EFFICIENT ON-DEVICE AGENTS VIA ADAPTIVE CONTEXT MANAGEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

On-device AI agents offer the potential for personalized, low-latency assistance, but their deployment is fundamentally constrained by limited memory capacity. Context in agentic settings worsens this problem due to large static tool schemas and a growing interaction history that continually expands the persistent KV cache. To maintain on-device feasibility, agents must operate near the minimum task-sufficient context, while preserving task performance. We introduce two complementary mechanisms: (1) a learned intra-session memory architecture that distills trajectories into an append-only Context State Object (CSO), preserving current state and relevant details from previous steps that may have future utility, while still supporting KV-cache reuse, and (2) a just-in-time schema-passing mechanism that loads full tool definitions only upon tool selection. We instantiate this framework by adapting 3B-parameter SLMs and profiling them on smartphone hardware. In our evaluations, CSO-based memory outperforms KV-cache compression and trained summarization while retaining performance near or even above full-history execution. Deployed on-device, our framework reduces initial tool context by over $6\times$ and interaction-context growth by 10–$25\times$, substantially reducing initial prompt latency and KV-cache memory, enabling persistent and capable on-device agents.

1 INTRODUCTION

On-device AI agents can locally access private data and applications while providing low-latency, personalized assistance (Ferev et al., 2024; Chen et al., 2024). Much of the existing on-device literature focuses on fitting capable models within device constraints through smaller architectures, quantization, and efficient function-calling interfaces (Xu et al., 2024; Abdin et al., 2024; Erdogan et al., 2024). However, fitting the model weights is only part of the deployment problem. Persistent agents also accumulate context from tool definitions, user messages, tool calls, and environment observations, yet this growing context has received comparatively less attention as a deployment bottleneck. In our 3B, 4-bit model deployment on a Samsung Galaxy S25, retaining 10K–20K tokens requires roughly 500 MB–1 GB of additional KV-cache memory. Tool schemas impose a large cost before execution begins, while interaction history creates a growing cost throughout a task. These constraints directly conflict with two capabilities that make agents useful: access to a broad tool set and persistent state across long interactions.

Our motivating target is therefore the *minimum task-sufficient context*. How little context can an agent retain while still completing its task reliably? In tool-using agents, this requires managing both the static tool schemas and dynamic interaction history. The latter is particularly challenging because relevance is determined by *future utility*, not only by what matters to the next response. An identifier returned by one tool may become an argument several turns later, a failed call may constrain subsequent actions, and a completed step changes the remaining task state. We refer to such information as *agentically relevant context*. An effective memory mechanism should discard redundant interaction text while preserving this future-use state, along with information that tracks the agent's current progress.

Existing context-reduction approaches address parts of this problem. KV-cache compression methods (Xiao et al., 2024; Zhang et al., 2023) retain a smaller set of representations from the original sequence, but do not explicitly transform an agent trajectory into a concise task state. Learned mem-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

ory and summarization (Wang et al., 2025; Zhou et al., 2025) instead construct compact semantic representations, but commonly maintain them through replacement-style updates. Rewriting the persistent memory invalidates its cached prefix and may make the temporal evolution of progress, failures, and corrections less explicit. Separately, tool-retrieval methods such as TinyAgent (Erdogan et al., 2024) reduce the schemas exposed to the model initially, but do not address the interaction history accumulated after execution begins.

We introduce a learned intra-session memory architecture built around a *Context State Object* (CSO). After each interaction, a dedicated LoRA adapter converts new information into a concise CSO update capturing longer-horizon state such as user goals, exact values, completed steps, constraints, and persistent errors. A separate agentic adapter acts on the accumulated CSO rather than the raw conversation history to complete the user task. The CSO is strictly append-only and previous state is left unchanged while new information is added chronologically. This provides an explicit temporal record of task state and, importantly for on-device inference, allows previously processed CSO tokens to remain reusable in the persistent KV cache. We address the separate tool-context cost through schema minification and just-in-time (JIT) schema loading, exposing concise tool descriptions initially and loading a selected tool's full compact schema only when required.

We instantiate this framework using small language models and evaluate it against full-history execution, KV-cache compression, and both training-free and trained semantic summarization. We additionally test transfer on a different SLM backbone, as well as different out-of-distribution tasks from ToolAlpaca, and profile the system on smartphone hardware. **Our contributions are:**

- **A learned intra-session memory method for tool-using agents.** We introduce the Context State Object (CSO), a compact task-state representation trained to preserve current task state and information with potential future utility—including exact tool outputs, constraints, progress, and persistent errors—while discarding redundant interaction history.
- **A cache-efficient inference architecture for persistent on-device agents.** We maintain the CSO through incremental append-only updates using separate State-Tracker and Executor LoRA adapters over a shared SLM. This separates memory maintenance from action generation, preserves the temporal evolution of task state, and allows previously computed persistent KV-cache prefixes to remain reusable across turns.
- **Just-in-time tool-context management.** We reduce the complementary static context cost through deterministic schema minification and a two-stage interface that exposes concise tool descriptions initially and loads a full compact schema only after tool selection.

Across SLM backbones, and OOD, these mechanisms maintain strong task performance while reducing interaction-context growth by 10–$25\times$ and initial tool context by over $6\times$, with corresponding reductions in KV-cache memory and initial prompt latency on a Samsung Galaxy S25.

2 Related Work

2.1 On-Device Agents and Function Calling

Small Language Models (SLMs) increasingly support function calling and agentic execution directly on personal devices (Xu et al., 2024; Team et al., 2025; Abdin et al., 2024; Chen et al., 2024). This line of work has primarily focused on fitting capable models within device constraints through smaller architectures, quantization, and specialized function-calling training. Systems such as TinyAgent (Erdogan et al., 2024) additionally reduce tool-interface overhead by retrieving or exposing only a subset of relevant tools.

These approaches make local agent execution increasingly practical, but they primarily address model size or the tool context presented before execution. Persistent agents introduce a complementary resource cost: interaction state accumulates across user messages, tool calls, tool observations, and delegated responses. This cost is particularly important on-device because context directly consumes KV-cache memory and increases inference overhead throughout a trajectory. Our work focuses on this persistent context-management problem, while also considering the static tool-schema cost through just-in-time schema loading.

2.2 KV-Cache Compression

A broad class of inference methods reduces context cost by retaining a subset of token representations or reallocating KV-cache capacity. StreamingLLM (Xiao et al., 2024) preserves attention-sink

and recent tokens, while H_2O (Zhang et al., 2023) prioritizes tokens according to accumulated attention. More recent methods such as HeadKV-R2 (Fu et al., 2025) allocate cache capacity using head-specific signals associated with retrieval and reasoning.

These methods operate directly over representations derived from the original token sequence. This is attractive because compression can be applied without constructing a separate semantic memory, and some methods explicitly model which parts of the sequence are important for later reasoning. Our setting, however, places a different requirement on context: later tool execution may depend on sparse semantic state such as exact identifiers, completed steps, constraints, or persistent failures. Rather than selecting which original representations to retain, our State-Tracker constructs an explicit task-state representation containing information expected to affect future execution. We therefore view KV-cache compression and semantic task-state construction as distinct approaches to reducing persistent context, and use HeadKV-R2 as our primary token-level comparison.

2.3 Semantic Memory and Summarization

Semantic-memory systems reduce dependence on complete interaction histories by compressing, externalizing, or reorganizing prior context. Recursive summarization (Wang et al., 2025) periodically replaces accumulated dialogue with a shorter textual summary, while systems such as MemGPT (Packer et al., 2024) manage information across external memory tiers. Learned approaches such as MEM1 (Zhou et al., 2025) further show that models can be trained to maintain compact memory representations for long-running interactions.

Our work shares the goal of replacing verbose history with semantic memory, but targets a different combination of requirements. First, the retained state is trained specifically around information with potential future utility for tool execution, rather than general conversational recall. Second, the CSO is maintained incrementally: previously committed state is not regenerated when new information arrives. This preserves the temporal evolution of progress, errors, and corrections while keeping the corresponding persistent KV-cache prefix reusable across turns. Memory maintenance is also separated from action generation using swappable LoRA adapters over a shared base model (Hu et al., 2021).

To distinguish these factors from the general benefit of learned semantic compression, we directly compare against a trained replacement-summary baseline using the same base model, training data, and teacher supervision. This provides a closely matched comparison between regenerating complete compressed memory and maintaining task state through append-only updates.

3 Framework for Context-Efficient On-Device Agents

3.1 System Architecture

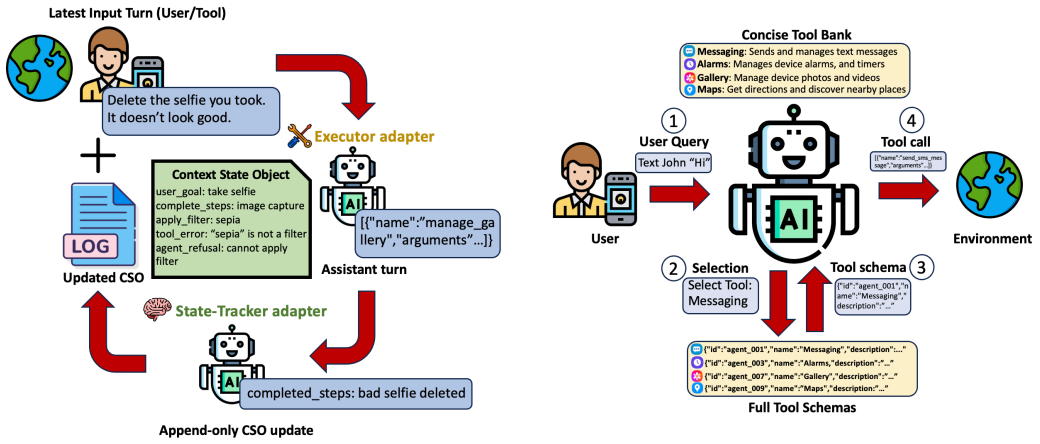
We instantiate our framework for context condensation in a hybrid on-device agent powered by a Small Language Model (SLM). Given a request, the agent can respond directly to a conversational input, invoke an on-device tool to manipulate local data, or delegate to a cloud agent for complex tasks and reasoning. Local tools can produce exact values needed in later steps, while cloud responses can introduce large amounts of text. Our context-management mechanisms are independent of the routing policy and focus on retaining the task state and tool information required for subsequent execution.

3.2 Intra-Session Memory via Context State Object

Rather than retaining the raw trajectory, we maintain a structured textual CSO containing concise longer-horizon task state, such as user goals, exact entity values, completed steps, active constraints, tool outcomes, and persistent errors. Information not expected to affect later execution is omitted.

Dual-Adapter Architecture. The CSO is maintained using two LoRA adapters (Hu et al., 2021) over a shared base model:

- **Executor ($\texttt{LoRA}_{\texttt{Exec}}$):** generates responses, tool calls, and cloud-delegation decisions conditioned on the current CSO.
- **State-Tracker ($\texttt{LoRA}_{\texttt{Mem}}$):** a 119M-parameter adapter that converts each completed interaction into a concise update to persistent task state.



(a) Interaction-history compression with the CSO. (b) Just-in-time loading from a concise tool bank.

Figure 1: Context-management mechanisms. (a) The State-Tracker converts interactions into compact persistent task state. (b) The tool interface loads a complete schema only after selection.

At step t, the Executor acts from CSO_{t-1} and the current observation. The State-Tracker then receives the new interaction and generates only the information to retain, Δ_t, appended to the state:

$$CSO_t = CSO_{t-1} \oplus \Delta_t.$$

Learning Future-Use Task State. The State-Tracker is trained from teacher-generated CSO updates to preserve information that may be useful in later turns rather than uniformly summarizing the interaction. The teacher is instructed to retain longer-horizon information that may affect subsequent execution and is additionally provided access to the full trajectory to identify these details.

Append-Only State and KV-Cache Reuse. Unlike replacement-style memory, the CSO never rewrites previously committed state. New updates are appended chronologically, preserving how the task evolves while avoiding repeated compression of previously retained information. This structure also supports incremental inference. Our runtime maintains separate persistent KV caches for the Executor and State-Tracker. Previously committed CSO tokens remain unchanged, so their cached representations remain valid across turns. Current user, assistant, and tool messages are processed as ephemeral context, after which the State-Tracker promotes the information that should persist into Δ_t. The system therefore processes newly appended state and the current interaction without reprocessing the previously committed CSO. The CSO uses a concise textual key-value representation and remains directly inspectable, allowing retained information and memory failures to be analyzed. Appendix A.12 provides a complete cache-management walkthrough.

3.3 Tool Schema Management

Tool definitions create a separate static context cost that is present before execution begins and grows with the available capability set.

Deterministic Schema Minification. Standard API representations such as OpenAPI (OpenAPI Initiative, 2024) contain formatting and metadata not required by our function-calling interface. We deterministically convert each definition into a compact representation retaining the tool name, description, parameters, and required arguments while removing non-essential formatting and whitespace. The agent is fine-tuned directly on this representation. An example is provided in Table 8.

Just-in-Time Schema Loading. Even compact full schemas scale linearly with the number of available tools. We therefore separate tool selection from argument generation (Figure 1b). During the **selection stage**, the agent sees only tool names and concise descriptions. After selecting a tool, its full compact schema is injected during the **execution stage** for argument generation. This preserves awareness of the full capability set without placing every schema in the initial prompt.

3.4 Data Generation and Training

Training requires supervision for agent execution, JIT tool selection, and CSO state updates. We construct this data using a multi-turn synthetic pipeline adapted to our on-device tool environment.

Trajectory Generation. We adapt the APIGen-MT pipeline (Prabhakar et al., 2025) using Gemini 2.0 Flash (Google Cloud, 2024) to synthesize multi-turn trajectories. We construct 5,000 tools by applying variations to 100 base tool families. For each scenario, separate instances of the LLM simulate the user, assistant, and tool-execution environment, with the environment including both successful and failed tool outcomes. Trajectories are first validated against their intended tool sequence and then filtered using an LLM-as-a-judge to remove inconsistent or unnatural interactions. The resulting data spans multi-tool, mixed on-device/cloud, single-tool, cloud-only, and conversational interactions, with multi-step tasks emphasized to exercise persistent task state. Distribution details, generation prompts, and filtering procedures are provided in Appendix A.11.

Executor Training. We fine-tune xLAM-2 3B (Zhang et al., 2025) using LoRA (Hu et al., 2021). Each example contains 10–14 candidate tools, including the required tools, cloud agent, and distractors. All training uses the compact schema representation. For JIT training, the model first selects from concise tool descriptions and then receives the selected full schema for argument generation.

State-Tracker Training. The State-Tracker is trained separately from teacher-generated updates. For each interaction, the teacher receives the previous CSO, the newly observed interaction, the task goal, and the complete generated trajectory, and produces only the incremental state that should persist. Providing the complete trajectory during supervision allows the teacher to identify details that become useful only later in the trajectory. The supervision prioritizes token efficiency and future utility, yielding targets for $\texttt{LoRA}_{\texttt{Mem}}$ that transform verbose interactions into append-only deltas.

4 Experimental Setup

We evaluate whether context reduction preserves task capability, transfers across interaction and model distributions, and yields practical on-device resource savings.

4.1 Evaluation Settings

Held-Out Tools and Simulators. Our primary evaluation contains 406 multi-turn tasks spanning multi-tool execution, cloud delegation, mixed on-device/cloud tasks, and conversational interactions. The tasks follow the same broad generation framework as training but use 19 entirely held-out tool definitions. Evaluation is interactive, since each method constructs a different context as the trajectory evolves. Training trajectories and State-Tracker supervision are generated using Gemini 2.0 Flash (Google Cloud, 2024). We evaluate under both Gemini 2.0 Flash and Qwen3.5-122B, which is not used in the training-data generation pipeline. The Gemini simulator is prompted to provide relevant information and clarifications readily, whereas the Qwen3.5 simulator is instructed not to volunteer information unless requested.

External Distribution and Backbone Transfer. We additionally evaluate on ToolAlpaca (Tang et al., 2023), an independently constructed external tool and task distribution. The primary evaluated model backbone is xLAM-2 3B (Zhang et al., 2025), but we analyze transfer by training Baseline, Memory-Efficient, and Combined variants on Qwen3 1.7B.

4.2 Baselines and Variants

Our primary comparison set represents three approaches to interaction context: **Baseline (FT)** retains the complete history, **HeadKV-R2** (Fu et al., 2025) performs token-level KV compression, and **Trained Summarization** learns a compact replacement memory. The trained summarization baseline uses the same base model, training data, teacher supervision, and overall training protocol as our memory model, but is trained to regenerate the complete compressed state rather than append a state delta. We additionally include the pretrained xLAM-2 model, training-free Recursive Summarization (Wang et al., 2025), StreamingLLM (Xiao et al., 2024), and H_2O-style Attention Eviction (Zhang et al., 2023) in our evaluations for broader coverage.

Our framework has three configurations:

- **Tool-Efficient:** compact schemas with JIT schema loading and full interaction history
- **Memory-Efficient:** append-only CSO with the complete compact tool bank
- **Combined:** CSO-based memory together with JIT schema loading.

Memory-Efficient isolates our interaction-memory contribution, while Combined is the full context-reduction configuration.

	Cloud Delegation			Multi-Tool			On-Device + Cloud		
Model Variant	F1	P	R	F1	P	R	F1	P	R
xLAM-2 3B	0.41	0.53	0.63	0.87	0.86	0.91	0.57	0.65	0.63
Baseline (FT)	**1.00**	**1.00**	**1.00**	0.83	0.79	0.92	0.88	0.94	0.87
Recursive Summarization	0.98	0.98	**1.00**	0.67	0.62	0.79	0.81	0.86	0.80
StreamingLLM	0.97	0.96	**1.00**	0.73	0.70	0.79	0.81	0.87	0.80
Attention Eviction	0.99	0.98	**1.00**	0.75	0.72	0.84	0.85	0.91	0.83
Tool-Efficient	0.93	0.97	0.95	0.86	**0.92**	0.88	0.73	0.85	0.74
Memory-Efficient	**1.00**	**1.00**	**1.00**	0.87	0.87	0.90	0.89	**0.95**	0.88
Combined	0.99	0.98	**1.00**	**0.93**	**0.92**	**0.95**	**0.94**	0.92	**0.92**

Table 1: Performance under the Gemini 2.0 Flash simulator. Qualitative interaction scores and conversational-only results are reported in Appendix A.8.

	Cloud Delegation			Multi-Tool			On-Device + Cloud		
Method	F1	P	R	F1	P	R	F1	P	R
Baseline (FT)	0.94	0.97	0.91	0.51	0.75	0.39	0.63	0.87	0.49
HeadKV-R2	0.95	0.97	0.93	0.59	0.81	0.46	0.60	0.88	0.46
Trained Summarization	0.93	0.96	0.90	0.64	0.75	0.56	0.66	0.80	0.56
Memory-Efficient	**1.00**	**1.00**	**1.00**	0.69	0.74	0.65	**0.73**	0.86	0.63
Combined	**1.00**	**1.00**	**1.00**	**0.73**	0.90	0.61	0.72	0.92	0.59

Table 2: Performance under the Qwen3.5-122B simulator against full-history execution, HeadKV-R2, and matched trained replacement summarization.

4.3 Metrics

Task Performance. We compare the tools invoked during each trajectory against the required tools and report precision, recall, and F1. A true positive corresponds to invoking a required tool, precision penalizes unnecessary or incorrect tool choices, while recall measures coverage of required tools. We compute these metrics per trajectory and macro-average precision, recall, and F1 across examples. This precision-recall decomposition is particularly informative for JIT loading, where incorrect first-stage selection can prevent access to the required schema. We additionally report a 1–5 LLM-as-judge measure of overall interaction quality in Appendix A.8. We use paired Wilcoxon signed-rank tests over matched evaluation instances for the principal comparisons between our variants and the full-history baseline.

Context and Device Efficiency. We separately measure **initial tool context** and **interaction-context growth**, corresponding respectively to the static schema cost and the persistent history accumulated during execution. We profile the Q4_K_M-quantized xLAM-2 3B model on a Samsung Galaxy S25 CPU using `llama.cpp`, measuring KV-cache memory, initial prompt latency, and State-Tracker update latency. Latency measurements are averaged across 10 trials.

5 Results

5.1 Task Performance and Architectural Trade-offs

We evaluate all methods interactively, allowing each context-management strategy to construct its own state as the trajectory evolves. Table 1 reports results under the Gemini 2.0 Flash simulator, while Table 2 reports the principal comparison under the second Qwen3.5-122B simulator.

Comparison with Context-Compression Baselines. Under the Gemini simulator, Memory-Efficient matches or improves upon full-history execution across the tool-using settings, while the training-free compression baselines degrade more noticeably on the stateful Multi-Tool and On-Device+Cloud tasks. Combined achieves the strongest overall performance in these two settings while additionally reducing static tool context.

The Qwen3.5 evaluation provides a more challenging interaction setting and shows a related but more informative pattern. Both HeadKV-R2 and trained replacement summarization improve over full-history execution on Multi-Tool tasks, suggesting that reducing accumulated history can itself be beneficial for performance. Memory-Efficient nevertheless performs better than both compression baselines across the stateful task categories and also improves upon full-history execution. Taken

	Cloud Delegation			Multi-Tool			On-Device + Cloud		
Method	F1	P	R	F1	P	R	F1	P	R
Baseline (FT)	**1.00**	**1.00**	**1.00**	0.63	0.81	0.52	0.67	0.89	0.54
Memory-Efficient	**1.00**	**1.00**	**1.00**	0.70	0.86	0.59	0.71	0.93	0.57
Combined	0.98	0.99	0.97	**0.76**	0.92	0.65	**0.72**	0.95	0.58

Table 3: Backbone-transfer results using Qwen3 1.7B.

together, these comparisons suggest that the benefit is not only due to retaining fewer tokens, but also to how persistent task state is represented.

Effect of JIT Schema Loading. Combined generally shifts toward higher precision and lower recall relative to Memory-Efficient under the Qwen3.5 evaluation. This is consistent with the additional selection stage introduced by JIT. When the required tool is not selected initially, its full schema is unavailable during argument generation. JIT therefore provides an additional reduction in static tool context while introducing a stricter tool-selection bottleneck.

Paired Wilcoxon signed-rank tests show significant differences between Memory-Efficient and the full-history Baseline ($p < 0.03$), and between Combined and the Baseline ($p < 0.01$).

5.2 Generalization and Transfer

We test transfer by varying the SLM backbone and the task distribution under the Qwen3.5 simulator.

Backbone Transfer. Table 3 repeats the core comparison using Qwen3 1.7B. The trend observed with xLAM-2 largely persists. Memory-Efficient improves over full-history on the Multi-Tool and On-Device+Cloud settings, while Combined provides a further gain on Multi-Tool tasks. This indicates that the CSO-based context-management is not specific to the xLAM-2 backbone.

External Tool Distribution. Table 4 evaluates transfer to ToolAlpaca, which consists of tasks with more parameters per tool call and more granular details (e.g. ticket number, plane number) in the trajectory. Memory-Efficient remains close to the full-history Baseline while outperforming HeadKV-R2 and Trained Summarization. Unlike the earlier evaluation, however, the compressed configurations exhibit a clearer recall cost, showing that interaction compression is not lossless on this distribution. Combined shifts further toward precision at the expense of recall, consistent with the additional JIT selection constraint. These results suggest that Memory-Efficient provides the stronger capability-context operating point on the external distribution, while Combined represents a more aggressive reduction in static tool context.

Method	**F1**	**P**	**R**
Baseline (FT)	**0.69**	0.56	**0.90**
HeadKV-R2	0.54	0.40	0.83
Trained Summarization	0.60	0.52	0.71
Memory-Efficient	0.67	0.63	0.72
Combined	0.65	**0.74**	0.58

Table 4: Performance on ToolAlpaca, an external tool and task distribution.

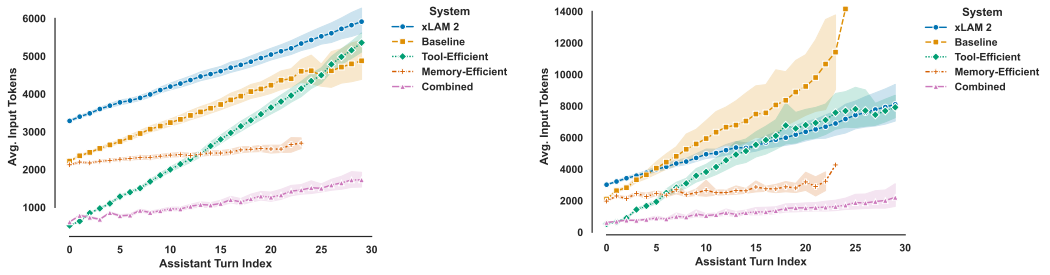
5.3 Context Growth and On-Device Efficiency

The two mechanisms address distinct deployment costs: JIT reduces the static tool context at initialization, while the CSO limits the dynamic interaction context accumulated throughout a trajectory.

Initial Tool Context. In our evaluations, xLAM-2 schemas require 3,246 initial tokens on average. Deterministic schema minification reduces this to 2,120 tokens, while JIT loading reduces the initial tool context further to 347 tokens. JIT therefore provides a $6.1\times$ reduction relative to compact full-schema execution and a $9.4\times$ reduction relative to the original schema representation.

Interaction-Context Growth. Figure 2 shows how context evolves over multi-turn execution. In Multi-Tool scenarios, full-history execution grows to approximately 5,000 tokens, while Memory-Efficient adds only 100–200 persistent tokens over 25 turns, corresponding to roughly a $10\times$ reduction in interaction-context growth. Tool-Efficient begins from a smaller tool prompt but continues to accumulate raw interaction history, whereas Combined reduces both sources of context overhead.

The difference is larger for Cloud Delegation, where verbose cloud responses cause the full-history context to grow beyond 10,000 tokens. The State-Tracker instead promotes selected information from these responses into concise CSO updates. Memory-Efficient adds approximately 500 persistent tokens over these trajectories, yielding more than a $20\times$ reduction in interaction-context growth.



(a) Multi-Tool (b) Cloud Delegation

Figure 2: Average model-visible input context over assistant turns for (a) Multi-Tool and (b) Cloud Delegation trajectories. Shaded regions denote 95% confidence intervals.

Across the evaluated trajectory types, CSO-based memory reduces interaction-context growth by approximately 10–25×.

On-Device Memory and Latency. We profile the quantized xLAM-2 3B deployment on a Samsung Galaxy S25 CPU. The 3,246-token standard initial context requires approximately 177 MB of KV-cache memory, while the 347-token JIT configuration requires approximately 55 MB and reduces initial Time-To-First-Token by approximately 5×.

As trajectories grow, retaining 10K–20K tokens requires roughly 500 MB–1 GB of KV-cache memory in our deployment. CSO-based execution instead keeps the evaluated long-running contexts at approximately 1K–2K active tokens, substantially reducing this growing memory cost. State tracking adds an average 1.31 s post-turn inference pass. Because committed CSO state is append-only, its previously computed KV-cache prefix remains reusable on subsequent turns.

6 Discussion

Our results suggest that context efficiency for persistent agents depends not only on retaining fewer tokens, but on preserving sparse information with future utility, such as exact identifiers, prior failures, and completed steps. We consider three implications for persistent-agent context management.

Persistent context should reflect future-use task state. The qualitative trajectories illustrate two different failure modes of history-based context management. With token-level compression (e.g. KV-cache compression), we observe cases where exact tool outputs, identifiers, or earlier error feedback needed by a later action are no longer retained. Full-history execution avoids this form of information loss, but must instead reason over an increasingly long trajectory. In several such trajectories, relevant earlier information remains present but is not effectively used by later actions, including repeated tool calls despite preceding corrective feedback. We interpret this behavior as consistent with attention dilution as interaction history grows.

The CSO takes a different approach by explicitly promoting selected information from the interaction into persistent semantic task state. Rather than deciding which representations from the original token sequence should remain accessible, the State-Tracker determines which information should continue to influence future execution. This distinction is reflected in the quantitative results, where Memory-Efficient provides a stronger capability-context operating point than the evaluated token-level compression methods across the stateful settings. The qualitative trajectories further show how exact values, progress, and persistent tool errors can remain explicit after the surrounding interaction has been discarded. Appendix A.14 provides an illustrative example of this error-persistence behavior.

Semantic compression and incremental inference impose different requirements. The comparison with trained replacement summarization suggests that learned semantic compression alone does not fully explain the performance of the CSO architecture. The trained summarization baseline uses the same base model, training data, teacher supervision, and overall training protocol, but regenerates the complete compressed state after each interaction. The CSO instead represents per-

sistent memory as a sequence of incremental updates, leaving previously committed state unchanged as new information is appended.

This non-destructive representation has two useful properties. First, the evolution of the task remains explicit: completed actions, failures, corrections, and newly observed constraints are retained in the order in which they arise rather than being repeatedly rewritten into a replacement summary. Our qualitative analysis suggests that this temporal structure is useful in trajectories where later actions depend on persistent errors, corrections, or intermediate state. This property may be particularly important in our aggressive compression setting, where the raw interaction history is discarded after state extraction and the CSO becomes the only persistent representation available to subsequent execution. Repeatedly rewriting this representation can therefore remove or blur intermediate progress, failures, and corrections that would otherwise remain recoverable from retained history. Further, append-only state is directly compatible with incremental inference. Because previously committed CSO tokens do not change, their cached KV representations remain valid across turns, allowing the system to process newly appended state without recomputing the persistent prefix.

Static and dynamic context create distinct deployment trade-offs. Interaction memory and tool schemas contribute different forms of context overhead. CSO-based memory limits the dynamic context accumulated as an interaction progresses, whereas JIT schema loading reduces the static tool context required before execution begins. The device measurements reflect this separation. Reducing interaction history primarily lowers long-horizon KV-cache growth, while JIT reduces initial prompt size and Time-To-First-Token.

JIT also introduces an additional tool-selection constraint. Across the evaluations, Combined generally shifts toward higher precision and lower recall relative to Memory-Efficient. If the required tool is not selected in the first stage, its full schema is unavailable during argument generation, making subsequent recovery or pivoting more difficult. We therefore view *Memory-Efficient* as the primary interaction-memory configuration, while *Combined* represents a more aggressive deployment operating point when the static tool bank itself is also constrained. The latter is particularly helpful with a large number of tools that could not be otherwise supported, and low-latency requirement settings. This separation allows the two mechanisms to be applied independently. CSO-based memory can reduce growing interaction context even when the complete compact tool bank is available, while JIT can be introduced when the cost of exposing that tool bank justifies the additional selection bottleneck.

7 Conclusion, Limitations, and Future Work

We introduce a context-management framework for persistent on-device agents that represents interaction history as compact future-use task state. A learned State-Tracker maintains this state through append-only CSO updates, allowing semantic compression while preserving reusable KV-cache prefixes. A complementary JIT interface reduces the static tool context required at initialization. Across evaluations, CSO-based memory remains competitive with or exceeds full-history execution while outperforming the evaluated context-compression baselines, and reduces interaction-context growth by approximately 10–$25\times$.

Limitations. Our approach requires supervised fine-tuning and therefore depends on the quality and coverage of State-Tracker supervision. State tracking also adds a 1.31 s post-turn inference pass in our deployment. JIT additionally introduces a tool-selection bottleneck that can reduce the agent's ability to pivot to new approaches. Finally, broader evaluation across models and open-domain agent settings can further establish a wider set of use cases.

Future Work. Future work could learn memory policies directly from long-horizon task utility, incorporate verification or uncertainty before discarding context, and develop mechanisms for correcting, consolidating, or expiring persistent state without sacrificing incremental KV-cache reuse.

AI use statement

In this work, we used generative AI for synthetic data generation, code editing and creation, and for editing the written draft. We did not use generative AI tools for research ideation, experimental

design, etc. All such work was conducted by the authors. All AI-assisted content was reviewed the authors before inclusion. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We document the information needed to reproduce our experiments throughout the main text and appendix:

- **Architecture and training.** The CSO, dual-adapter architecture, JIT schema loading, and training procedure are described in Section 3 and Section 3.4. Exact model, quantization, LoRA, fine-tuning, compute, and training-data configurations are reported in Appendix A.2.
- **Data generation and supervision.** Appendix A.6 provides the complete State-Tracker supervision prompt, Appendix A.11 reproduces the scenario-generation prompts, and Appendix A.7 gives representative evaluation scenarios.
- **Evaluation.** Sections 4.1–4.3 define the evaluation settings, baselines, and metrics. Full task composition, simulator settings, ToolAlpaca evaluation, and backbone-transfer details are given in Appendix A.3; baseline implementations in Appendix A.4; and metric and device-measurement definitions in Appendix A.5. The LLM-as-judge evaluation and prompt are provided in Appendices A.8 and A.9.
- **Systems implementation and analysis.** Appendix A.12 details dual-adapter KV-cache management, Appendix A.13 specifies deterministic schema minification and JIT loading, Appendix A.14 provides the qualitative error-recovery analysis, and Appendix A.15 reports the KV-cache memory-scaling setup.

REFERENCES

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL https://arxiv.org/abs/2404.14219.

Wei Chen, Zhiyuan Li, and Mingyuan Ma. Octopus: On-device language model for function calling of software apis, 2024. URL https://arxiv.org/abs/2404.01549.

Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Tinyagent: Function calling at the edge, 2024. URL https://arxiv.org/abs/2409.00608.

Milen Ferev, Yishuang Pang, and Pulkit Bhuwalka. On-device llm-driven application function calling for natural language interaction. 2024.

Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. Not all heads matter: A head-level kv cache compression method with integrated retrieval and reasoning, 2025. URL https://arxiv.org/abs/2410.19258.

Georgi Gerganov and The llama.cpp contributors. llama.cpp: Inference of llama model in pure c/c++. https://github.com/ggerganov/llama.cpp, 2023. GitHub repository.

Google Cloud. Gemini 2.0 Flash and 2.0 Pro update. https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash, 2024. Accessed: 2024-10-27.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

OpenAPI Initiative. OpenAPI Specification, 2024. URL https://github.com/OAI/OpenAPI-Specification.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay, 2025. URL https://arxiv.org/abs/2504.03601.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases, 2023. URL https://arxiv.org/abs/2306.05301.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein

Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Qingyue Wang, Yanhe Fu, Yanan Cao, Shuai Wang, Zhiliang Tian, and Liang Ding. Recursively summarizing enables long-term dialogue memory in large language models, 2025. URL https://arxiv.org/abs/2308.15022.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024. URL https://arxiv.org/abs/2309.17453.

Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. On-device language models: A comprehensive review, 2024. URL https://arxiv.org/abs/2409.00088.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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 Awalgaonkar, Rithesh R N, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. xLAM: A family of large action models to empower AI agent systems. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *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)*, pp. 11583–11597, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.578. URL https://aclanthology.org/2025.naacl-long.578/.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023. URL https://arxiv.org/abs/2306.14048.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents, 2025. URL https://arxiv.org/abs/2506.15841.

A Additional Experimental Details

This appendix provides additional implementation, training, evaluation, and qualitative-analysis details for our context-efficient agent framework. We include representative tool definitions, model and

training specifications, evaluation protocols, baseline details, the complete State-Tracker supervision prompt, scenario-generation prompts, a step-by-step KV-cache walkthrough, schema-minification examples, and qualitative trajectory analyses.

A.1 Example On-Device Tools

Our on-device environment contains tools spanning device control, application interaction, local information access, and cloud delegation. Each tool is specified by a name, description, and structured function schema. We show a representative subset below.

Example: Messaging Tool

Description: Sends and manages text messages or chats via installed messaging applications.
Function: `manage_messages(action, recipient, content, app_preference (optional), ...)`

Example: Gallery & Photo Tool

Description: Manages photos and videos in the device gallery, including search, viewing, sharing, and lightweight edits.
Function: `manage_gallery_items(action, search_keywords (optional), item_identifier (optional), ...)`

Example: Navigation Tool

Description: Provides directions and finds nearby points of interest using locally available map information.
Function: `get_local_navigation_info(request_type, destination_or_poi_type, ...)`

Example: Ride Service Launcher

Description: Launches a supported ride-service application with the requested destination pre-filled.
Function: `launch_ride_service_app(destination, origin_address, ...)`

Example: Cloud Delegation Tool

Description: Delegates requests requiring capabilities beyond the available on-device tools, including complex reasoning, generation, or broad external knowledge.
Function: `process_with_cloud_llm(user_query_context, reason_for_escalation)`

A.2 Training Specifications

A.2.1 Base Models and Deployment

Our primary experiments use xLAM-2 3B (Zhang et al., 2025), selected for its function-calling capabilities at a scale suitable for on-device deployment. We additionally evaluate backbone transfer using Qwen3 1.7B (Yang et al., 2025).

All parameter-efficient fine-tuning is performed using LlamaFactory (Zheng et al., 2024). The primary xLAM-2 experiments are trained on a single NVIDIA A6000 GPU with 48 GB of VRAM.

For smartphone deployment, the xLAM-2 model is quantized using the `Q4_K_M` 4-bit quantization scheme in `llama.cpp` (Gerganov & llama.cpp contributors, 2023).

A.2.2 LoRA Hyperparameters

We use Low-Rank Adaptation (LoRA) for parameter-efficient fine-tuning. For the xLAM-2 experiments, the same LoRA configuration is used for the Executor, State-Tracker, Baseline, and Tool-

Efficient adapters where applicable. Each xLAM-2 adapter contains approximately 119M trainable parameters.

Table 5: LoRA hyperparameters used for the primary xLAM-2 experiments.

Hyperparameter	**Value**
LoRA rank	64
LoRA alpha	128
LoRA dropout	0.05
Target modules	All linear layers
Trainable parameters per adapter	$\sim$119M

A.2.3 Fine-Tuning Hyperparameters

Models are fine-tuned for a single epoch. Depending on the training objective, the corresponding dataset contains approximately 150K–200K examples. We use AdamW with a cosine learning-rate schedule.

Table 6: Primary fine-tuning hyperparameters.

Hyperparameter	**Value**
Epochs	1
Learning rate	1×10^{-4}
Optimizer	AdamW
Learning-rate scheduler	Cosine
Warmup ratio	0.1
Effective batch size	32
Maximum sequence length	4096

A complete fine-tuning run for the primary 3B model requires approximately 4.5 hours on a single A6000 GPU.

A.2.4 Training Data Composition

The training distribution contains five task categories designed to cover both simple and stateful agent interactions:

- **Multi-Tool Tasks:** 40%
- **On-Device then Cloud Tasks:** 20%
- **Conversational Tasks:** 10%
- **Single-Tool Tasks:** 15%
- **Cloud-Only Complex Tasks:** 15%

Multi-step interactions receive the largest share because they most directly exercise persistent task state. All agent training uses our deterministic token-efficient tool-schema format.

A.3 Evaluation Protocol

A.3.1 Primary Evaluation Set

Our primary evaluation consists of 406 multi-turn tasks distributed as follows:

- **Multi-Tool Tasks:** 181
- **Cloud Delegation Tasks:** 75
- **On-Device+Cloud Tasks:** 100

- **Conversational Tasks:** 50

The evaluation follows the same broad scenario-generation framework as the training data but uses 19 entirely held-out tool definitions that do not appear during fine-tuning. We therefore characterize this setting as held-out-tool generalization rather than a fully independent task distribution.

Each model is evaluated interactively. This is necessary because the compared methods construct different contexts as a trajectory develops. Full-history execution retains previous turns directly; KV-compression methods operate on representations of the original sequence; replacement summarization maintains a rewritten semantic state; and our Memory-Efficient model incrementally constructs the CSO.

A.3.2 Simulation Settings

Gemini 2.0 Flash is used in the synthetic training-data generation pipeline and in our original interactive evaluation. We additionally evaluate the principal configurations using Qwen3.5-122B as a second interaction simulator. Qwen3.5-122B is not used anywhere in the training-data generation pipeline and therefore provides a different interaction distribution at evaluation time.

The two simulator evaluations are reported separately because simulator behavior changes the trajectories encountered by the agent. We therefore compare methods primarily within the same simulator rather than interpreting absolute scores across simulator settings.

A.3.3 ToolAlpaca

We additionally evaluate on ToolAlpaca to test transfer beyond our generated on-device tool distribution. ToolAlpaca provides independently constructed tools and task scenarios, and therefore introduces changes in both API structure and task distribution.

We adapt the ToolAlpaca interactions to the same interactive execution interface used by our agents while preserving the benchmark's tool semantics and expected actions. The evaluation contains airline and retail scenarios. Unlike the primary held-out-tool setting, no ToolAlpaca tools are used in our agent training-data generation pipeline.

A.3.4 Backbone Transfer

The main experiments use xLAM-2 3B. To determine whether the context-management interface depends on this particular function-calling model, we additionally train Baseline, Memory-Efficient, and Combined configurations using Qwen3 1.7B. This experiment is intended as a backbone-transfer test rather than a complete rerun of every compression baseline.

A.4 Baseline Implementation Details

We compare methods representing three broad approaches to managing interaction context: retaining the original history, compressing representations from the original token sequence, and replacing the history with a semantic memory.

Baseline (FT). The fine-tuned full-history baseline uses the same agent training distribution as our proposed models and receives the complete accumulated interaction history. It uses the deterministic compact schema representation but performs no interaction-history compression.

StreamingLLM. StreamingLLM (Xiao et al., 2024) restricts the retained sequence while preserving attention-sink tokens and recent context. We include it in the Gemini evaluation as a training-free token-retention baseline.

Attention Eviction. Our attention-eviction baseline follows the heavy-hitter principle of H_2O (Zhang et al., 2023), retaining tokens with high accumulated attention importance while evicting lower-importance representations under a constrained cache budget.

HeadKV-R2. HeadKV-R2 is our primary stronger KV-compression comparison. It uses head-specific signals associated with contextual retrieval and reasoning to allocate the available KV budget. We use a KV-cache budget of 3,000 tokens in our experiments. Unlike the CSO, HeadKV-R2 continues to operate on representations derived from the original sequence rather than constructing an explicit semantic task-state representation.

Recursive Summarization. Recursive Summarization periodically converts accumulated interaction history into a compact textual summary without training a task-specific memory mechanism. It is included in the Gemini evaluation as a training-free semantic compression baseline.

Trained Summarization. The Trained Summarization baseline provides a closely matched learned replacement-memory comparison. It uses the same base model, training interactions, teacher supervision, and overall training protocol as the State-Tracker. The difference is the memory-update target: rather than generating only the incremental state delta Δ_t, the model regenerates the complete compressed state after each interaction, replacing the previous memory. This comparison tests whether learned semantic compression alone can account for the performance of the CSO-based memory architecture.

Our Variants. **Tool-Efficient** applies deterministic schema minification and JIT schema loading while retaining raw interaction history. **Memory-Efficient** uses the State-Tracker/Executor architecture and append-only CSO while making the compact full tool bank available. **Combined** applies both interaction-memory compression and JIT schema management.

A.5 Evaluation Metrics

Tool-Call Metrics. For each trajectory, generated tool use is compared against the expected tool behavior using set-based true positives, false positives, and false negatives. We compute precision, recall, and F1 at the trajectory level and macro-average the resulting values over evaluation examples.

Because precision, recall, and F1 are averaged independently across examples, the reported aggregate F1 need not equal the harmonic mean obtained by applying the F1 formula directly to the displayed aggregate precision and recall.

For each task category, we report:

$$P = \frac{TP}{TP + FP}, \qquad R = \frac{TP}{TP + FN}, \qquad F_1 = \frac{2PR}{P + R},$$

with the corresponding per-example scores subsequently macro-averaged.

Context Measurements. For every assistant step, we record the model-visible input context. We report initial tool-context size separately from interaction-context growth because the two correspond to different deployment costs. JIT schema loading primarily changes the former, while CSO compression primarily changes the latter.

Device Measurements. On-device measurements are obtained using the xLAM-2 3B model quantized with `Q4_K_M` and executed using `llama.cpp` on a Samsung Galaxy S25 CPU. We report KV-cache memory, initial prompt latency, and State-Tracker update latency. Latency measurements are averaged over 10 trials.

A.6 State-Tracker Supervision Prompt

The following is the teacher instruction used with Gemini 2.0 Flash to generate State-Tracker supervision. At each completed interaction, the teacher receives the current CSO and newly observed user, assistant, and tool messages. During supervision generation, it additionally receives the task goal and complete generated trajectory as reference context, allowing information that becomes useful only in later steps to be identified. The teacher outputs only the new lines to append to the CSO.

The prompt specifies this incremental update protocol and includes demonstrations of the desired memory behavior.

State-Tracker Generation Prompt

You are an expert AI assistant that analyzes new conversation turns and generates a concise, append-only log entry. Your output MUST ONLY be the new lines to be appended to a checklist. DO NOT repeat any information already present in the "Full Checklist So Far". You are responsible for maintaining a summary of the conversation so far such that all important details are captured concisely. We do not want to degrade performance, but we want to be token efficient.

Your Rules:

- Be Extremely Concise: Use single words for keys (e.g., `user_goal`).
- Log Core Facts: If new, add the user's `user_goal` or any `agent_refusals`.
- Log Progress: If a step is completed, add it under `completed_steps` but do not make it repetitive.
- Log persistent, unresolved tool errors concisely to prevent the agent from making the same mistake again.
- You can format the new lines to append as you want (with or without keys) with the overall goal that the new lines capture the most information in the most concise manner.
- Our primary goal is creating a very concise summary but preserving all the key details of the conversation so no other messages are required for reference.
- You can create some new fields (do NOT overdo it) in the checklist such that crucial information to the task is kept track of.
- You DO NOT need to add the summarized user or tool message, just longer-horizon information about the task that will help in later turns.
- DO NOT output the full checklist. Only output the new lines.
- If there are no new lines you wish to append, output this text without anything else: `# NO_UPDATE`
- Follow these examples perfectly:

Example 1: Generating the first checklist entries
Full Checklist So Far:

```
# This is the start of the conversation.
```

New Messages:
user: Predict the stock market prices for Google tomorrow.
assistant: I can't provide specific stock market predictions. However, I can search for information related to Google's stock and market trends.
New Lines to Append:

```
user_goal: get analyst predictions for google stock
agent_refusals: cannot predict stock prices
```

Example 2: Appending a completed step
Full Checklist So Far:

```
user_goal: enable power saving mode
```

New Messages:
assistant: {"name":"manage_network_settings","arguments":{"action":"toggle_wifi"}}
tool: {"status":"success","action":"toggle_wifi","new_state":"disabled"}
New Lines to Append:

```
completed_steps: wifi disabled
```

Now, perform the task for the following input:

```
[Task Goal]
```
{*The scenario-level task goal is injected here*}
```
[Full Generated Trajectory for Reference]
```
{*The complete generated trajectory is injected here*}
```
[Full Checklist So Far]
```
{*The full text of the current CSO is injected here*}
```
[New Messages]
```
{*The text of the new user, tool, and assistant turns is injected here*}
```
[New Lines to Append]
```

A.7 Example Evaluation Scenarios

We provide representative examples from the task categories used in the primary evaluation. Each example includes the underlying goal, the initial simulated-user utterance, and scenario-level information used to guide the interactive evaluation.

Representative Evaluation Scenarios

Multi-Tool
Goal: Play a specific song and then adjust the volume.
Initial Utterance: "Play 'Bohemian Rhapsody' and turn it up."
Scenario Notes: Requires playing a song and subsequently adjusting the volume. The model must resolve "turn it up" using the preceding action.

Goal: Find a nearby coffee shop and send the location to a friend.
Initial Utterance: "Coffee?"
Scenario Notes: Requires resolving an ambiguous request, retrieving a location, and transferring that result into a subsequent messaging action.

On-Device + Cloud
Goal: Set a timer and then obtain recipe suggestions that fit within that time.
Initial Utterance: "Set a timer for 20 minutes."
Scenario Notes: The local timer action establishes a duration that must subsequently be transferred to the cloud-processing agent.

Goal: Identify a song playing nearby and find similar music.
Initial Utterance: "What song is playing right now?"
Scenario Notes: An on-device audio capability identifies the current song; the returned title is subsequently required by the cloud-processing agent.

Cloud Delegation
Goal: Solve a mathematical problem requiring capabilities outside the local tool set.
Initial Utterance: "Can you help me solve this differential equation: $dy/dx + 2xy = x$?"
Scenario Notes: The request should be delegated rather than mapped to an unrelated local tool.

Goal: Generate a recipe from a set of ingredients.
Initial Utterance: "I have chicken breast, broccoli, rice, soy sauce, and ginger. Can you suggest a creative recipe I can make?"
Scenario Notes: The request requires open-ended generation rather than a supported on-device action.

Conversational
Goal: The user shares a frustrating experience and seeks a conversational response.
Initial Utterance: "My internet has been down all day! It's so annoying."
Scenario Notes: Tests whether the model avoids unnecessary tool use when the user's intent is conversational.

Goal: Engage in an abstract discussion.
Initial Utterance: "Do you think there's such a thing as true free will?"
Scenario Notes: Tests non-tool conversational behavior.

A.8 Qualitative Interaction Evaluation

In addition to rule-based tool-execution metrics, the Gemini evaluation includes an LLM-as-judge score for overall interaction quality. We treat this signal as complementary rather than primary because it depends on model-based evaluation and combines several aspects of trajectory quality.

A.9 LLM-as-Judge Prompt

The following prompt is used for the complementary trajectory-quality evaluation.

Method	Cloud	Multi-Tool	On-Device+Cloud	Conversational
xLAM-2 3B	2.07	3.59	2.27	2.18
Baseline (FT)	3.79	3.17	3.45	2.87
Recursive Summarization	3.41	2.26	1.96	3.23
StreamingLLM	4.00	2.78	2.02	3.59
Attention Eviction	4.18	2.88	2.16	3.70
Tool-Efficient	3.55	3.61	3.32	3.21
Memory-Efficient	4.11	3.37	3.64	3.32
Combined	2.68	3.32	3.00	3.80

Table 7: LLM-as-judge assistant-quality scores under the Gemini simulator. Scores range from 1 to 5, with higher values indicating better judged interaction quality.

LLM-as-Judge Evaluation Prompt

System Prompt
You are an expert AI evaluator. Your task is to meticulously review a generated conversational trajectory, focusing specifically on the assistant's performance. Your focus is on QUALITATIVE aspects of the ASSISTANT: naturalness, logical flow, contextual appropriateness of tool calls and their arguments, and overall goal achievement.
You will be given:

1. The original User Intent Description & Scenario Notes.
2. The Expected Tool Sequence (as abstract tool IDs).
3. The actual Generated Conversational Trajectory (as a JSON string).
4. A list of tools the assistant was aware of during generation.

Evaluation Criteria for the Assistant:

- **Naturalness & Realism:** Are the assistant's responses natural and helpful?
- **Contextual Appropriateness of Tool Calls:** Did the assistant choose the right tool at the right time?
- **Argument Sensibility:** Are argument values sensible and correctly derived from the conversation context?
- **Clarification Handling:** Did the assistant ask for clarification when needed before calling a tool?
- **Adherence to Intent:** Did the assistant make reasonable efforts to address the user's intent?
- **Information Grounding:** Did the assistant hallucinate information?

Binary Classification
`classification`: 1 if the entire trajectory is high quality and coherent; 0 if there are significant issues.
Assistant Quality Score
Rate the assistant's performance on a scale from 1 to 5 (1 = Poor, 5 = Excellent).
Output Format

```
{
  "classification": 0 or 1,
  "assistant_quality_score": an integer from 1 to 5,
  "feedback": "Detailed feedback and reasoning."
}
```

Trajectory Evaluation Input

Please evaluate the following trajectory based on the qualitative criteria and scoring guidelines provided in the system prompt:
1. User Intent Description:
{*The original scenario description is injected here*}
2. Scenario Notes:
{*The scenario notes are injected here, or N/A*}
3. Expected Tool Sequence:
{*The intended tool sequence is injected here*}
4. Generated Conversational Trajectory:
{*The complete generated trajectory is injected here*}

5. Tools Available to the Assistant:
{*A formatted list of available tools is injected here*}
Now provide your evaluation in the specified JSON format.

A.10 Qualitative Analysis of Simulated Interactions

We qualitatively inspect generated trajectories to understand the behavior of the simulated user and the failure modes elicited during evaluation.

Goal and Persona Adherence. The simulator generally maintains the assigned high-level goal while producing responses consistent with the supplied user persona. For example, scenarios specifying frustration tend to produce impatient or corrective follow-ups, whereas exploratory personas are more likely to provide additional information or ask follow-up questions.

Ambiguity and Clarification. Several scenarios intentionally withhold information required for immediate execution. These interactions test whether the agent clarifies, retrieves missing information through an appropriate tool, or incorrectly commits to an action. Conversational scenarios similarly test whether the agent can distinguish between an actionable request and an utterance requiring only a natural-language response.

Error-Recovery Behavior. Longer interactions also expose repeated-action failure modes. In some trajectories, a tool invocation fails and the agent subsequently repeats the same or closely related invalid action despite receiving explicit feedback. These trajectories provide a useful setting for evaluating whether compressed persistent state preserves failure information across later steps.

Qualitative Failure Patterns Across Context Strategies. Inspection of evaluation trajectories reveals two recurring types of context failure. In token-level compression trajectories, we observe cases where exact tool-returned values, identifiers, or earlier tool feedback needed by a later action are no longer available after compression. In full-history trajectories, the relevant information remains present in the context, but we observe cases where later actions do not make effective use of it, including repeated invalid tool calls despite earlier corrective feedback. We interpret the latter pattern as consistent with attention dilution as interaction history grows, but these examples do not isolate attention dilution as a causal mechanism. The examples below illustrate how explicitly promoting persistent error information into the CSO can make such feedback directly available to later execution.

Overall, qualitative inspection suggests that the simulator provides useful goal-directed interactions spanning clarification, tool use, and error recovery. As with any LLM-based simulator, however, the resulting interaction distribution may contain model-specific artifacts and is not a substitute for evaluation with real users. We therefore complement this setting with a second simulator and ToolAlpaca in the main experiments.

A.11 Scenario-Generation Prompts

For reproducibility, we reproduce the prompts used to construct the principal training and evaluation scenario families. The text below reflects the prompt templates used in the data-generation pipeline.

A.11.1 Complex Multi-Step Scenarios

Prompt: Complex Multi-Step Scenarios

You are an expert task designer creating realistic, complex user scenarios for a new AI assistant. Generate {*num_scenarios*} scenarios that require a sequence of **3 to 5 different tools** to fully resolve. These scenarios should involve planning, research, and execution.
Available Tools (for reference):
{*A manifest of up to 40 randomly sampled, family-agnostic tools is injected here.*}
Cloud Agent (for complex reasoning):
{`cloud_processing_agent_id`}

For each scenario, provide:

1. `scenario_description`: a high-level story of the user's complex goal.
2. `user_persona`: the user's personality and context.
3. `initial_user_utterance`: the exact first user message.
4. `intended_tool_sequence`: 3–5 Tool IDs representing the ideal path.
5. `constraints_and_context`: important implicit information.
6. `scenario_notes`: why the scenario requires a long sequence.

Example:

```
{
  "scenario_description":
    "User wants to find a good time to meet their friend Alex
     for dinner this week, suggest a restaurant, book a table,
     and get directions.",
  "user_persona":
    "Social planner, trying to organize a get-together.",
  "initial_user_utterance":
    "Find a time Alex and I are both free for dinner this week,
     suggest a good Italian place, and book it for us.",
  "intended_tool_sequence":
    ["agent_007", "agent_021", "agent_015", "agent_011"],
  "constraints_and_context":
    {"contact_name": "Alex", "cuisine_preference": "Italian"}
}
```

A.11.2 Constraint-Based Single-Tool Scenarios

Prompt: Constraint-Based Single-Tool Scenarios

You are an expert task designer. Generate {*num_scenarios*} scenarios where the user's request targets a single on-device tool, but includes complex constraints, preferences, or conditions that the AI must understand and respect.
Available Tools:
{*A manifest of available on-device tools is injected here.*}
For each scenario, provide:

1. `scenario_description`
2. `user_persona`
3. `initial_user_utterance`
4. `intended_tool_sequence`: a single Tool ID
5. `constraints_and_context`
6. `scenario_notes`

Example:

```
{
  "scenario_description":
    "User wants to schedule a meeting while avoiding a
     particular day and time range.",
  "user_persona":
    "Organized project manager, direct and specific.",
  "initial_user_utterance":
    "Book a 1-hour 'Q3 Planning' meeting with Dana,
     but make sure it's not on Friday and not before 10 AM.",
  "intended_tool_sequence": ["agent_007"],
  "constraints_and_context":
    {"avoid_day": "Friday", "earliest_time": "10:00AM"}
}
```

A.11.3 On-Device then Cloud Scenarios

Prompt: On-Device then Cloud Scenarios

Generate {*num_scenarios*} user scenarios. Each scenario MUST require a sequence of EXACTLY 2 tools:

1. First, an ON-DEVICE tool performs an action or gathers local information.
2. Second, the CLOUD PROCESSING AGENT ({`cloud_agent_id`}) uses the result from the first step for more complex processing.

Available On-Device Tools:
{*A manifest of available on-device tools is injected here.*}
For each scenario, provide:

- `scenario_description`
- `initial_user_utterance`
- `intended_tool_sequence`
- `scenario_notes`

Example:

```
{
  "scenario_description":
    "User took a screenshot of an error and wants the
     cloud agent to explain it.",
  "initial_user_utterance":
    "Ugh, what does this error mean? *User gestures
     to a screenshot*",
  "intended_tool_sequence": ["agent_024", "agent_026"],
  "scenario_notes":
    "Screenshot content from the local tool is passed
     to the cloud agent for analysis."
}
```

A.11.4 Purely Conversational Scenarios

Prompt: Purely Conversational Scenarios

Generate {*num_scenarios*} purely conversational scenarios. These should simulate natural, open-ended chitchat where the user's goal is social interaction rather than task completion. Focus on seeking opinions, sharing feelings, or friendly banter.
For each scenario, provide:

- `scenario_description`
- `user_persona`
- `initial_user_utterance`
- `intended_tool_sequence`: `[""]` or `["CONVERSATIONAL_ROUTER"]`
- `scenario_notes`

Example:

```
{
  "scenario_description":
    "User wants to discuss movies with the assistant
     and get its opinion.",
  "user_persona":
    "Casual movie fan, looking for a discussion.",
  "initial_user_utterance":
    "Hey, in your opinion, what's a film that everyone
     should see at least once?",
  "intended_tool_sequence": [""]
}
```

A.11.5 CLOUD-ONLY COMPLEX SCENARIOS

Prompt: Cloud-Only Complex Scenarios

You are a user scenario designer. Generate user scenarios that CLEARLY require the advanced capabilities of the CLOUD PROCESSING AGENT ({`cloud_processing_agent_id`}). These tasks involve complex reasoning, math problems, coding, creative text generation, or synthesizing broad external knowledge.
Simpler on-device tools exist but should be insufficient for these tasks.
For each scenario, provide:

- `scenario_description`
- `user_persona`
- `initial_user_utterance`
- `intended_tool_sequence`: only the cloud-processing tool
- `constraints_and_context` (optional)
- `expected_outcome` (optional)
- `scenario_notes`

Example:

```
{
  "scenario_description":
    "User asks their phone's AI to write a Python script
     to scrape website titles from a list of URLs.",
  "user_persona":
    "Developer looking for a quick utility.",
  "initial_user_utterance":
    "Can you write a Python script to take a list of URLs
     and grab the title tag from each one?",
  "intended_tool_sequence":
    ["cloud_processing_agent"],
  "constraints_and_context":
    {"task_type": "code_generation", "language": "Python"}
}
```

A.12 DUAL-ADAPTER KV-CACHE MANAGEMENT

We now illustrate how the append-only CSO interacts with KV-cache management. This walkthrough corresponds to the **Memory-Efficient** configuration, where the Executor is initialized with the compact full tool bank. Combined uses the same CSO mechanism but additionally changes which tool schemas are visible through JIT loading.

The implementation separates model-visible state into two components: *permanent context*, consisting of the static prefix and accumulated CSO, and *ephemeral context*, consisting of the current user/tool interaction. Separate caches are maintained for the Executor and State-Tracker.

Dual-Adapter KV-Cache Walkthrough

Turn 0: Initialization
Both adapters are initialized with their respective system prompts and the initial CSO:

```
# This is the start of the conversation.
```

The Executor's permanent cache contains its system prompt, compact tool definitions, and initial CSO. The State-Tracker cache contains its own instruction prefix and the same current CSO. In one representative run, these prefixes contain approximately 1710 and 206 tokens, respectively.

Turn 1: User Creates an IT Ticket
User: "My Wi-Fi is not working, please create an IT ticket."
The user query is processed on top of the Executor's committed persistent prefix but is not itself committed to persistent state. The Executor then generates a tool invocation such as:

```
<tool_call>
{
```

```
  "name": "manage_it_support_ticket",
  ...
}
</tool_call>
```

The conversational tokens involved in this action are ephemeral: they are needed to produce the current action but are not automatically committed to the persistent context.
The State-Tracker receives the new interaction and produces an incremental update:

```
user_goal: create_it_ticket
ticket_details:
  - issue: wifi_not_working
```

This update is appended to the persistent CSO. After the tool executes, suppose it returns:

```
{
  "status": "success",
  "ticket_id": "IT7390"
}
```

The corresponding identifier is subsequently retained by the State-Tracker:

```
ticket_id: IT7390
```

The raw tool observation itself need not remain in the persistent conversational history once the relevant state has been recorded.

Turn 2: User Refers to Earlier State
User: "What's the status of that ticket?"
Before processing the new turn, the implementation rewinds the caches to their committed permanent positions, discarding ephemeral conversational tokens from the preceding turn. The new user request is then processed on top of the unchanged persistent prefix containing:

```
user_goal: create_it_ticket
ticket_details:
  - issue: wifi_not_working
ticket_id: IT7390
```

The Executor can therefore produce:

```
<tool_call>
{
  "name": "manage_it_support_ticket",
  "arguments": {
    "action": "check_ticket_status",
    "ticket_id": "IT7390"
  }
}
</tool_call>
```

The example illustrates two properties of the CSO: an exact identifier needed by a later action remains explicitly available after the corresponding raw interaction is discarded, while previously committed state remains unchanged as new information is appended.

A.13 Deterministic Tool-Schema Minification

Schema minification is deterministic preprocessing rather than a learned component. It removes formatting and representational overhead while retaining the fields used by our function-calling interface, including tool names, descriptions, parameter definitions, and required arguments. The model is fine-tuned directly on the resulting compact representation. Table 8 gives a representative example.

JIT schema loading applies an additional reduction. At initialization, the Executor receives only tool names and short descriptions. Once a tool is selected, its full compact schema is introduced for argument generation. This reduces the static initial tool context but creates an additional selection decision, producing the precision–recall trade-off discussed in Section 6.

Table 8: Representative standard OpenAPI schema and its deterministic compact representation. The example decreases from 121 to 72 tokens.

Standard OpenAPI (121 tokens)

```
{
  "type": "function",
  "function": {
    "name": "set_timer",
    "description":
      "Sets a timer for a
       specified duration.",
    "parameters": {
      "type": "object",
      "properties": {
        "duration_seconds": {
          "type": "integer",
          "description":
            "The duration of the
             timer in seconds."
        },
        "timer_name": {
          "type": "string",
          "description":
            "An optional name
             for the timer."
        }
      },
      "required":
        ["duration_seconds"]
    }
  }
}
```

Compact representation (72 tokens)

```
{"type":"function",
"function":{
"name":"set_timer",
"description":"Sets a timer
for a specified duration.",
"parameters":{
"type":"object",
"properties":{
"duration_seconds":{
"type":"integer",
"description":"The duration
of the timer in seconds."},
"timer_name":{
"type":"string",
"description":"An optional
name for the timer."}},
"required":
["duration_seconds"]}}}
```

A.14 Qualitative Examples of Error Persistence and Recovery

We next provide an illustrative example of how persistent error information is represented and subsequently used by the CSO. The Memory-Efficient sequence below is drawn directly from an evaluation trajectory. The Baseline sequence is a composite constructed from a recurring failure pattern observed across baseline trajectories and is included only to illustrate that pattern; it is not a matched quantitative comparison.

Error Persistence and Recovery

User Goal: Find all `.txt` files in `/documents/projects` containing the word `configuration`.

Representative Baseline Failure Pattern

The agent selects `manage_device_files` but attempts to express content search using a parameter that is not valid for the selected operation:

```
<tool_call>{
  "name":"manage_device_files",
  "arguments":{
    "operation":"search_files",
    "path_or_search_query":"/documents/projects",
    "file_type_filter_for_search":"txt",
    "new_name_or_destination_path":"configuration"
  }
}</tool_call>
```

The tool returns:

```
{
  "error":
    "Invalid request. The
     'new_name_or_destination_path'
     parameter is not applicable for
     'search_files'...",
  "success": false
}
```

A recurring failure pattern in baseline trajectories is to subsequently repeat the same invalid invocation despite the preceding tool feedback or user correction.

Observed Memory-Efficient Recovery
In the Memory-Efficient trajectory, an invalid invocation produces the same type of tool feedback. The State-Tracker subsequently records:

```
tool_error:
  invalid parameter
  new_name_or_destination_path
  for search_files
```

After the user confirms the desired operation, the Executor generates:

```
<tool_call>{
  "name":"manage_device_files",
  "arguments":{
    "operation":"search_files",
    "path_or_search_query":
      "/documents/projects",
    "file_type_filter_for_search":"txt"
  }
}</tool_call>
```

Analysis. The Memory-Efficient trajectory illustrates the intended role of the CSO: tool feedback that may matter to future decisions can be promoted into concise persistent state rather than depending on the complete raw trajectory remaining available. The baseline composite illustrates a qualitatively observed repetition failure. These examples are illustrative and do not constitute a controlled causal comparison between append-only state and full-history execution.

A.15 KV-Cache Memory Scaling

Figure 3 shows the measured KV-cache memory overhead as context length increases for the quantized 3B deployment model. The approximately linear increase makes persistent context a meaningful resource constraint even when the underlying model nominally supports longer context windows.

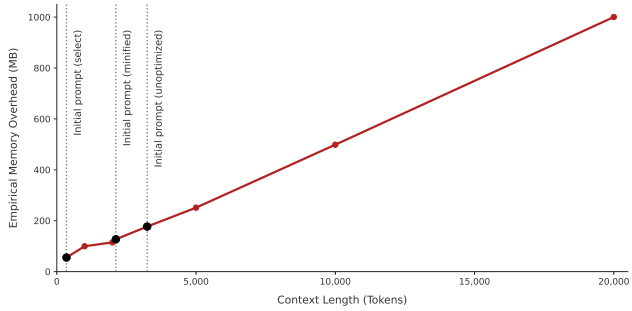


Figure 3: KV-cache memory overhead as a function of context length for the `Q4_K_M`-quantized xLAM-2 3B model under `llama.cpp`. Retaining approximately 10K–20K tokens requires roughly 500–1,000 MB of KV-cache memory in our deployment. CSO-based interaction compression reduces this long-horizon context requirement by replacing the accumulated raw trajectory with compact persistent task state.