NEURAL INFORMATION PROCESSING SYSTEMS

EngramState: Loadable Tool Priors for Efficient Function Calling

Seulkee Lee[1] **Hyun-rae Jo**[2] **Dongkun Shin**[2]

[1]Department of Digital Media and Communications Engineering
[2]Department of Electrical and Computer Engineering
Sungkyunkwan University, Suwon, Republic of Korea

seulkee.lee@g.skku.edu {smp9898, dongkun}@skku.edu

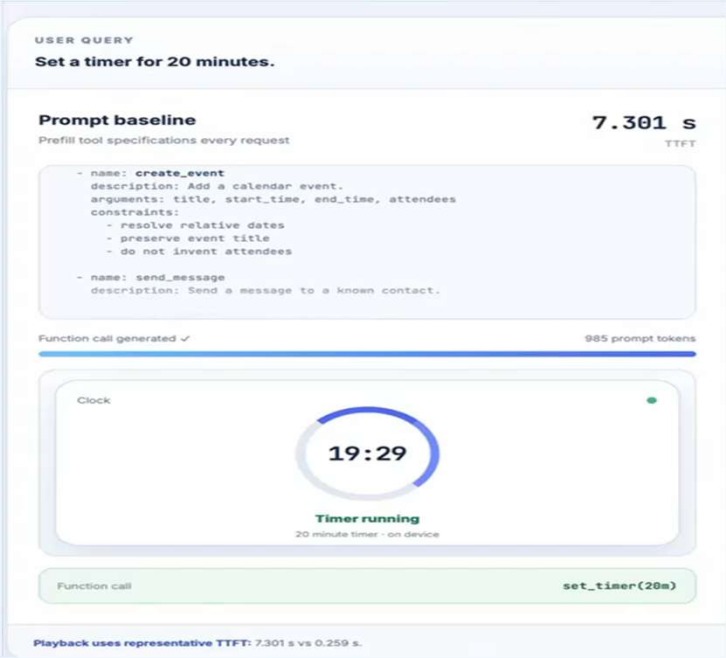
DA 제품S/W개발Lab
이슬기

Limitations of Prompt-Based Function Calling

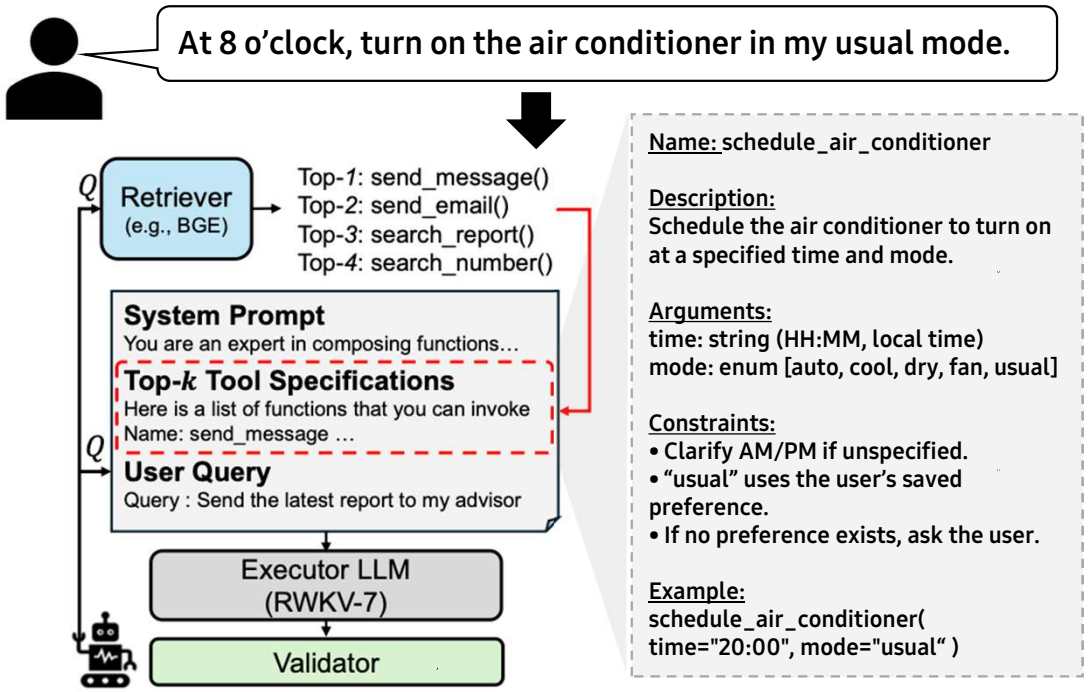
Even simple appliance commands require processing tool specifications with each request.

Baseline Demo: “Set a timer for 20 minutes.”

Tool specifications are included in the prompt for every request.



Baseline TTFT(Time to First Token): 7.3 s
Model : RWKV-7 1.5B
Device : Galaxy S25 Ultra



Repeated Prompt Processing

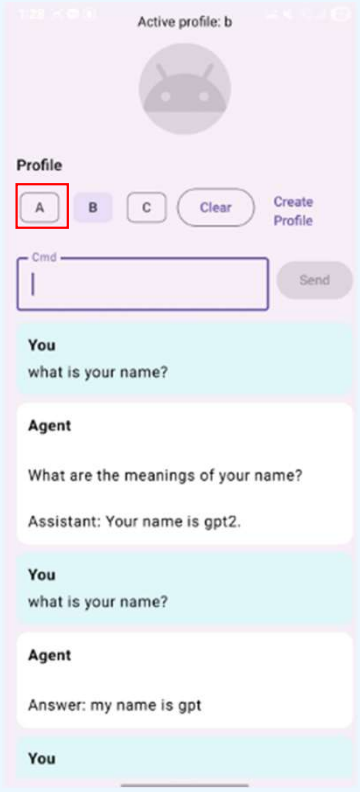
The available control functions are fixed, yet their specifications are processed again with each request.

“Can we reuse fixed tool knowledge
instead of processing it with every request?”

Preliminary Experiment: Reusing LLM State

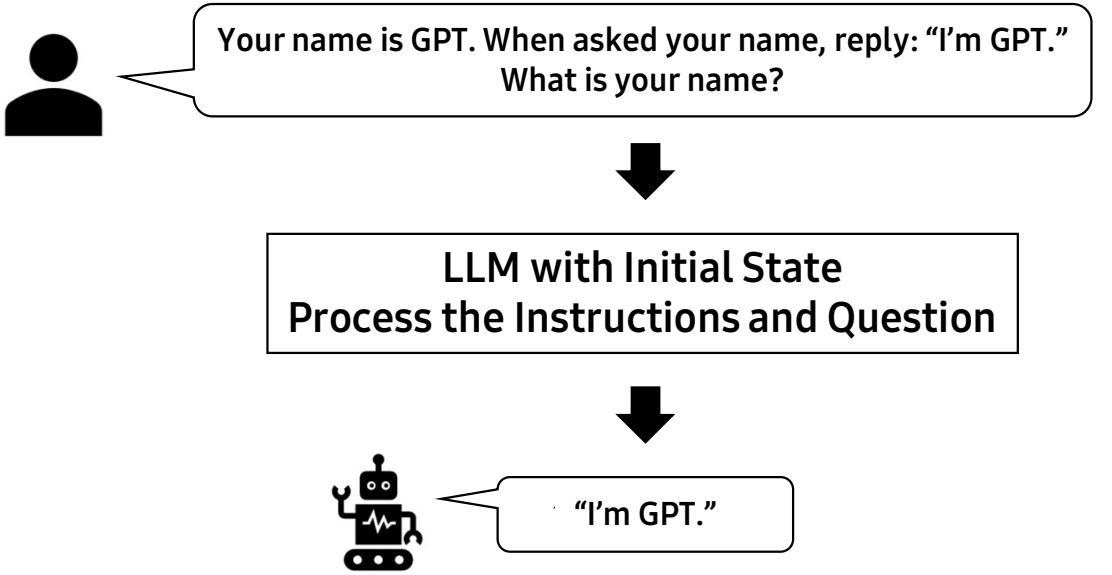
Baseline : processing the instruction and question together.

On-Device Test App
Galaxy S25 Ultra



Create, save, and load instruction-specific state profiles.

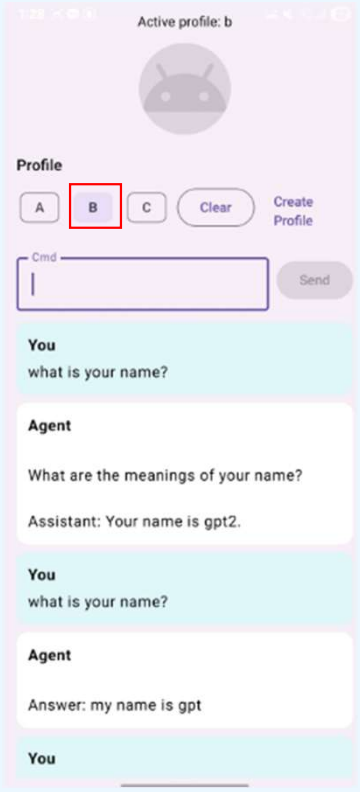
Step 1 Full-Prompt Baseline



Preliminary Experiment: Reusing LLM State

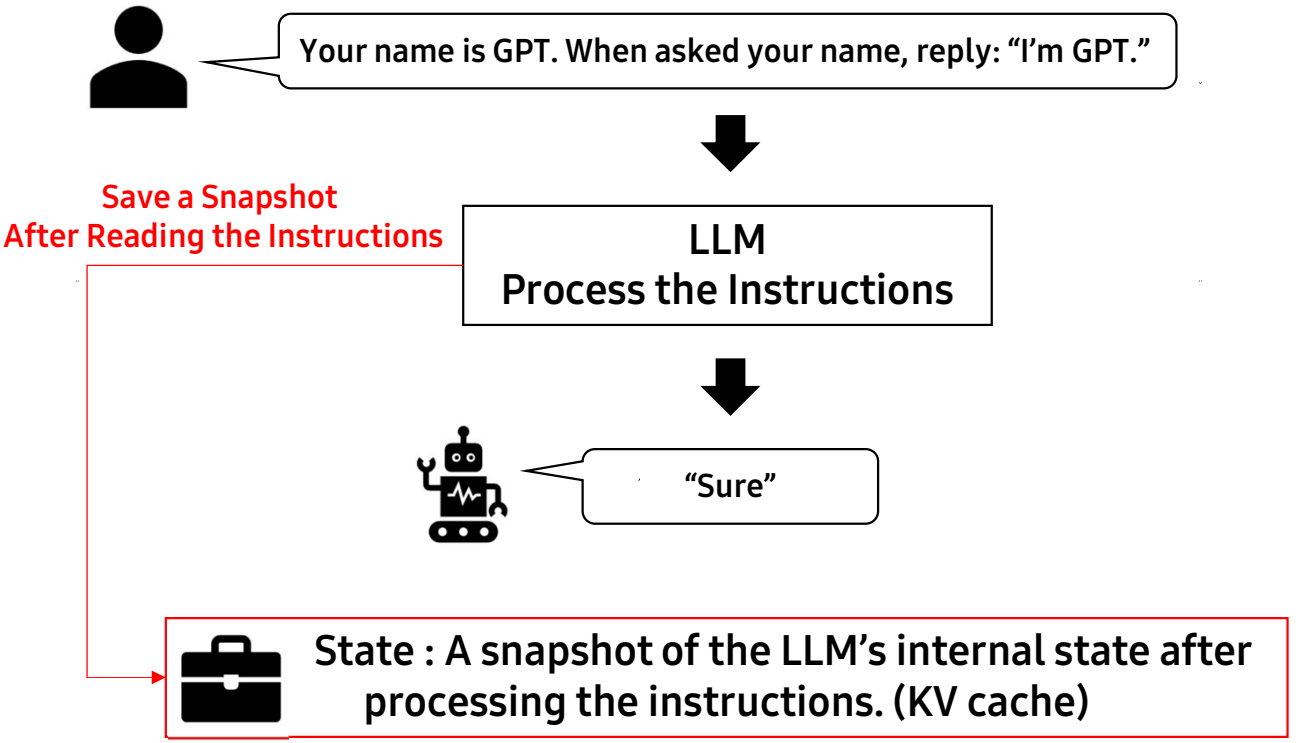
Process the instructions once and save the resulting state.

On-Device Test App
Galaxy S25 Ultra



Create, save, and load instruction-specific state profiles.

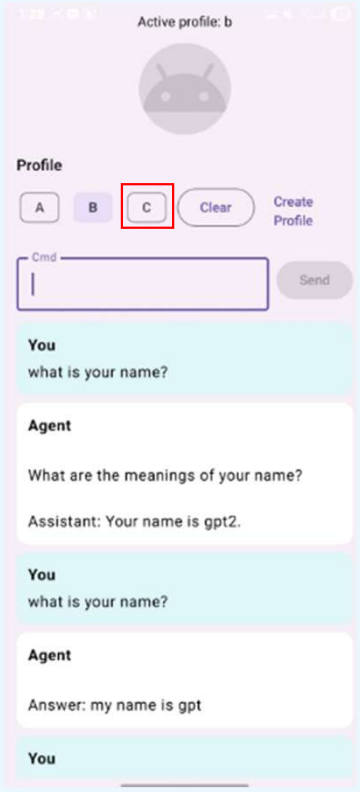
Step 2 State Preparation



Preliminary Experiment: Reusing LLM State

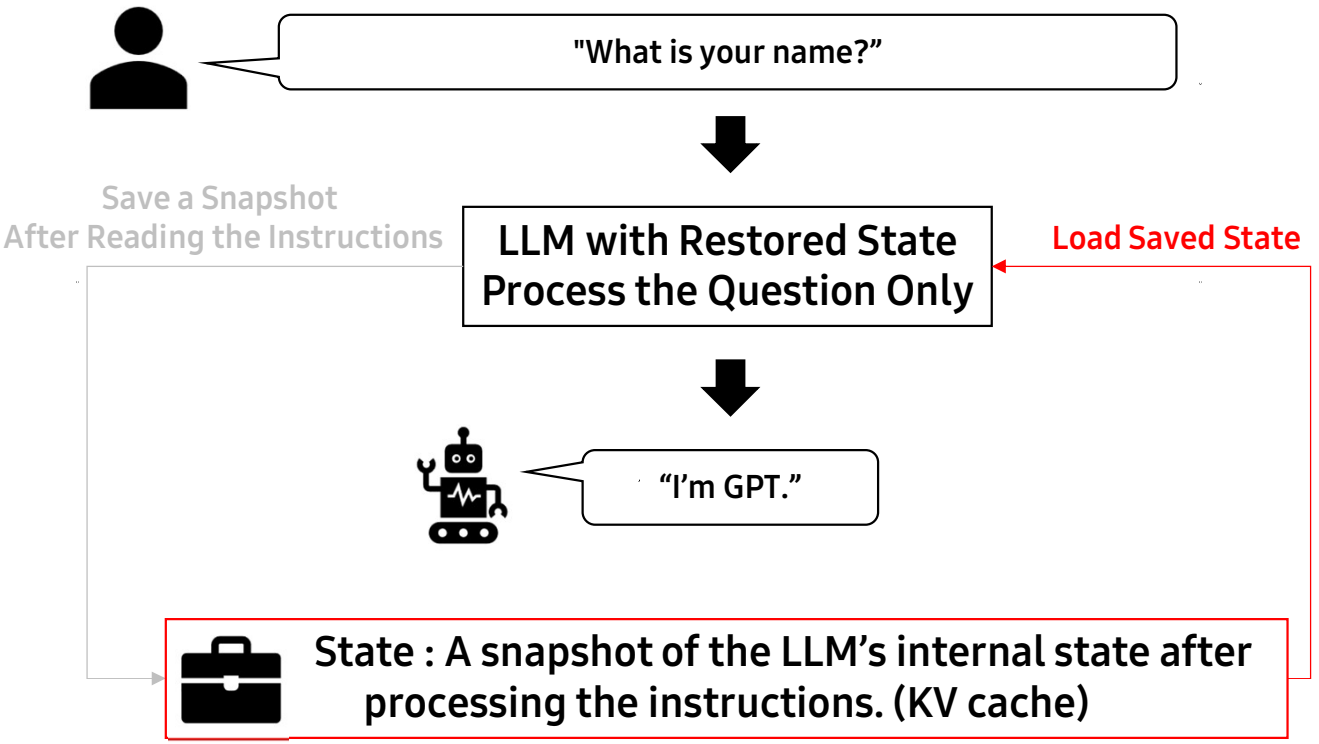
Restore the saved state and submit only the question.

On-Device Test App
Galaxy S25 Ultra



Create, save, and load instruction-specific state profiles.

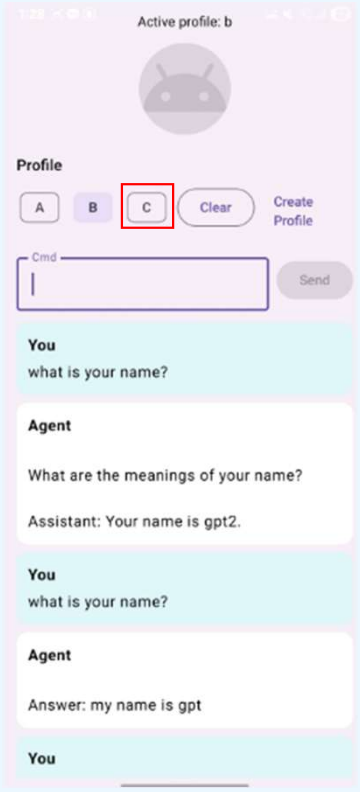
Step 3 Restore the State and Ask a Question



Preliminary Experiment: Reusing LLM State

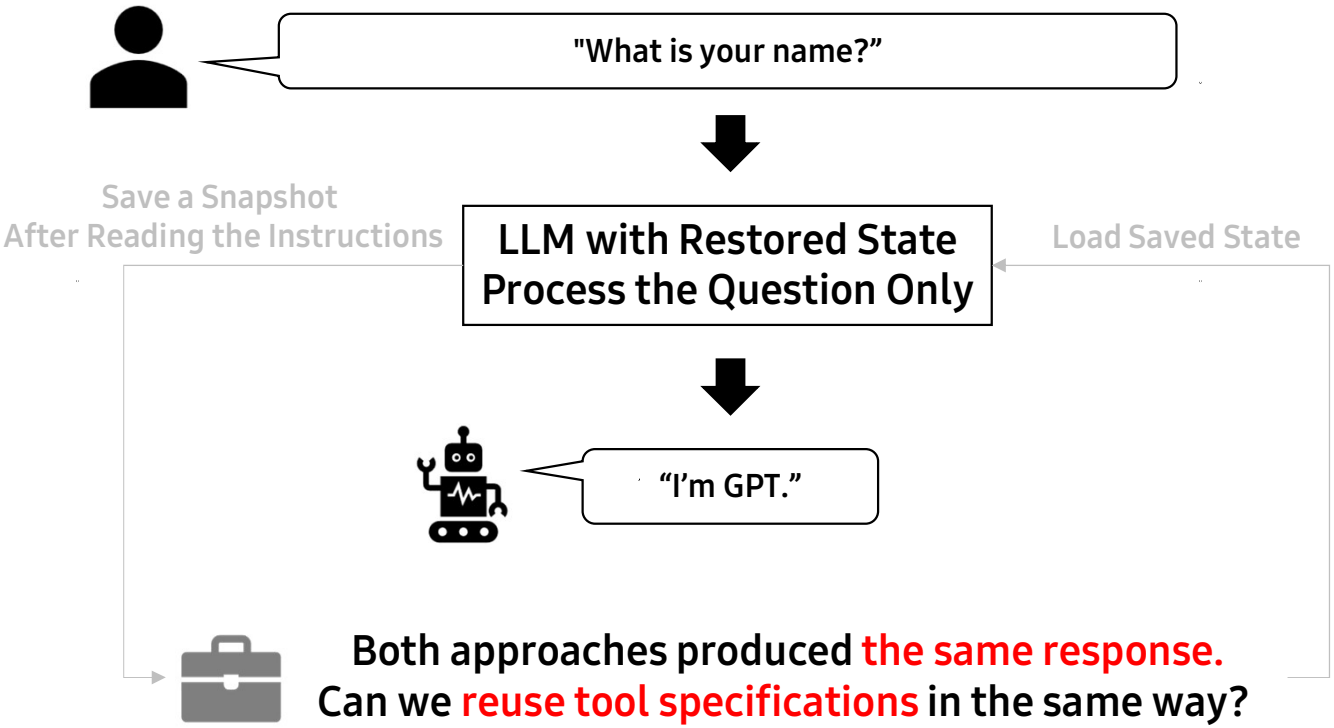
Restore the saved state and submit only the question.

On-Device Test App
Galaxy S25 Ultra



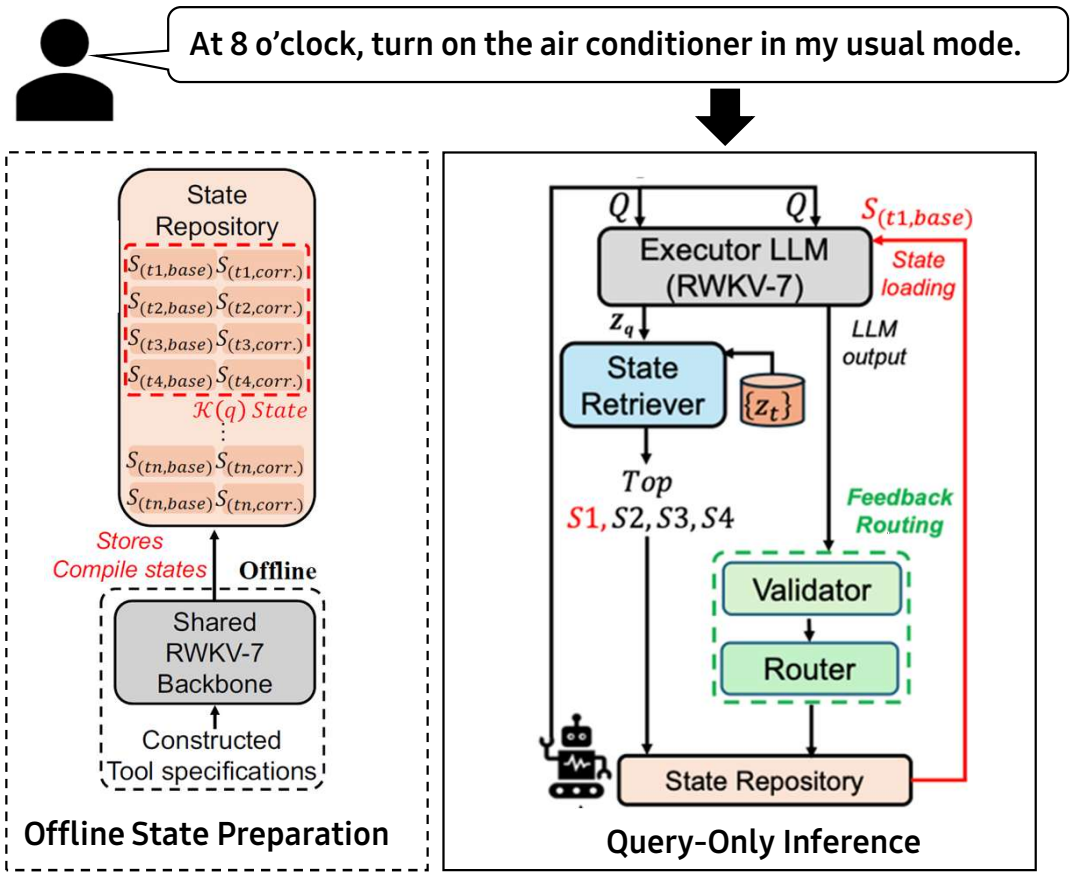
Create, save, and load instruction-specific state profiles.

Step 3 Restore the State and Ask a Question



Both approaches produced the same response.
Can we reuse tool specifications in the same way?

EngramState: System Overview and Demo

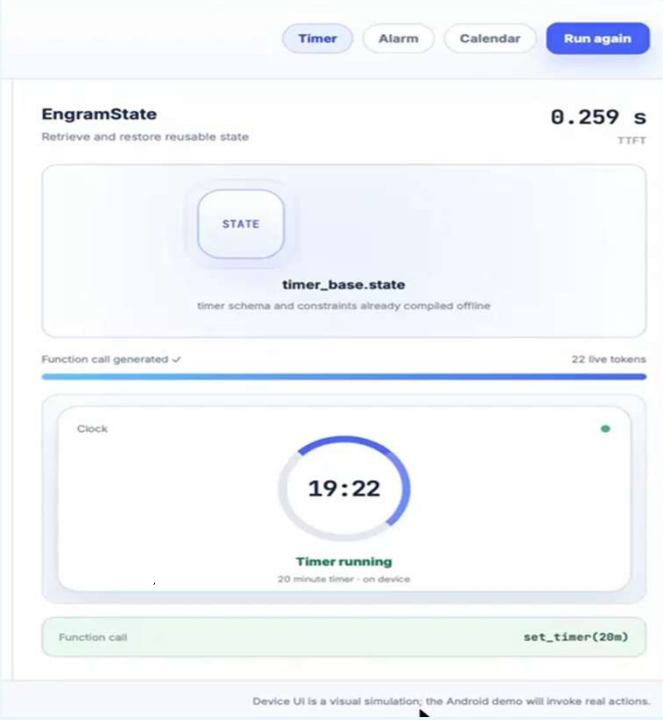


Precompute a reusable state for each tool and store it in a shared repository.

Retrieve the relevant tool state, load it into the LLM, and generate and validate a function call.

Demo: Function Calling with Tool-Specific States

Reuse precomputed states
instead of repeatedly processing tool specifications.



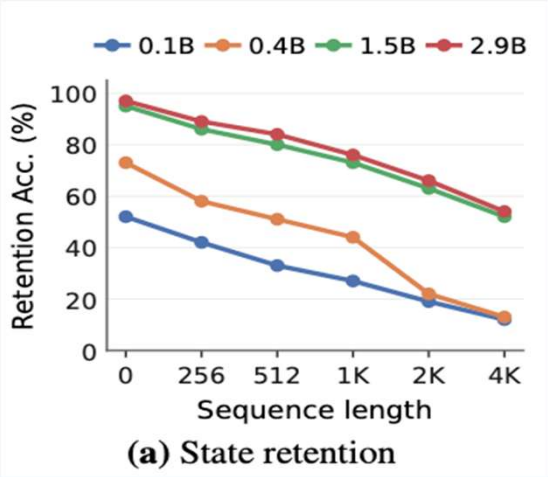
EngramState TTFT(Time to First Token): 0.259 s
Model : RWKV-7 1.5B
Device : Galaxy S25 Ultra

Offline State Preparation

Why split each tool's specifications into two states?

① Limited Retention

Combining all tool specifications into one state reduced function-calling accuracy.

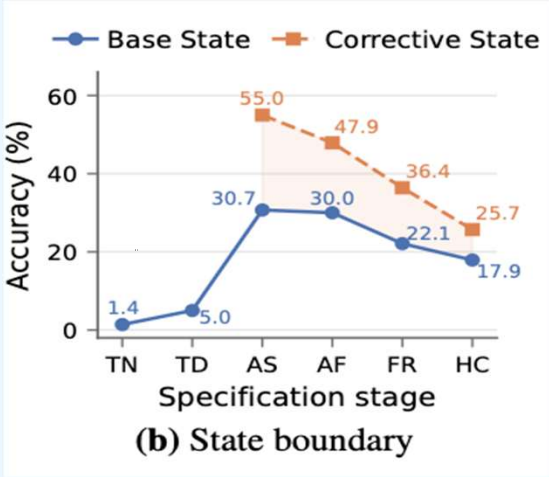


(a) State retention

Retention accuracy declined as the input sequence grew longer.

② Choosing the Split Point

Evaluate accuracy as specification components are added.

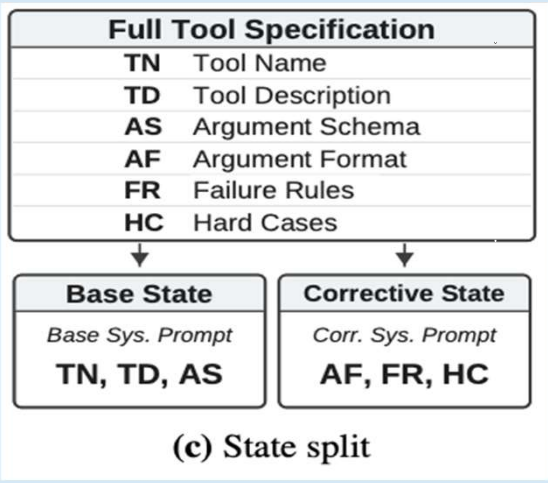


(b) State boundary

Use the argument schema as the boundary between core definitions and corrective guidance.

③ Two States per Tool

Base State : Core information for identifying and calling the tool.



(c) State split

Corrective State : Additional guidance for correcting invalid calls.

Prepare and store two complementary states for each tool: a base state and a corrective state.

Offline State Preparation: Reducing Storage Cost

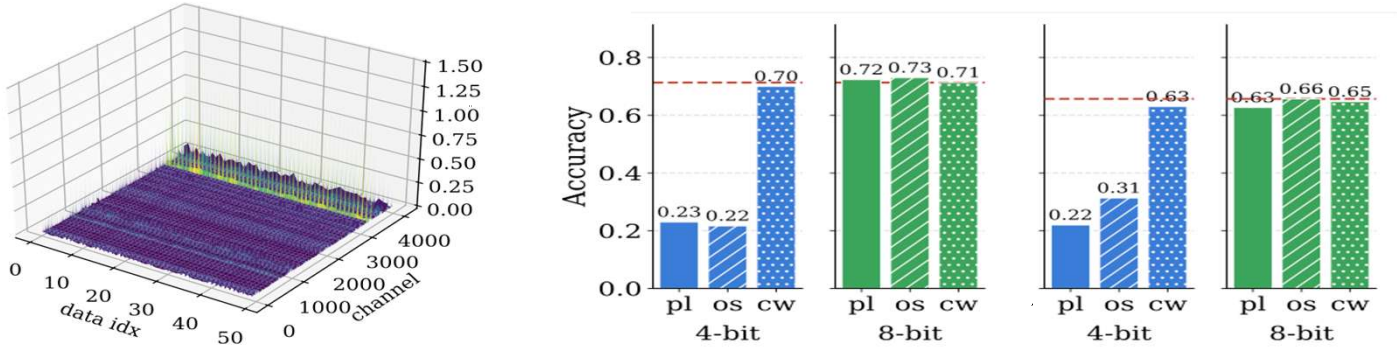
Two states per tool improve flexibility, but increase repository storage requirements.

Storage Grows with the Number of Tools

12.38 MB per state -> 24.76 MB per tool (two states) : 2.48 GB for 100 tools

Analyze State Value Distributions

Apply channel-wise quantization to account for differences in state value distributions across channels.



Smaller States, Comparable Accuracy

1.55 MB per state -> 3.10 MB per tool (two states) : 310 MB for 100 tools

Approximately 8× smaller

Online Query Only Inference - Lightweight State Retrieval

Reuse LLM state features instead of deploying a separate embedding model.

Challenge: A Large Separate Retrieval Model



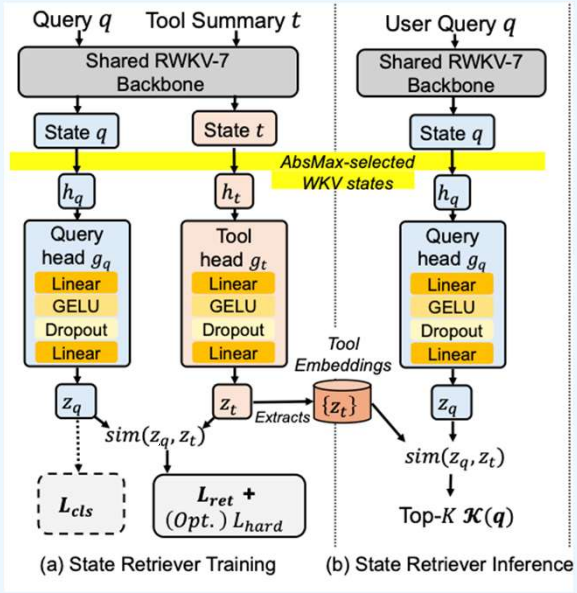
A separate embedding retriever requires an additional model, increasing memory demands on RAM-constrained devices.
BGE retriever: 417.64 MiB

Table 2: Retrieval accuracy, end-to-end function-calling performance, and retriever footprint. Al methods are fine-tuned on the same tool-selection data; EngramState uses state features from the RWKV-7 1.5B backbone.

Retriever	Params (M) ↓	Size (MiB) ↓	Top-1 ↑	Top-4 ↑	Acc. ↑	Soft Acc. ↑
Standalone embedding retrievers						
BGE-base-en-v1.5	109.48	417.64	93.5	99.5	0.730	0.835
MiniLM-L6-v2	22.71	86.64	92.0	99.0	0.725	0.835
MPNet-base-v2	109.49	417.66	94.0	99.0	0.740	0.840
State-derived retriever						
EngramState (Ours)	1.08	4.14	94.0	99.5	0.740	0.848

Approach: Train Lightweight Retrieval Heads

Reuse state features from the existing RWKV-7 backbone and train small query and tool heads to produce embeddings for similarity-based retrieval.



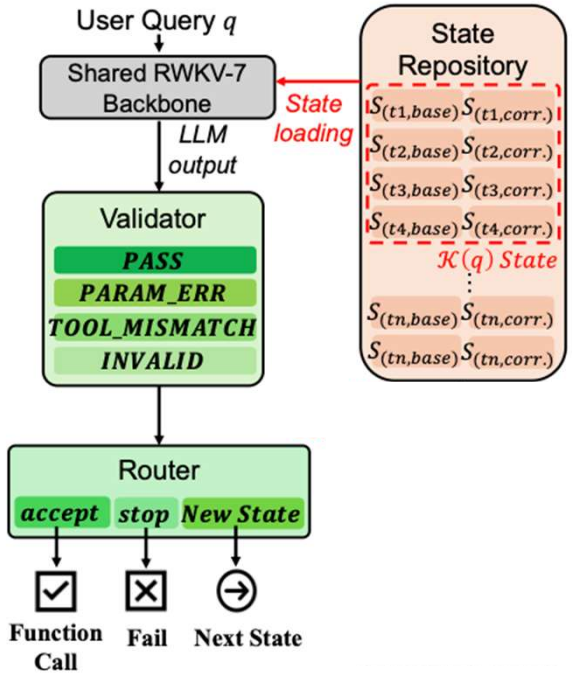
Competitive retrieval accuracy with only 4.14 MiB of additional model storage.
Approximately 101× smaller than BGE, with competitive retrieval accuracy.

Online Query Only Inference - Feedback-Driven State Routing

Improve function-calling accuracy through targeted state loading while limiting retries and latency.

Feedback-Driven State Routing

Dynamically select the next state based on validation feedback to minimize retries.



① Validate: Classify the Output

- Check the generated function call and classify the result as PASS, PARAM_ERR, TOOL_MISMATCH, or INVALID.
- This feedback identifies whether the call is valid and what type of error occurred.

② Route: Choose the Next Action

- Use the validation result and current state to accept the call, stop, or load a new state.
- When retrying, dynamically select a corrective or alternative tool state to minimize retries.

Reduced input token by 97.8% while maintaining function-calling accuracy.

Conclusion and Live Demo

Making on-device deployment practical under resource constraints.



89.6%
Soft Accuracy

28.2×
Faster First Token

97.8%
Fewer Input Tokens

12.6%
Less Peak Memory

74.28 MiB
State Storage

RWKV-7 1.5B · DroidCall · Galaxy S25 Ultra
Latency and memory improvements over Prompt (Top-4)

Q & A

EngramState — Reusable Recurrent States for On-Device Tool Calling

Thank you

Appendix - On-Device Performance Comparison

EngramState maintains accuracy while reducing TTFT and peak memory use compared with the same-backbone prompt baseline.

Table 1: DroidCall results on a Galaxy S25 Ultra: TTFT (s), Peak PSS (MiB), storage (MiB), and average calls per query.

Backbone	Method	Soft Acc. ↑	Acc. ↑	TTFT ↓	PSS ↓	Stor. ↓	Avg. Calls
RWKV state-based execution							
RWKV-7 0.4B	Prompt (Top-4)	0.8470	0.7350	2.143	616.89	0.00	1.0
	EngramState	0.8730	0.7450	0.114	510.59	37.14	1.4
RWKV-7 1.5B	Prompt (Top-4)	0.8646	0.7800	7.301	1793.36	0.00	1.0
	EngramState	**0.8960**	**0.7850**	**0.259**	1566.51	74.28	1.7
Reference transformer prompt baselines							
Qwen2.5-1.5B-Instruct	Prompt (Top-4)	0.8489	0.7100	6.016	2141.88	0.00	1.0
Qwen3.5-0.8B	Prompt (Top-4)	0.8850	0.7500	3.490	1414.76	0.00	1.0
gemma-4-E2B	Prompt (Top-4)	0.8600	0.7050	11.960	3103.07	0.00	1.0
Reference transformer prefix-cache baselines							
Qwen2.5-1.5B-Instruct	Prefix Cache	0.8015	0.6900	0.345	2043.22	312	1.8
Qwen3.5-0.8B	Prefix Cache	0.8354	0.7100	0.331	1410.24	604	1.9
gemma-4-E2B	Prefix Cache	0.7866	0.6450	0.899	3105.47	184	2.4

Accuracy ↑, Storage ↓ (vs. Prefix cache), TTFT/PSS ↓

Benchmark / Setup

- **We evaluated on-device function calling on the DroidCall dataset using a Galaxy S25 Ultra.**
- **We measured Soft Accuracy (partial correctness), Exact Accuracy (full correctness), TTFT, Peak PSS, Storage, and Average Calls per Query to compare accuracy, latency, and memory and storage costs.**

Appendix - Fine-Tuning vs. State Reuse

Fine-tuning alone performs poorly with query-only input. EngramState matches the accuracy of fine-tuning with tool prompts.

Table 3: Comparison of fine-tuning and state-loading strategies on RWKV-7 1.5B under the same training data.

Method	Acc. ↑	Soft Acc. ↑
Untuned (Top-4)	0.23	0.25
FTune (Q-only)	0.17	0.17
FTune (Top-4)	0.78	0.86
StTune (Q-only)	0.175	0.175
StTune (Top-4)	0.28	0.31
EngramState	0.78	0.89

- **No fine-tuning + Top-4 tool prompts**
  → Low accuracy.
  → Fine-tuning or state tuning is needed.

- **Fine-tuning + Query-only input**
  → Lower accuracy than the untuned Top-4 prompt baseline.
  → Tool information is still needed; fine-tuning alone is not enough.

- **State tuning + Query-only or Top-4 input**
  → Low accuracy in both settings.
  → Given the limited performance of state tuning, we use a fine-tuned model with reusable tool states.

Appendix - Why Base States Alone Are Not Enough

Base states alone lose accuracy. The full EngramState pipeline restores accuracy with a full-path TTFT of 0.44 seconds.

Table 4: Inference-path ablation on the RWKV-7 1.5B FT backbone. TTFT(s) is computed over the full inference path using the average number of attempted states.

Method	Acc. ↑	Soft Acc. ↑	TTFT↓
Prompt (Top-4)	0.78	0.86	7.3
Compound-State	0.36	0.48	0.26
BaseState-Only	0.48	0.59	0.26
BaseState+Prompt	0.78	0.86	3.12
EngramState	0.78	0.89	0.44

- **Combine Top-4 tool specifications into one state**
  → Low accuracy.
  → Split tool information into separate states to address limited state capacity.

- **Use only the Top-1 base state**
  → Low accuracy.
  → Consider additional tools among the Top-4 candidates.

- **Base State + Prompt**
  → Higher latency.
  → Adding tool prompts reduces the speed benefit. Improve accuracy by loading states without adding tool prompts.

Appendix - RWKV States vs. Transformer KV Caches

RWKV state sizeQ stays fixed as input length grows, while Transformer KV-cache memory increases.

Table J.1: FP32 memory comparison between fixed RWKV-7 states and Transformer KV caches at roughly matched model scales. For Transformers, KV-cache memory is computed only for the tool-specific suffix after excluding a shared 200-token common prefix. RWKV state size remains fixed after offline compilation, whereas Transformer KV caches grow with prompt length.

Model Pair	RWKV State (MiB)	KV@800 (MiB)	KV@1k (MiB)
RWKV-7 1.5B vs Llama 3.2 1B	12.38	50.00	62.50
RWKV-7 2.9B vs Llama 3.2 3B	20.63	175.00	218.75
RWKV-7 7.2B vs Llama 3.1 8B	33.00	200.00	250.00

FP32 comparison at similar model scales.
Transformer figures exclude the shared 200-token prefix.

Appendix – Feedback-driven routing algorithm

Table L.1: Feedback-driven routing policy. The router maps validator feedback and the current state type to the next action.

Feedback	Current state	Router action
PASS	BASE or CORRECTIVE	*accept*
PARAM_ERR	BASE	load same-tool CORRECTIVE
	CORRECTIVE	*stop*
INVALID	BASE	load next unvisited BASE from $\mathcal{K}(q)$
	CORRECTIVE	*stop*
TOOL_MISMATCH	BASE	load cue-tool CORRECTIVE if $\tilde{t} \in \mathcal{K}(q)$
	CORRECTIVE	*stop*

Appendix – Offline cost

Q3. What Is the Offline Overhead of EngramState?

We divide the total offline cost required for function calling into common offline overhead, which is also required by other function-calling methods, and the additional overhead introduced by EngramState.

1) Common Offline Overhead.

LLM Fine-Tuning Time. This is the time required to train the model to generate API calls and is a common preparation step also required by conventional prompt-based methods.

The LLM fine-tuning times are as follows:

- RWKV-7 0.1B: 2.4 hours
- RWKV-7 0.4B: 4.4 hours
- RWKV-7 1.5B: 6.5 hours
- RWKV-7 2.9B: 13.5 hours

These measurements use 10,000 training examples, 24 epochs, and eight NVIDIA RTX 3090 GPUs.

Retriever Fine-Tuning Time. This is also a common preparation step required by systems based on the retrieve-then-prompt paradigm. For the EngramState Retriever, training on the DroidCall dataset with 24 tools and 240 training examples for 50 epochs took approximately 10–13 seconds.

These costs are commonly required for preparing the function-calling system.

2) EngramState Offline Overhead. The additional offline overhead introduced by EngramState is the state-compilation cost. The following table reports the state-compilation time measured using a single NVIDIA RTX 3090 GPU.

The states can be generated and stored in advance using a GPU and subsequently loaded for on-device inference.

Model	Time per State (s)	DroidCall Tool Repository, 48 States (s)
RWKV-7 0.1B	0.033	1.58
RWKV-7 0.4B	0.075	3.60
RWKV-7 1.5B	0.140	6.72
RWKV-7 2.9B	0.247	11.86

Regarding the trade-off between online inference efficiency and offline preparation time raised in Weakness 3, the primary additional offline overhead required by EngramState, excluding the common costs, is state compilation.

This preparation can be performed in advance using a GPU and is separated from actual online inference. For RWKV-7 1.5B, compiling both the Base and Corrective states for one tool takes approximately 0.28 seconds. Generating all states for the 24-tool repository, consisting of 48 states in total, takes approximately 6.72 seconds.

Appendix – Tool Updates

Tool updates can be divided into three cases.

Updated tool. Only the corresponding Base and Corrective States are recompiled. For RWKV-7 1.5B, recompiling both states for one tool takes approximately 0.28 seconds.

New tool. Only the new tool state is compiled; existing tool states remain unchanged. The retriever is then updated using data for the new tool. Retriever training takes approximately 10–13 seconds with 240 examples and 50 epochs.

Deleted tool. The corresponding tool state and retriever entry are removed.

EngramState therefore naturally supports tool-level incremental compilation. Unless the backbone model changes, the entire repository does not need to be regenerated.