS

Mobile accelerators. SchemaWeaver runs on the mobile CPU. The schedule change itself is not CPU-specific, since every pass it adds or removes is an ordinary batched forward pass, but it brings no gain on the phone’s other processors today. On the GPU, a pass that advances four value sequences costs 2.5 serial decode steps against 1.2 on the CPU, a serial step is $1.7\times$ slower than the CPU’s at the same clock, and running the whole design there makes the call slower than constrained decoding ($0.39\times$). The contention with the GUI and the longer initialization time on GPU are also problematic. On the NPU, a four-sequence pass costs about three serial steps, and a serial step takes about $3\times$ longer than on the CPU, since the inference engine keeps the model’s output layer on the CPU for now and every decode step then pays a round trip between the two processors. What the accelerators do offer is a cheap prefilled token, $10\times$ cheaper than a decode step on the GPU and over $70\times$ on the NPU [11, 72]. This matches the skeleton and write-back prefills of SchemaWeaver; letting the NPU run those prefills while the CPU decodes the value sequences is our future work, if moving the state between the two processors costs less than it saves. MNN-AECS [28] runs the bandwidth-bound decode on fewer and slower CPU cores to save energy; SchemaWeaver shortens that same phase, so the two designs can be combined.

Other inference engines. SchemaWeaver needs two things from an engine: a schema-derived grammar and attention masks supplied by the caller in the batched decode pass. Only llama.cpp offers both today, while MNN-LLM [61], ExecuTorch, Google AI Edge SDK [23] and MLC-LLM [44] with TVM [12] miss necessary features on Android. Since the technique behind SchemaWeaver is not engine-specific, we expect similar speedup on other inference frameworks.

10 CONCLUSION

In this paper, we present SchemaWeaver, a new technique to accelerate the use of LLM tools on mobile devices by allowing the parallel generation of argument values for tool calls. SchemaWeaver achieves up to $3.9\times$ speedup in tool call latency without noticeable correctness loss compared to the

existing schemes, and also reduces energy consumption by $2.1\times$.

REFERENCES

- [1] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatami-fard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 12562–12584. doi:10.18653/v1/2024.acl-long.678
- [2] Android Developers. 2026. Overview of AppFunctions. <https://developer.android.com/ai/appfunctions>. Accessed 2026-08-20.
- [3] Android Developers Blog. 2026. Build intelligent Android apps: Integrate into Android’s intelligence system using AppFunctions. <https://android-developers.googleblog.com/2026/07/build-intelligent-android-apps-appfunctions.html>. Accessed 2026-08-20.
- [4] Apple Developer Documentation. 2024. App Intents. <https://developer.apple.com/documentation/appintents>. Accessed 2026-08-20.
- [5] Huajun Bai, Weiwei Lv, Huichuan Zheng, Youyou Lu, and Jiwu Shu. 2026. SPORK: Self-speculative forking to accelerate agentic LLM inference. arXiv:2607.03333
- [6] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, et al. 2025. SmolLM2: When smol goes big – data-centric training of a small language model. arXiv:2502.02737
- [7] Kat Black and Ravin Kumar. 2025. FunctionGemma: Bringing bespoke function calling to the edge. <https://blog.google/technology/developers/functiongemma/>. Accessed 2026-08-30.
- [8] Guanyu Cai, Ruiming Tian, Lang Yang, Zhouhong Ren, Jinliang Yuan, Lingkun Li, and Jiliang Wang. 2026. Is your NPU ready for LLMs? Dissecting the hidden efficiency bottlenecks in mobile LLM inference. *arXiv preprint arXiv:2607.05475* (2026). doi:10.48550/arXiv.2607.05475
- [9] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *International Conference on Machine Learning (ICML)*. 5209–5235. doi:10.48550/arXiv.2401.10774
- [10] Jian Chen, Yesheng Liang, and Zhijian Liu. 2026. DFlash: Block diffusion for flash speculative decoding. In *International Conference on Machine Learning (ICML)*. doi:10.48550/arXiv.2602.06036
- [11] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing mobile SoC for accelerating heterogeneous LLM inference. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*. 359–374. doi:10.1145/3731569.3764808
- [12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.
- [13] Wei Chen and Zhiyuan Li. 2024. Octopus v2: On-device language model for super agent. arXiv:2404.01744
- [14] Zhiyang Chen, Daliang Xu, Haiyang Shen, Chiheng Lou, Mengwei Xu, Shangguang Wang, Xin Jin, and Yun Ma. 2025. Accelerating mobile language model via speculative decoding and NPU-coordinated execution. arXiv:2510.15312
- [15] Jinha Chung, Byeongjun Shin, Jiin Kim, and Minsoo Rhu. 2026. Agent-X: Full pipeline acceleration of on-device AI agents. In *Proceedings of the 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*. 144–157. doi:10.1145/3745756.3809195
- [16] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. XGrammar: Flexible and efficient structured generation engine for large language models. In *Proceedings of Machine Learning and Systems (MLSys)*. doi:10.48550/arXiv.2411.15100
- [17] Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. TinyAgent: Function calling at the edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*. 80–88. doi:10.18653/v1/2024.emnlp-demo.9
- [18] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* 29, 3 (1986), 218–221. doi:10.1145/5666.5673
- [19] Elias Frantar, Saleh Ashkboos, Torsten Hoeffer, and Dan Alistarh. 2023. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2210.17323
- [20] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of LLM inference using lookahead decoding. In *International Conference on Machine Learning (ICML)*. 14060–14079. doi:10.48550/arXiv.2402.02057
- [21] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 10932–10952. doi:10.18653/v1/2023.emnlp-main.674
- [22] Georgi Gerganov et al. 2023. llama.cpp. <https://github.com/ggml-org/llama.cpp>. Accessed 2026-08-19.
- [23] Google AI Edge. 2026. AI Edge Function Calling guide for Android. https://developers.google.com/edge/mediapipe/solutions/genai/function_calling/android. Accessed 2026-08-22.
- [24] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 herd of models. arXiv:2407.21783
- [25] Mohd Ariful Haque, Fahad Rahman, Kishor Datta Gupta, Khalil Shujaee, and Roy George. 2025. TinyLLM: Evaluation and optimization of small language models for agentic tasks on edge devices. *arXiv preprint arXiv:2511.22138* (2025). doi:10.48550/arXiv.2511.22138
- [26] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach* (6 ed.). Morgan Kaufmann.
- [27] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model context protocol (MCP): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278* (2025). doi:10.48550/arXiv.2503.23278
- [28] Zhengxiang Huang, Chaoyue Niu, Zhaode Wang, Jiarui Xue, Hanming Zhang, Yugang Wang, Zewei Xin, Xiaotang Jiang, Chengfei Lv, Fan Wu, and Guihai Chen. 2025. MNN-AECS: Energy Optimization for LLM Decoding on Mobile Devices via Adaptive Core Selection. arXiv:2506.19884
- [29] Tian Jin, Ellie Y. Cheng, Zack Ankner, Nikunj Saunshi, Blake M. Elias, Amir Yazdanbakhsh, Jonathan Ragan-Kelley, Suvinay Subramanian, and Michael Carbin. 2025. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding. In *International Conference on Machine Learning (ICML)*. 27941–27956. doi:10.48550/arXiv.2502.11517
- [30] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 611–626. doi:10.1145/3600006.3613165
- [31] LangChain-AI. 2022. LangChain: A framework for building agents and LLM-powered applications. <https://github.com/langchain-ai/langchain>. Accessed 2026-08-31.

- [32] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*. 19274–19286. doi:10.48550/arXiv.2211.17192
- [33] Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. 2026. XGrammar-2: Dynamic and efficient structured generation engine for agentic LLMs. In *Proceedings of the ACM Conference on AI and Agent Systems*. 1009–1022. doi:10.1145/3786335.3813124
- [34] Pu Li, Jiawen Qi, and Qinyu Chen. 2026. When NPUs are not always faster: A stage-level analysis of mobile LLM inference. *arXiv preprint arXiv:2605.27435* (2026). doi:10.48550/arXiv.2605.27435
- [35] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning (ICML)*. 28935–28948. doi:10.48550/arXiv.2401.15077
- [36] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2503.01840
- [37] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. 2024. Personal LLM agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459* (2024). doi:10.48550/arXiv.2401.05459
- [38] Weibin Liao, Jian-guang Lou, and Haoyi Xiong. 2026. HyFunc: Accelerating LLM-based function calls for agentic AI through hybrid-model cascade and dynamic templating. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. doi:10.48550/arXiv.2602.13665
- [39] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Proceedings of Machine Learning and Systems (MLSys)*. doi:10.48550/arXiv.2306.00978
- [40] Jiachen Liu, Jae-Won Chung, Zhiyu Wu, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and enhancing quality-of-experience in LLM-based text streaming services. *arXiv preprint arXiv:2404.16283* (2024). doi:10.48550/arXiv.2404.16283
- [41] Mingdao Liu, Aohan Zeng, Bowen Wang, Peng Zhang, Jie Tang, and Yuxiao Dong. 2024. APAR: LLMs can do auto-parallel auto-regressive decoding. *arXiv preprint arXiv:2401.06761* (2024). doi:10.48550/arXiv.2401.06761
- [42] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuan-dong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. 2024. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In *International Conference on Machine Learning (ICML)*. 32431–32454. doi:10.48550/arXiv.2402.14905
- [43] Zirui Ma, Zhihua Fan, Wenxing Li, Haibin Wu, Fulin Zhang, Xiaochun Ye, and Wenming Li. 2026. AHASD: Asynchronous heterogeneous architecture for LLM adaptive drafting speculative decoding on mobile devices. In *Design Automation Conference (DAC)*. doi:10.1145/3770743.3803965
- [44] MLC team. 2023. MLC-LLM: Universal LLM Deployment Engine with ML Compilation. <https://github.com/mlc-ai/mlc-llm>. Accessed 2026-08-21.
- [45] Model Context Protocol. 2025. Model Context Protocol Specification, revision 2025-11-25. <https://modelcontextprotocol.io/specification/2025-11-25>. Accessed 2026-09-02.
- [46] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. 2024. Skeleton-of-Thought: Prompting LLMs for efficient parallel generation. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2307.15337
- [47] Songqin Nong, Jiali Zhu, Rui Wu, Jiongchao Jin, Shuo Shan, Xiutian Huang, and Wenhao Xu. 2024. MobileFlow: A multimodal LLM for mobile GUI agent. *arXiv preprint arXiv:2407.04346* (2024). doi:10.48550/arXiv.2407.04346
- [48] Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. 2025. SuffixDecoding: Extreme speculative decoding for emerging AI applications. In *Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2411.04975
- [49] Varatheepan Paramanayakam, Andreas Karatzas, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. 2025. Less is more: Optimizing function calling for LLM execution on edge devices. In *Design, Automation & Test in Europe Conference (DATE)*. 1–7. doi:10.23919/DATE64628.2025.10992798
- [50] Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning (ICML)*. 48371–48392.
- [51] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems (MLSys)*. doi:10.48550/arXiv.2211.05102
- [52] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2307.16789
- [53] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*. 68539–68551. doi:10.52202/075280-2997
- [54] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. doi:10.18653/v1/P16-1162
- [55] Xiaoxin Shi, Jiaxin Wan, Linkang Dong, Wei Jiang, Yue Liu, and Zengfeng Huang. 2026. SimpleTool: Parallel decoding for real-time LLM function calling. *arXiv preprint arXiv:2603.00030* (2026). doi:10.48550/arXiv.2603.00030
- [56] Yifan Sui, Han Zhao, Rui Ma, Zhiyuan He, Hao Wang, Jianxun Li, Kaiqiang Xu, Kai Chen, and Yuqing Yang. 2026. Parallelizing tool execution and LLM generation for low-latency agent serving. *arXiv:2603.18897*
- [57] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158* (2024). doi:10.48550/arXiv.2401.16158
- [58] Tuowei Wang, Fengzu Li, Yanfan Sun, Wei Gao, and Ju Ren. 2026. Lever: Speculative LLM inference on smartphones. *arXiv preprint arXiv:2605.16786* (2026). doi:10.48550/arXiv.2605.16786
- [59] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*. 50208–50232. doi:10.48550/arXiv.2402.01030
- [60] Xin Wang, Ziming Miao, Yi Zhu, Hui Shen, Zhongwei Wan, Fan Yang, and Mi Zhang. 2026. AgentSpec: Speculative decoding for batch inference of LLM agents. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.48550/arXiv.2608.24004

- [61] Zhaode Wang, Jingbang Yang, Xinyu Qian, Shiwen Xing, Xiaotang Jiang, Chengfei Lv, and Shengyu Zhang. 2024. MNN-LLM: A Generic Inference Engine for Fast Large Language Model Deployment on Mobile Devices. In *Proceedings of the 6th ACM International Conference on Multimedia in Asia Workshops (MMAsia Workshops)*. 1–7. doi:10.1145/3700410.3702126
- [62] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered task automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 543–557. doi:10.1145/3636534.3649379
- [63] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. 2024. AutoDroid-V2: Boosting SLM-based GUI agents via code generation. *arXiv preprint arXiv:2412.18116* (2024). doi:10.48550/arXiv.2412.18116
- [64] Brandon T. Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702* (2023). doi:10.48550/arXiv.2307.09702
- [65] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryan W. White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling (COLM)*. doi:10.48550/arXiv.2308.08155
- [66] Heming Xia, Yongqi Li, Cunxiao Du, Mingbo Song, and Wenjie Li. 2026. ToolSpec: Accelerating tool calling via schema-aware and retrieval-augmented speculative decoding. *arXiv:2604.13519*
- [67] Heming Xia, Yongqi Li, Jun Zhang, Cunxiao Du, and Wenjie Li. 2025. SWIFT: On-the-fly self-speculative decoding for LLM inference acceleration. In *International Conference on Learning Representations (ICLR)*.
- [68] Chang Xiao and Zixiaofan Yang. 2025. Streaming, fast and slow: Cognitive load-aware streaming for efficient LLM serving. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 1–13. doi:10.1145/3746059.3747721
- [69] Weikai Xie, Li Zhang, Shihe Wang, Rongjie Yi, and Mengwei Xu. 2025. DroidCall: A dataset for LLM-powered Android intent invocation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*. 9116–9134. doi:10.18653/v1/2025.findings-emnlp.484
- [70] Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023. LLMcad: Fast and scalable on-device large language model inference. *arXiv:2309.04255*
- [71] Daliang Xu, Wangsong Yin, Hao Zhang, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2025. EdgeLLM: Fast on-device LLM inference with speculative decoding. *IEEE Transactions on Mobile Computing* 24, 4 (2025), 3256–3273. doi:10.1109/TMC.2024.3513457
- [72] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast on-device LLM inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 445–462. doi:10.1145/3669940.3707239
- [73] Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. 2024. On-device language models: A comprehensive review. *arXiv preprint arXiv:2409.00088* (2024). doi:10.48550/arXiv.2409.00088
- [74] An Yang, Anfeng Li, Baosong Yang, et al. 2025. Qwen3 technical report. *arXiv:2505.09388*
- [75] An Yang, Baosong Yang, Beichen Zhang, et al. 2024. Qwen2.5 technical report. *arXiv:2412.15115*
- [76] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*. 50528–50652. doi:10.52202/079017-1601
- [77] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2210.03629
- [78] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. 2024. LLM as a system service on mobile devices. *arXiv preprint arXiv:2403.11805* (2024). doi:10.48550/arXiv.2403.11805
- [79] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. AppAgent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, 1–20. doi:10.1145/3706598.3713600
- [80] Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Quoc Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, Zhiwei Liu, Yihao Feng, Tulika Manoj Awalganekar, Rithesh R N, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. 2025. xLAM: A family of large action models to empower AI agent systems. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Association for Computational Linguistics, 11583–11597. doi:10.18653/v1/2025.naacl-long.578
- [81] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2024. Draft & Verify: Lossless large language model acceleration via self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. 11263–11282. doi:10.18653/v1/2024.acl-long.607
- [82] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2312.07104

A THE CORE SET OF THE MEASUREMENT PROTOCOL

The measurement protocol of Section 7 pins one engine thread to each of four CPU cores and fixes the clock of these cores. On the OnePlus 12R, the four cores are the A715 cores at 1.79 GHz; on the Pixel 7, they are the two X1 cores at 1.745 GHz and the two A78 cores at 1.663 GHz. These are the highest clocks that the two phones sustain under continuous load with the cooling fan on. The pinning is necessary. Without it, the Android scheduler stacks two or three of the four threads on one core and leaves another core idle, and the same engine decodes $1.4\times$ to $2.3\times$ slower on the 12R.

Table 10 explains the choice of these four cores. It lists the cost of a serial decode step, a four-sequence step and a prefill token on different core sets of the two SoCs, measured with llama.cpp’s batched benchmark on Qwen3-1.7B at Q8_0 with the clocks pinned as above.

Phone	Core set	Threads	Step (ms)		Ratio	Prefill ms/token
			serial	4-sequence		
12R	2 A715	2	65.0	112.3	1.73	20.1
12R	4 A715	4	54.2	65.9	1.22	9.8
12R	4 A715 + X3	5	43.2	50.8	1.18	5.6
12R	all 8	8	59.1	73.7	1.25	6.9
12R	3 A510 + X3	4	73.7	98.6	1.34	13.9
Pixel 7	2 X1	2	77.4	93.2	1.21	15.9
Pixel 7	2 X1 + 2 A78	4	79.2	88.0	1.11	10.2
Pixel 7	all 8	8	100.9	119.9	1.19	11.4
Pixel 7	4 A55	4	174.0	286.8	1.65	42.0

Table 10: Cost of a serial decode step, a four-sequence step and a prefill token on different core sets of the two phones; Ratio is the four-sequence step over the serial step; the protocol’s core set in bold.

Three observations follow. First, the extra cost of a four-sequence step over a serial step is the lowest on the four chosen cores, $1.22\times$ on the 12R and $1.11\times$ on the Pixel 7, and adding more cores does not lower it ($1.18\text{--}1.25\times$ on the 12R, $1.19\times$ on the Pixel 7). Second, the little cores slow down every step, serial and batched alike. The engine waits for all threads after every layer, so a pass runs at the speed of the slowest core. Using all eight cores is 9% slower than using four on the 12R and 27% slower on the Pixel 7, and the four little cores of the Pixel 7 alone are two to three times slower. Third, the largest core of the 12R, the X3, shortens every pass when it is added, the serial step by 20% and the prefill token by 43%. However, its clock is not sustainable under continuous load, and it shortens the baseline and SchemaWeaver alike, so the protocol leaves it out. The two X1 cores of the Pixel 7 already saturate its memory bus, since two threads decode as fast as four.