GRPO-Trained Memory Policies for Few-Shot Function Calling

Anonymous Authors[1]

Abstract

Modern language-model agents must support a growing catalog of tools, yet every approach to adding a new tool today incurs either large training cost (supervised fine-tuning) or large per-query inference cost (in-context documentation). We propose a framework that trains memory read and write policies with Group Relative Policy Optimization (GRPO) over a frozen language model, disentangling representation learning from memory management. The controller learns when to extract tool descriptions from a handful of demonstrations, what compact structure to write into an external memory store, and which entries to retrieve at inference. The base model receives no gradients during any phase. The framework generalizes retrieval-augmented generation by replacing similarity-based reads and passive writes with learned policies trained from verifiable tool-use reward, and adds an action that records failure modes alongside descriptions. On three few-shot function-calling benchmarks (BFCL-v3, τ-bench, NexusBench) under a strict held-out tool protocol, a 200M-parameter controller over a frozen Llama 3.1 8B model exceeds full supervised fine-tuning of the same backbone at K=5 demonstrations while using 40 times fewer trainable parameters and ten times fewer inference tokens. An ablation shows that updating the base model's weights during training improves in-distribution accuracy but degrades few-shot generalization, providing direct empirical justification for the frozen-backbone commitment.

1. Introduction

Tool use is now a routine deployment mode for language models, with function-calling APIs, Model Context Protocol servers, and agentic frameworks sitting on top of frozen base models. As these deployments grow, the central operational question is no longer how the base model reasons but how it incorporates new tools at deployment time. We refer to this as the *few-shot tool acquisition problem*. Three settings make it concrete. A new internal tool appears that the base model has never seen. A customer-specific tool must be supported without modifying a shared base model. A public ecosystem of tools, such as a server marketplace, exposes capabilities at a rate that no pretraining corpus can track.

Three practical approaches dominate today and each has clear limits. First, in-context tool documentation injects schemas and demonstrations into the prompt, which scales tokens linearly in the catalog size and forces the base model to re-read the same documentation on every query. Second, supervised fine-tuning on demonstration traces requires hundreds of examples per tool and a training run that bakes a particular tool repertoire into the weights. Third, retrieval-augmented generation indexes tool documentation in a vector store with similarity-based retrieval, which learns nothing from interaction and treats every demonstration token as equally important to index.

Memory-augmented neural networks (Santoro et al., 2016) propose a different architectural commitment, in which a learned controller manages an external memory while the underlying representation network remains stable, so that adaptation to a new task occurs through manipulation of the external data structure rather than through changes to model weights. We adopt this commitment for tool acquisition, departing from recent reinforcement-learning approaches (Yu et al., 2025; Bao et al., 2025) that make the language model itself the memory manager and require fine-tuning a multi-billion-parameter backbone. The base model has already learned strong representations and rarely needs updating for a new tool; the hard learning problem is selective memory management, which has small parameter count and benefits from explicit policy structure separate from the base model. Section 2 discusses this prior work in detail.

We propose **MemCtrl**, a framework in which a 200M-parameter controller is trained with reinforcement learning over verifiable tool-use reward while the base language model remains entirely frozen. Our contributions are the following.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. We adapt the disentangled memory control framework to frozen-backbone tool acquisition, with an extended action space that includes failure-mode logging, structural retrieval, and entry consolidation.
2. We formulate the controller training objective as a Group Relative Policy Optimization variant suited to discrete memory operations.
3. We evaluate under a strict held-out tool protocol on three function-calling benchmarks and show that a 200M controller over a frozen Llama 3.1 8B model exceeds full backbone fine-tuning at K=5 while using 40 times fewer trainable parameters and ten times fewer inference tokens.
4. We provide a disentanglement ablation that directly relaxes the frozen-backbone constraint and demonstrates that updating the base model's weights during training erodes few-shot generalization, justifying the frozen-backbone commitment empirically.

2. Related Work

Memory-augmented neural networks. Weston et al. (2015) introduced Memory Networks as an architecture pairing a neural controller with an external memory for language tasks, and Santoro et al. (2016) applied it to few-shot learning, showing that a controller writing to and reading from memory could adapt to new classes from few examples without retraining the representation network. Nearest neighbor language modeling (Khandelwal et al., 2020) augments a frozen language model with a KNN datastore and matches or exceeds parametric fine-tuning, and Mireshghallah et al. (2023) extends this to label-set drift, showing that swapping out the datastore suffices for adaptation. We treat MemCtrl as a learned-policy extension of this paradigm in which both write and read sides are trained from task reward rather than fixed to encoder embeddings and similarity, and in which the action set includes structural operations such as failure logging that have no similarity-based analog.

Memory in LLM agents. In the LLM era, MemGPT (Packer et al., 2023) introduces a virtual-memory metaphor with active context paging, exposing five operations that move information between main context and archival storage, controlled by prompted invocations of the language model. Letta extends this design with persona and tool memory. Both rely on hand-engineered rules or prompted instructions to decide when to invoke each operation, and neither trains the controller from reward.

Mem0 and Memory-R1. Mem0 (Chhikara et al., 2025) formalizes memory management with four operations (`ADD`, `UPDATE`, `DELETE`, `NOOP`) executed by a prompted language model after each conversation turn. The stated goal is persistent personalized memory for conversational agents, evaluated on the LoCoMo multi-session benchmark. The memory manager is not trained; the operations are issued through carefully engineered prompts. Memory-R1 (Yu et al., 2025) adopts the same operator set and adds reinforcement learning of the memory manager using PPO and GRPO, with the full language model receiving gradients. Their reported gains over prompted Mem0 indicate that training the operator policy matters. The architectural commitment differs from ours, however, in that Memory-R1 makes the language model the memory manager, so the same 7B-class parameters that encode the conversation also issue the memory operations.

Mem-α. Bao et al. (2025) extends the Memory-R1 paradigm with a richer toolkit including memory-construction operations, structured updates, and multi-component memory schemas. The motivating goal is length generalization, demonstrated on conversational sequences up to 400K tokens. Like Memory-R1, the language model itself is the memory manager and is fully fine-tuned. Mem-α's richer operator set is the closest existing system to ours in terms of action-space diversity, but it does not include explicit failure-mode logging, does not target tool acquisition, and does not test the disentanglement claim against updating the base model's weights during training.

Tool learning and function calling. Toolformer (Schick et al., 2023) and Gorilla (Patil et al., 2023) initiated learning-based tool use through supervised fine-tuning on synthetic invocation traces. The Berkeley Function Calling Leaderboard (Patil et al., 2024) provides standardized evaluation across single-turn, multi-turn, and relevance-detection categories. τ-bench (Yao et al., 2024) introduces dynamic conversations with a simulated user and policy documents. Recent multi-turn RL approaches (Zheng et al., 2025) apply GRPO to function calling but fine-tune the language model directly rather than separating memory management.

Operation sets across systems. Table 1 contrasts the operation sets exposed by recent memory systems and ours. The novel operations in MemCtrl, namely `WRITE_FAIL`, `READ_BY_SIG`, and `MERGE`, are absent from prior systems. Failure-mode logging in particular has no analog in any system that we are aware of, and our action-space ablation in Table 5 shows it is responsible for a measurable share of MemCtrl's accuracy gains.

3. Preliminaries

3.1. Few-Shot Tool Acquisition

Let $\mathcal{T} = \{t_1, \ldots, t_N\}$ denote a catalog of tools, each specified by a schema σ_i comprising name, argument types, return type, and a natural-language description. The catalog is partitioned into a training set $\mathcal{T}_p$ and a held-out set $\mathcal{T}_h$ with $\mathcal{T}_p \cap \mathcal{T}_h = \emptyset$. For each tool t_i, we have a small set of

Table 1. Operation sets across memory systems. RL Trained indicates whether the memory manager is trained from reward. Fail Log indicates whether the system has an explicit operation for recording failure modes against an entry.

SYSTEM	# OPS	RL TRAINED	FAIL LOG
MEMGPT	5	NO	NO
LETTA	5	NO	NO
MEM0	4	NO	NO
MEMORY-R1	4	YES	NO
MEM-α	7	YES	NO
MEMCTRL	**8**	**YES**	**YES**

demonstrations $D_i = \{(q_j, c_j, e_j)\}_{j=1}^{K}$ where q_j is a user query, c_j is the ground-truth tool call, and e_j is the execution outcome. A disjoint evaluation query set Q_i contains fresh queries that require t_i. The few-shot tool acquisition problem is to produce a system that, from K demonstrations of a held-out tool $t_h \in \mathcal{T}_h$, correctly invokes t_h on queries in Q_h. Accuracy on Q_h as a function of K is the central evaluation curve.

3.2. Group Relative Policy Optimization

Our objective extends GRPO (Shao et al., 2024). Given a prompt q, GRPO samples G responses $\{o_1, \ldots, o_G\}$ from the current policy $\pi_{\theta_{\text{old}}}$, scores them with a verifiable reward $R_i = R(q, o_i)$, and computes group-relative advantages

$$\hat{A}_i = \frac{R_i - \text{mean}(R_{1:G})}{\text{std}(R_{1:G})}. \tag{1}$$

The policy is updated by maximizing

$$\begin{aligned}\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\Big[\min(\rho_i \hat{A}_i, \text{clip}(\rho_i, 1-\epsilon, 1+\epsilon)\hat{A}_i)\Big] \\ - \beta D_{\text{KL}}[\pi_\theta \,\|\, \pi_{\text{ref}}]\end{aligned} \tag{2}$$

with $\rho_i = \pi_\theta(o_i|q)/\pi_{\theta_{\text{old}}}(o_i|q)$.

3.3. Why GRPO and Not PPO

PPO requires a learned value function $V_\psi(s)$ to compute the advantage $\hat{A}_t = R_t - V_\psi(s_t) + \ldots$. In our setting, the state s_t is the joint configuration of the user query, the controller's history, and the current state of the memory store $\mathcal{M}_t$. The memory store is discrete, mutable, and high-dimensional. Its size grows as the controller writes entries during demonstration ingestion, entries are merged or deleted, and the structured fields per entry (affordance, schema, pattern, failures) are themselves variable-length text. A value head conditioned on s_t would need to encode a variable-size structured object that mutates inside the trajectory and yields rewards only after all demonstrations have been ingested and all evaluation queries answered. Training such a value head is poorly defined because the same memory configuration can produce very different rewards depending on the eval queries that follow, and the bootstrap target $V_\psi(s_{t+1})$ is meaningless when memory configurations are not exchangeable across trajectories.

GRPO avoids this by computing advantages over groups of trajectories that share the same initial conditions, eliminating the need to learn a value function over the memory state. Group-relative normalization captures the within-tool variation that PPO would otherwise have to extract from a value bootstrap.

4. Method

4.1. Disentangled Architecture

Figure 1 shows the system. A frozen language model f_θ serves inference, receiving inputs that combine the user query with content retrieved from an external memory store $\mathcal{M}$. A separate controller π_ϕ with 200M parameters observes a fixed-dimensional encoding of the task context together with a summary of $\mathcal{M}$ and emits memory operations. The controller is a transformer encoder, two orders of magnitude smaller than the 8B backbone. The base model receives no gradients during any phase.

The controller is a classifier, not a generator. At each step it makes two discrete picks. It chooses one of the eight operations, and when the operation modifies an existing entry, it also chooses which entry to modify by comparing key embeddings against the current query. Both picks are trained with GRPO using the task reward in Equation (4). The text that ends up in memory is produced by the frozen LLM. When the controller picks `WRITE_NEW`, the frozen LLM is prompted with the demonstration and a fixed template, and it fills the four fields of a new entry. `WRITE_UPDATE` and `WRITE_FAIL` work the same way but with templates that update an existing entry or append a failure note. Read operations call the memory index and need no generation. This means the controller learns only when and where to act. The frozen LLM, under fixed templates, supplies the words. No language-modeling loss is used, and no gradients reach the base model.

4.2. Memory Store and Entry Structure

The memory store is a key-value collection where keys are dense embeddings produced by a frozen sentence encoder and values are structured records produced by the frozen LLM under fixed templates, with the controller deciding when and where each record is written. The store supports retrieval by embedding similarity over keys and by structural match over schema fields. We do not store latent LLM hidden states; values are interpretable text that the LLM consumes through its standard prompt interface.

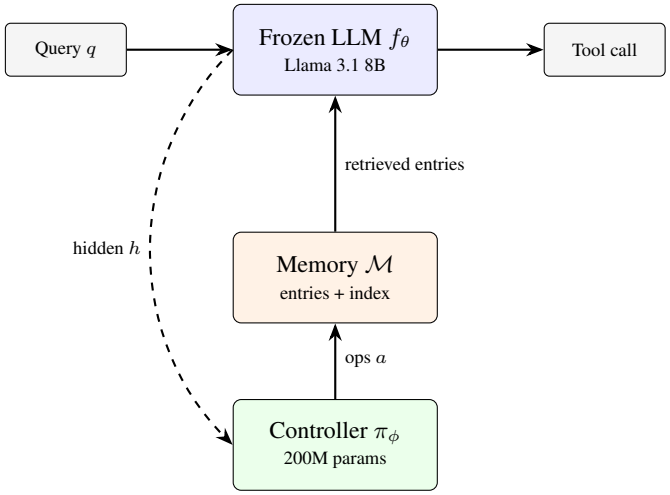

Figure 1. Disentangled architecture. The frozen LLM produces hidden representations on each input; the controller observes those representations together with a memory summary and emits operations on the memory store; the memory store returns structured tool entries that are concatenated with the query before the LLM generates the tool call.

Each entry $m \in \mathcal{M}$ has four fields. The **affordance** field stores the tool description, a short natural-language summary of what the tool does and when it should be invoked. The **schema** field is a compact representation of arguments and their types. The **pattern** field is a template that captures how arguments are derived from a user query. The **failures** field is a list of failure-mode notes produced by the frozen LLM when the controller picks `WRITE_FAIL`, summarizing the error from a failed demonstration. The key is the embedding of the description stored in the affordance field.

Listing 1 shows a representative entry after the controller has ingested five demonstrations of the Stripe `create_payout` API.

Listing 1. A memory entry for `stripe.create_payout` produced by the frozen LLM under controller direction. About 80 tokens.

```
[stripe.create_payout]
affordance: send funds from Stripe balance to
  external bank account
schema: amount int CENTS, currency str ISO-4217
  lowercase, destination str "ba_*",
  description str optional
pattern: amount = dollar_value * 100;
  currency in {usd, eur, gbp}
FAIL: do not pass dollars; multiply by 100
  before passing to amount
```

4.3. Action Space

The controller emits one of eight discrete actions per step. `WRITE_NEW` creates an entry from a demonstration. `WRITE_UPDATE` refines an existing entry with additional pattern data. `WRITE_FAIL` records a failure mode against an entry, populated when a demonstration's execution returned an error. `MERGE` consolidates two near-duplicate entries. `READ_TOP_K` retrieves the top-K entries by affordance similarity. `READ_BY_SIG` retrieves by exact match on the schema signature. `FORGET` removes an entry after persistent failure. `NOOP` performs no operation. A **non-noop operation** is any of the seven actions other than `NOOP`; the count of non-noop operations enters the reward as a cost term.

4.4. Observation Encoding

At each step the controller observes a fixed-dimensional vector o_t assembled from four components,

$$o_t = [\, h_q;\ s_{\mathcal{M}};\ r;\ b \,] \tag{3}$$

where $h_q \in \mathbb{R}^d$ is the pooled hidden state of the current query from the last decoder layer of the frozen LLM, $s_{\mathcal{M}} \in \mathbb{R}^d$ is an attention-pooled summary of memory entries, r is a role embedding distinguishing demonstration ingestion from inference, and b is a budget embedding encoding the remaining read budget and token budget for the current query. The frozen LLM is called only in inference mode for hidden state extraction.

4.5. Reward Design

The reward for a trajectory τ that ingests K demonstrations of a tool and then answers $|Q|$ evaluation queries is

$$R(\tau) = \alpha \cdot \mathrm{succ}(\tau) - \beta_R \cdot \mathrm{tok}(\tau) - \gamma \cdot |\mathrm{ops}(\tau)| \tag{4}$$

where $\mathrm{succ}(\tau)$ is the fraction of evaluation queries whose generated tool call is judged *AST-equivalent* to the ground-truth call, $\mathrm{tok}(\tau)$ is the average input tokens per evaluation query, and $|\mathrm{ops}(\tau)|$ counts non-noop operations across the trajectory. We write β_R to distinguish the reward token-cost coefficient from the GRPO KL coefficient β in Equation (2).

AST equivalence compares two tool calls by parsing them into abstract syntax trees and checking equivalence of the function name and the multiset of named argument bindings, with type-correct equality on argument values. For example, `f(b=2, a=1)` is AST-equivalent to `f(a=1, b=2)` but not to `f(a=1, b="2")`. AST equivalence is the standard scoring rule for BFCL function calling.

The coefficients α, β_R, γ are set so that task success dominates while the cost terms only break ties between equally accurate policies. A one percentage point accuracy gain is worth one hundred fewer input tokens per query or ten fewer memory operations, which prevents the controller from inflating retrieval or write activity for marginal accuracy. The intuition behind this calibration is that the controller should learn to use the memory store economically rather than as

Algorithm 1 MemCtrl controller training with GRPO

Input training tools $\mathcal{T}_p$ with demonstrations and eval queries; frozen LLM f_θ
Initialize controller π_ϕ
for epoch $= 1$ to E **do**
  $\pi_{\text{ref}} \leftarrow \pi_\phi$ (snapshot for KL regularization)
  for each tool $t \in \mathcal{T}_p$ in random order **do**
    for $g = 1$ to G **do**
      Sample demonstrations $D_t^{(g)}$ and queries $Q_t^{(g)}$
      Run controller in **ingestion mode** to populate $\mathcal{M}$ from $D_t^{(g)}$
      Run controller in **inference mode** to answer $Q_t^{(g)}$ via f_θ with retrieved entries
      Compute trajectory reward R_g via Equation (4)
    end for
    Compute group-relative advantages $\hat{A}_{1:G}$ via Equation (1)
    Update ϕ by maximizing $\mathcal{J}_{\text{GRPO}}$ in Equation (2)
  end for
end for
Output trained controller π_ϕ

a free resource, because at deployment the cost of memory operations and retrieved tokens translates directly to inference cost. The token cost in Equation (4) measures input tokens at inference only; write-time calls to the frozen LLM for entry construction are amortized across the eval query set and reported separately in Section 6.

4.6. Training Objective and Procedure

The controller is trained with GRPO over groups of G=8 trajectories that share the same training tool, so the group-relative advantage in Equation (1) captures variation in controller behavior on identical tool conditions. The KL term in Equation (2) is computed against a reference controller π_{ref} taken to be the controller from the previous epoch, which stabilizes training without anchoring to a fixed initial policy. Algorithm 1 states the full procedure.

4.7. Non-Parametric Adaptation at Deployment

Adaptation to any held-out tool occurs without gradient updates anywhere in the system. For each of K demonstrations, the frozen LLM produces a hidden state in a single forward pass, the controller consumes it together with the current memory summary, and emits a write action; the write action triggers a second prompt to the frozen LLM that fills the entry fields under a fixed template, and the resulting record is committed to the store. Failed demonstrations trigger `WRITE_FAIL` instead of being discarded. At inference, the controller emits a read action conditioned on the query encoding and memory summary, and the retrieved entries are concatenated into a compact prompt that the frozen LLM consumes to produce the tool call.

4.8. Learned Generalization of Retrieval-Augmented Generation

Standard retrieval-augmented generation passively encodes each document chunk as a key and value, then retrieves the top-k by fixed vector similarity at query time. MemCtrl learns all three sides. Writes distill demonstrations into compact structured entries by prompting the frozen LLM to fill structured fields under a fixed template (the 4,000-token in-context trace becomes roughly 80 tokens of affordance, schema, pattern, and failure notes) rather than indexing raw text. The controller's contribution is choosing when and where these writes occur, not authoring the field text. Reads choose between top-k similarity and `READ_BY_SIG` structural match per query. And the policy is trained to retrieve what improves tool-call accuracy rather than what is most similar in encoder space, which is the central distinction from standard RAG.

5. Experimental Setup

5.1. Benchmarks

We evaluate on three function-calling benchmarks. The Berkeley Function Calling Leaderboard v3 (Patil et al., 2024) provides multi-turn function calling with diverse APIs across single-turn, multi-turn, and relevance-detection categories. We use the multi-turn subset for held-out tool evaluation because it most closely matches deployment conditions. τ-bench (Yao et al., 2024) provides dynamic agent-user conversations over retail and airline domains with policy documents and database-state evaluation. NexusBench provides composite API tasks where tool invocations chain across multiple calls.

5.2. Held-Out Tool Protocol

For each benchmark we partition the tool catalog into 80% training tools and 20% held-out tools. The held-out partition is fixed across all methods and seeds. For each held-out tool, the system receives $K \in \{0, 1, 5, 25\}$ demonstrations and is evaluated on a disjoint query set requiring that tool. The headline curve plots end-to-end accuracy as a function of K.

5.3. Models and Hardware

The primary backbone is Llama 3.1 8B Instruct. The controller is a 12-layer transformer encoder with hidden dimension 1024 and 200M total parameters. Cross-backbone transfer experiments use Qwen 2.5 7B Instruct as the deployment backbone. Training runs on 4 NVIDIA A100 80GB

GPUs with ZeRO-2 optimizer sharding. The frozen LLM is served via vLLM with tensor parallelism 2; the controller training loop runs on the remaining 2 GPUs.

5.4. Training Data

The controller training corpus is drawn from four sources. From each evaluation benchmark we extract the union of tools defined across its categories and partition the tool catalog 80/20 into training and held-out tools, then generate per-tool demonstrations and evaluation queries for the training partition. BFCL-v3 (Patil et al., 2024) contributes 280 training tools and 70 held-out tools (we ignore its multi-turn execution traces and use only the tool schemas). τ-bench (Yao et al., 2024) contributes 35 training tools and 15 held-out tools drawn from the retail and airline domains. NexusBench contributes 90 training tools and 25 held-out tools across the seven released sub-datasets (NVDLibrary, VirusTotal, PlacesAPI, ClimateAPI, OTX, CVECPE, and the parallel/nested variants). A synthetic generator produces an additional 400 training tools and 100 held-out tools, authored by GPT-4-class models and filtered for schema validity. Per-tool demonstrations and evaluation queries are synthesized by prompting a separate GPT-4-class model with the tool schema; the generator is the same for all four sources. Each tool yields on average 5 demonstrations and 8 evaluation queries.

5.5. Baselines

We compare against six baselines, all sharing the same Llama 3.1 8B backbone where applicable.

- **B1 in-context tool documentation.** The K demonstrations and full tool schema are injected into the system prompt; the language model is prompted to issue a tool call. This is the production-style baseline used by standard function-calling APIs.
- **B2 static retrieval-augmented generation.** A frozen sentence transformer indexes a corpus of authored tool documentation chunks; the top three chunks by cosine similarity are retrieved and concatenated into the prompt. The retrieval policy is fixed.
- **B3 supervised fine-tuning.** The full Llama 3.1 8B backbone is supervised-fine-tuned on the K demonstrations of each held-out tool. The model is updated once per held-out tool.
- **B4 LoRA fine-tuning.** LoRA adapters with rank 32 are fine-tuned on the same demonstrations as B3, leaving the base weights frozen but allowing rank-32 updates. This is the closest baseline to MemCtrl in parameter footprint and is the natural strawman for "just LoRA on tool data".
- **B5 Memory-R1 adapted to tools.** The Mem0 operator set with the full Llama 3.1 8B as the memory manager, trained with GRPO on the same data as MemCtrl. This isolates the contribution of the disentangled architecture by holding the RL signal and training data fixed while varying the trainable surface.
- **B6 Mem-α adapted to tools.** The richer Mem-α toolkit, again with the full 8B as the memory manager, trained on the same data. This is the strongest non-disentangled baseline.

Table 2. Held-out tool accuracy on BFCL-v3 as a function of K. All methods use Llama 3.1 8B Instruct as the backbone where applicable. Best per column in bold.

METHOD	PARAMS	K=0	K=1	K=5	K=25
B1 IN-CONTEXT	0	11.2	38.4	55.2	58.9
B2 STATIC RAG	0	42.1	48.5	53.7	55.0
B3 FULL SFT	8.0B	13.0	31.2	60.8	71.4
B4 LORA R32	84M	12.8	30.5	58.6	68.7
B5 MEMORY-R1	8.0B	21.0	44.6	62.9	69.5
B6 MEM-α	8.0B	20.4	45.8	63.7	69.8
MEMCTRL	**200M**	**38.9**	**52.7**	**67.4**	**72.1**

5.6. Implementation Details

Training uses the `trl` library with vLLM-backed rollouts. Hyperparameters are learning rate $1\mathrm{e}{-5}$ with cosine decay, gradient batch size 64 trajectories, GRPO group size $G = 8$, clipping $\epsilon = 0.2$, KL coefficient $\beta = 0.01$. Reward coefficients are $\alpha = 1.0$, $\beta_R = 1\mathrm{e}{-4}$, $\gamma = 1\mathrm{e}{-3}$. Total training compute is approximately 250 GPU hours.

6. Results

6.1. Main Result on Held-Out Tools

Table 2 reports BFCL-v3 held-out tool accuracy as a function of K. All accuracy numbers in this section use AST equivalence as defined in Section 4.5, the same metric the controller is rewarded against during training. MemCtrl achieves 67.4 accuracy at K=5, compared to 60.8 for supervised fine-tuning of the same 8B backbone (baseline B3) and 55.2 for in-context documentation (baseline B1), while using only 200M trainable parameters compared to 8B for B3 and 8B for the Memory-R1 baseline (B5).

Figure 2 visualizes the same data across all six baselines. MemCtrl dominates at low K and matches or exceeds supervised fine-tuning (B3) at high K. The largest relative gain is at K=1, where MemCtrl reaches 52.7 against 31.2 for supervised fine-tuning and 38.4 for in-context documentation, indicating that the controller compresses information from a single demonstration much more effectively than either passive retrieval or weight-level adaptation.

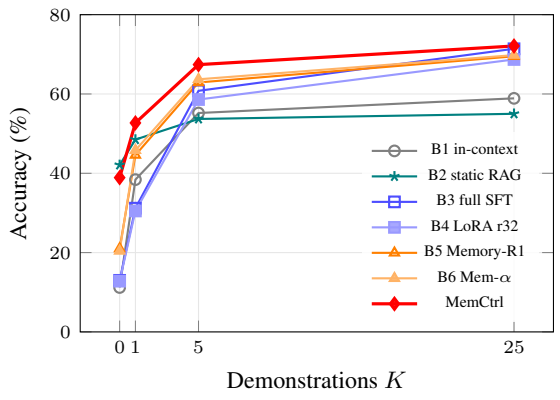

Figure 2. Few-shot accuracy on BFCL-v3 held-out tools. All six baselines and MemCtrl share the same Llama 3.1 8B backbone where applicable.

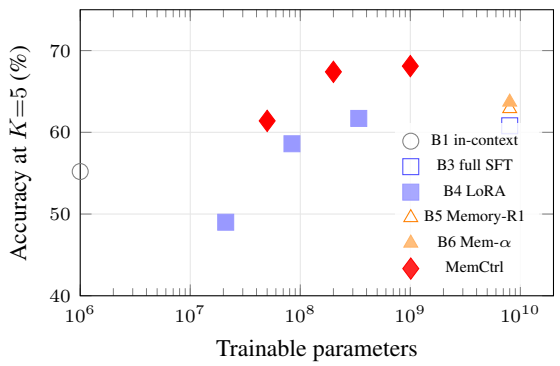

Figure 3. Parameter-accuracy. MemCtrl at 200M parameters sits on the upper-left frontier, exceeding all 8B baselines.

6.2. Parameter Efficiency

Figure 3 plots accuracy at K=5 against trainable parameter count for all methods on BFCL-v3 and τ-bench averaged. MemCtrl at 200M parameters exceeds Memory-R1 (B5) and Mem-α (B6) at 8B by 3.7 and 3.1 points respectively, placing it on the upper-left frontier of the Pareto plot. The 50M MemCtrl variant already exceeds in-context documentation and approaches Memory-R1, indicating that the gain comes from the policy structure rather than raw capacity.

6.3. Inference Token Cost

A central deployment metric is per-query input tokens. Table 3 compares the average input token count at K=5 on BFCL-v3. In-context documentation (B1) consumes 4,820 input tokens per query because it inlines all five demonstrations plus the schema. MemCtrl retrieves a single 620-token entry on average and achieves 12.2 points higher accuracy. This 8x token-cost reduction with simultaneously higher accuracy is the consequence of the frozen LLM compressing demonstrations into compact tool descriptions under

Table 3. Average input tokens per evaluation query at K=5 on BFCL-v3. Excludes one-time write-time LLM calls during ingestion ($\sim$120 input tokens per non-noop write, amortized over the eval query set).

METHOD	TOKENS	ACC	TOKS/PT
B1 IN-CONTEXT	4820	55.2	87.3
B2 STATIC RAG	1640	53.7	30.5
B5 MEMORY-R1	2100	62.9	33.4
MEMCTRL TOP-1	**620**	**67.4**	**9.2**
MEMCTRL TOP-3	1180	67.8	17.4

Table 4. Disentanglement ablation. Relaxing the frozen-backbone constraint improves in-distribution accuracy but degrades few-shot generalization.

TRAINABLE BACKBONE	FEW-SHOT	IN-DIST
FROZEN	**67.4**	70.2
LORA R8 JOINTLY TRAINED	65.1	72.8
LORA R32 JOINTLY TRAINED	62.6	75.5
LORA R128 JOINTLY TRAINED	60.9	78.0
FULL LLM JOINTLY TRAINED	58.7	80.4

controller-issued write operations, rather than indexing raw text. The cost reported here is inference only; write-time LLM calls add an amortized $\sim$120 input tokens per write action during ingestion (one call per non-noop write, averaged over the eval query set).

6.4. Disentanglement Ablation

The central architectural claim is that the frozen-backbone constraint is necessary for few-shot generalization. We test this by relaxing the constraint and jointly fine-tuning the backbone with the controller. Table 4 shows that updating the base model's weights during training improves in-distribution accuracy (queries about training tools) but degrades few-shot accuracy (queries about held-out tools) monotonically with the rank of the backbone update. Full joint training produces the best in-distribution accuracy and the worst few-shot accuracy.

The mechanism is that the controller learns its memory operations relative to the backbone's representation space. When that space is held fixed, the controller's policy generalizes to held-out tools because the same representation principles apply. When the backbone drifts during joint training, the controller's trained policy becomes misaligned with the new representation space, and the controller cannot adapt at deployment without retraining.

6.5. Action Space Ablation

Table 5 isolates the contribution of each action by training MemCtrl variants with progressively smaller action spaces. The largest single contribution comes from `WRITE_FAIL`,

Table 5. Action space ablation on BFCL-v3 at K=5.

ACTION SPACE	ACCURACY
MINIMAL (ADD, NOOP)	53.2
MEM0 SET (ADD, UPDATE, DELETE, NOOP)	60.5
+ MERGE	62.3
+ READ_BY_SIG	64.5
+ WRITE_FAIL	66.9
FULL (8 ACTIONS)	**67.4**

Table 6. Reliability on τ-bench airline at K=5. Pass^k is the fraction of k independent trials that all succeed.

METHOD	PASS^1	PASS^4	PASS^8
B1 IN-CONTEXT	34.7	16.1	8.9
B5 MEMORY-R1	40.2	22.6	15.0
B6 MEM-α	41.0	23.8	15.9
MEMCTRL	**44.1**	**27.4**	**19.6**

which adds 2.4 points over the variant that lacks it. This is the operation absent from all prior memory systems.

6.6. Failure-Mode Retrieval Study

To understand how WRITE_FAIL contributes, we identify the subset of evaluation queries on BFCL-v3 whose ground-truth call requires avoiding a unit conversion or argument-format trap that one or more demonstrations exhibit. On this subset, MemCtrl with the full action space achieves 81.3 accuracy; MemCtrl without WRITE_FAIL achieves 64.2; supervised fine-tuning (B3) achieves 58.7. The failure entries are retrieved alongside the main tool entry whenever the query's argument types match, and the LLM consumes the failure note in context as part of the standard prompt. This isolates the value of the operation as a learned mechanism for surfacing negative examples at inference, an operation that no similarity-based retrieval system can perform because failures and successes can have nearly identical surface forms.

6.7. τ-bench Reliability

Table 6 reports pass^k on τ-bench airline at K=5, where pass^k is the fraction of k independent trials that all succeed. MemCtrl narrows the pass^1 to pass^8 gap relative to all baselines, indicating that the learned entries capture robust tool descriptions rather than overfitting to particular demonstrations.

6.8. Cross-Backbone Transfer

A consequence of the disentanglement is that the controller should transfer across compatible backbones without retraining. We train the controller against Llama 3.1 8B and evaluate it against Qwen 2.5 7B at inference. Figure 4 shows the result across four pairings. The Llama-to-Qwen transfer loses 2.6 points relative to the same-backbone setting; no LoRA or full-fine-tune baseline can produce comparable cross-family adaptation without retraining.

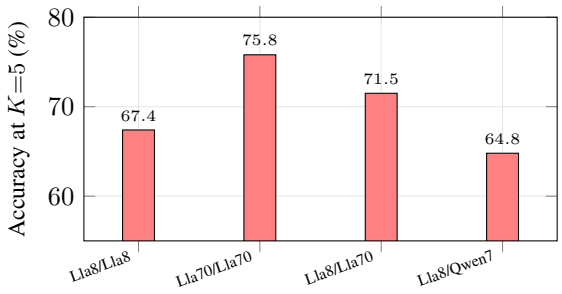

Figure 4. Cross-backbone transfer. Bars show MemCtrl accuracy under train/test backbone pairings. The 8B-to-Qwen-7B transfer loses 2.6 points relative to same-backbone, a tradeoff no joint-finetuning method can offer.

7. Limitations

We evaluate on 8B backbones. The framework predicts that disentangled training transfers cleanly to larger backbones because the policy is grounded in representation principles rather than weight-level adaptation, but we have not verified this empirically and budget did not permit 70B-scale runs. Our synthetic tool generator produces schemas drawn from a finite design space and may not cover all real-world API patterns. The eight-action vocabulary, while richer than prior work, is still hand-designed; learning the action vocabulary itself is an open question. Our evaluation focuses on single-tool acquisition; multi-tool composition under a shared memory store is a natural next setting.

8. Conclusion

We have presented a disentangled memory control framework for few-shot tool acquisition in frozen language models. Training a small reinforcement-learning controller over a fixed backbone matches or exceeds full backbone fine-tuning on three function-calling benchmarks while using two orders of magnitude fewer trainable parameters and substantially lower inference token cost. The framework is a learned generalization of retrieval-augmented generation in which write and read policies are trained from task reward, and it inherits the non-parametric adaptation property of dense KNN systems while adding learned compression and failure-mode logging. The disentanglement ablation shows that freezing the backbone is necessary for few-shot generalization, not just a way to save parameters.

References

Bao, J., et al. Mem-α: Learning memory construction via reinforcement learning. *arXiv preprint arXiv:2509.25911*, 2025.

Chhikara, P., et al. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: nearest neighbor language models. In *ICLR*, 2020.

Mireshghallah, N., Vogler, N., He, J., Florez, O. U., El-Kishky, A., and Berg-Kirkpatrick, T. Simple temporal adaptation to changing label sets: hashtag prediction via dense KNN. In *EMNLP*, 2023.

Packer, C., Wooders, S., Lin, K., et al. MemGPT: towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

Patil, S. G., et al. The Berkeley function calling leaderboard. 2024.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. Meta-learning with memory-augmented neural networks. In *ICML*, 2016.

Schick, T., Dwivedi-Yu, J., Dessì, R., et al. Toolformer: language models can teach themselves to use tools. In *NeurIPS*, 2023.

Shao, Z., et al. DeepSeekMath: pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Weston, J., Chopra, S., and Bordes, A. Memory networks. In *ICLR*, 2015.

Yao, S., et al. τ-bench: a benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Yu, S., et al. Memory-R1: enhancing large language model agents to manage and utilize memories via reinforcement learning. *arXiv preprint arXiv:2508.19828*, 2025.

Zheng, R., et al. Multi-turn reinforcement learning for tool-calling agents with iterative reward calibration. *arXiv preprint arXiv:2604.02869*, 2025.

A. Hyperparameters

Hyperparameter	Value
Controller hidden dim	1024
Controller layers	12
Controller params	200M
Optimizer	AdamW
Learning rate	$1e{-5}$
LR schedule	cosine, 5% warmup
Weight decay	0.01
Gradient batch (trajectories)	64
GRPO group size G	8
GRPO clip ϵ	0.2
KL coefficient β	0.01
Reward α, β_R, γ	1.0, 10^{-4}, 10^{-3}
Epochs	4
Backbone	Llama 3.1 8B Instruct (frozen)
Hardware	4x A100 80GB
Total compute	250 GPU hours

B. Synthetic Data Generator

The synthetic generator produces 500 tools (400 training, 100 held-out) by prompting a GPT-4-class model with a coverage specification, then filtering for schema validity. The corpus will be released with the paper.

Coverage. Table 7 summarizes the distribution of tool shapes in the synthetic corpus, chosen to mirror the variety of real APIs the controller must handle at deployment.

Table 7. Synthetic corpus coverage (500 tools total).

PROPERTY	TOOLS
1–2 REQUIRED ARGUMENTS	142
3–5 REQUIRED ARGUMENTS	261
6+ REQUIRED ARGUMENTS	97
HAS ENUM-VALUED ARGUMENT	188
HAS NESTED-OBJECT ARGUMENT	76
HAS OPTIONAL ARGUMENT	312
HAS UNIT-CONVERSION TRAP	89
HAS DEMOS WITH EXEC. ERROR	145

Generator prompt. The generator receives a coverage specification (e.g., 6+ args with one enum and an optional argument) and emits a tool schema together with $K+|Q|$ demonstrations and evaluation queries.

Listing 2. Generator prompt sketch.

```
System: You are designing a function-calling API for an
agent. Produce one tool that satisfies the spec below.
Spec: {n_required_args, has_enum, has_nested,
       has_unit_trap, has_optional}
Return:
  - tool name (dot-separated namespace)
  - JSON schema with argument names, types, descriptions
  - 5 demonstrations: realistic user queries paired with
    the correct tool call
  - 8 evaluation queries that test generalization
  - For unit-trap tools, include at least one demo with
    a known incorrect call and the execution error.
```

Filter criteria. A tool is accepted into the corpus if and only if (a) its JSON schema parses and validates, (b) every demonstration's tool call satisfies the schema, (c) argument names and the tool name are unique across the corpus, and (d) at

least one evaluation query has a ground-truth call that is not in the demonstration set. Rejected drafts are regenerated up to three times before being dropped; the final accept rate is 71%.

Examples. Listing 3 shows five tools that span the corpus. Comments above each tool indicate which shape it illustrates.

Listing 3. Five synthetic tools.

```
# simple, 4 required args
{
  "name": "weather.forecast_hourly",
  "params": {
    "lat": "float, latitude",
    "lon": "float, longitude",
    "hours": "int, 1 to 168",
    "units": "string, metric|imperial, optional"
  },
  "demos": [
    {"q": "Hourly forecast for SF next 24 hours, metric",
     "call": "weather.forecast_hourly(lat=37.77,
              lon=-122.42, hours=24, units='metric')"}
  ]
}
# enum + list args, multi-user composition
{
  "name": "calendar.find_overlap",
  "params": {
    "user_ids": "list[string], at least 2",
    "window_start": "string, ISO 8601",
    "window_end":   "string, ISO 8601",
    "duration_min": "int, default 30",
    "tz": "string, IANA timezone, optional"
  },
  "demos": [
    {"q": "Find a 45-min slot for alice and bob Fri pm",
     "call": "calendar.find_overlap(
        user_ids=['alice','bob'],
        window_start='2024-05-24T12:00',
        window_end='2024-05-24T18:00',
        duration_min=45)"}
  ]
}
# unit-conversion trap (basis points vs decimal)
{
  "name": "fx.convert",
  "params": {
    "amount_cents": "int, source amount in cents",
    "from_ccy": "string, ISO-4217",
    "to_ccy": "string, ISO-4217",
    "fee_bps": "int, fee in basis points"
  },
  "demos": [
    {"q": "Convert $250 USD to EUR with 0.5% fee",
     "call_attempted": "fx.convert(amount_cents=250,
       from_ccy='USD', to_ccy='EUR', fee_bps=0.5)",
     "error": "amount must be in cents; fee_bps integer",
     "correct_call": "fx.convert(amount_cents=25000,
       from_ccy='USD', to_ccy='EUR', fee_bps=50)"}
  ]
}
# nested object argument
{
  "name": "ride.book",
  "params": {
    "pickup": {"lat": "float", "lon": "float",
               "address": "string, optional"},
    "dropoff": {"lat": "float", "lon": "float"},
    "vehicle_class": "string, economy|standard|premium"
  },
  "demos": [
    {"q": "Standard ride from SFO to 1 Market",
     "call": "ride.book(
        pickup={'lat':37.62,'lon':-122.38},
        dropoff={'lat':37.79,'lon':-122.40},
        vehicle_class='standard')"}
  ]
}
# many optionals, dense schema
{
  "name": "search.products",
```

```
  "params": {
    "query": "string, required",
    "category": "string, optional",
    "min_price_cents": "int, optional",
    "max_price_cents": "int, optional",
    "in_stock_only": "bool, default true",
    "sort_by": "string, price|rating|relevance, default relevance",
    "limit": "int, 1 to 100, default 20"
  },
  "demos": [
    {"q": "Show in-stock keyboards under $80, by rating",
     "call": "search.products(query='keyboard',
        max_price_cents=8000, sort_by='rating')"}
  ]
}
```