MCPool: LEARNING JOINT MEMORY ALLOCATION FOR HETEROGENEOUS REUSE IN MCP AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents using the Model Context Protocol (MCP) can reduce tool-invocation latency by retaining initialized execution contexts and reusing eligible tool results. These two forms of reuse differ in memory footprint, lifetime, and validity, yet compete for the same memory budget. Their benefits also overlap: a result hit avoids both context acquisition and tool execution. We formulate this heterogeneous dual-resource management task as a partially observable sequential decision problem, where future requests are unknown and retention decisions yield delayed latency benefits. To address this, we present MCPool, an RL-based controller for jointly managing context and result retention under a shared memory budget. Specifically, a recurrent policy summarizes class-level runtime statistics and selects a memory split and class-specific admission thresholds. A deterministic manager translates these controls into entity-level admission and eviction decisions, and enforces the shared budget limit independently of the learned policy. This separation enables robust scalability to shifting cache populations. Our extensive experiments across a wide range of tasks and baselines demonstrate that MCPool achieves a $1.34\times \sim 2.38\times$ speedup while satisfying the memory budget with zero observed violations.

1 INTRODUCTION

LLM agents increasingly invoke external tools through the Model Context Protocol (MCP) (Anthropic, 2024; Qu et al., 2025). Each invocation incurs host-side overhead (Hasan et al., 2026; Zhai et al., 2026): a cold invocation locates the target server, starts its process or opens its connection, initializes its tools, and only then executes the tool. In our profiling (Fig. 2), context acquisition accounts for 4%–35% of cold-start latency, and warm starts reduce this overhead. Meanwhile, agent trajectories exhibit temporal locality through repeated server visits and queries (Jia & Li, 2026; Begum et al., 2026), enabling cached tool results to bypass repeated execution (Vijaya Kumar et al., 2026; Zhai et al., 2026).

As illustrated in Fig. 1, MCP exposes two complementary opportunities for acceleration. **1) Context reuse accelerates preparation.** Retaining an initialized server process or active connection lets later calls reuse the same execution context instead of paying its setup cost again (Ahmadi et al., 2025). This opportunity persists even when successive calls to a server invoke different tools or arguments. **2) Result reuse accelerates execution.** When a prior result matches the request, the host can return it without re-executing the tool (Zhai et al., 2026). Such a hit bypasses context acquisition as well as execution, so the two caches accelerate different portions of the invocation path with overlapping rather than independently additive benefits. Contexts and results differ in memory footprint, lifetime, invalidation semantics, and latency benefit (Begum et al., 2026; Vijaya Kumar et al., 2026), yet compete for the same retained-memory budget: a memory-heavy server context can displace many small results,

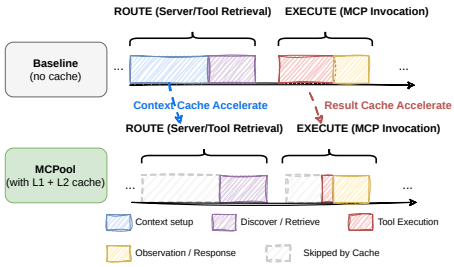

Figure 1: Two-stage acceleration in MCPool: context reuse avoids repeated setup, while result reuse bypasses redundant tool execution.

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.

while hoarding low-value results can force a frequently used context to be evicted and reinitialized repeatedly. Their benefits are also coupled. A retained context helps only requests whose results miss, and a retained result saves both context acquisition and execution when the context is cold. The value of retaining either kind therefore depends on what is retained of the other, and the preferred split shifts with the balance between exact-result locality and broader server-level locality.

Existing systems provide building blocks for each reuse opportunity. Tool and result caching reuses prior tool outputs through TTLs, adaptive admission and eviction, or state-aware matching (Kulkarni et al., 2026; Zhai et al., 2026; Vijaya Kumar et al., 2026). Context and warm-state management retains server processes or execution state to avoid repeated initialization (Ahmadi et al., 2025). More generally, adaptive cache management accounts for object cost and size, repartitions memory as demand changes, or learns admission and replacement decisions online (Cao & Irani, 1997; Hu et al., 2015; Cidon et al., 2017; Vietri et al., 2018; Rodriguez et al., 2021; Kirilin et al., 2019). These approaches largely optimize a single type of reusable entity or allocate resources among similar cached states. Our focus instead is on allocating a shared retained-memory budget between two heterogeneous reuse classes with coupled benefits, using learned class-level control and a policy-independent feasibility manager. Related resource-sharing problems also arise in LLM serving, where different forms of KV state compete for limited memory (Kwon et al., 2023; Zheng et al., 2024). We formalize the MCP instance as an *online sequential resource control problem*. At each control epoch, a policy observes class-level runtime statistics and selects class-level controls, a memory split and per-class admission thresholds; a deterministic manager then makes the entity-level admission and eviction decisions that keep retained memory within budget. MCPool is a learned controller with this structure.

Our main contributions are summarized as follows:

Optimization Opportunities and Problem Formulation: We identify complementary opportunities to reduce MCP invocation latency by reusing warm execution contexts and tool results, and expose the trade-off created by their coupled latency benefits and unequal memory footprints. We formulate their joint management as a partially observable online sequential decision problem that minimizes latency under a retained-memory budget.

RL-Based Joint Resource Control: MCPool pairs conventional recurrent proximal policy optimization (RPPO) with a deterministic cache manager. The policy acts on a fixed-dimensional class-level action space (a memory split and two admission thresholds), so the actor interface does not grow with the cache population, although entity-level bookkeeping still does; recurrent history and interval returns carry workload trends and delayed reuse benefits. The manager translates these controls into entity-level admission and eviction under shared memory constraints, and its retained-memory guarantee holds under the manager's accounting for any policy (Proposition 1).

Experimental Evaluation: We evaluate MCPool against five baselines on MCP-Bench across three foundation models. MCPool achieves $1.34\times$–$2.38\times$ mean task-latency speedups and up to $2.62\times$ P95 speedup over Static-Control, with zero observed memory-budget violations. Budget sweeps, component ablations, and workload variations further assess the benefits of joint learned control.

2 From MCP Cache Opportunities to Sequential Decisions

Modern tool-augmented agent frameworks suffer substantial protocol-level overheads. While prior work addresses tool-result reuse (Zhai et al., 2026; Vijaya Kumar et al., 2026), our profiling reveals critical setup bottlenecks across transport protocols (Fig. 2). As shown in Fig. 2, context acquisition accounts for approximately 4%–34% of cold-start latency across the two transport settings. Under HTTP, setup overhead drops to $\sim 4\%$, as execution latency is predominantly governed by network conditions during tool execution. In both cases, transitioning from a Cold Start to a Warm Start completely eliminates these protocol setup overheads, leaving only the physical tool execution time. This phenomenon motivates two complementary reuse paradigms: (1) *Context Reuse*, which retains initialized session contexts to bypass cold-start setup delays, and (2) *Result Reuse*, which buffers valid execution outputs to eliminate physical execution latency entirely. However, co-optimizing both paradigms introduces severe spatial competition under a shared memory budget. As illustrated in Fig. 3, warm-context footprints vary significantly across servers, ranging from $\sim$ 70 MB for lightweight endpoints to over 175 MB for complex environments, representing a $> 2.5\times$ disparity.

Retaining execution contexts across hundreds of MCP servers can impose substantial memory overhead. Even across just 28 MCP-Bench environments, aggregate context retention exceeds 1.85,GB (Appendix B), making memory a practical constraint for context reuse. In contrast, cached tool results carry lightweight individual footprints but scale rapidly in volume. Because a retained context can serve diverse tools on the same server, shifting workload distributions mandate joint, adaptive memory allocation.

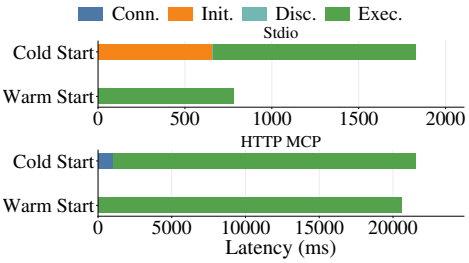

Figure 2: Cold and warm invocation latency breakdown.

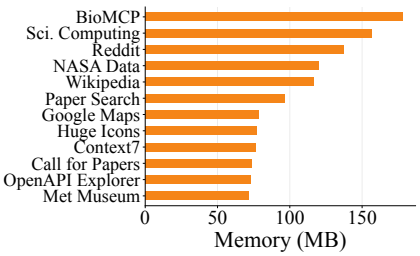

Figure 3: Warm-context memory footprints for 12 profiled servers.

Managing this dual-reuse system online presents four core challenges: (1) *uncertain future reuse* due to invisible agent trajectories; (2) *partial observability* from LLM internal planning states; (3) *delayed latency feedback* complicating credit assignment; and (4) *hard resource constraints* requiring zero budget violations at every cache event. We formalize this task as a constrained partially observable sequential decision process (Section 3), and resolve it via a recurrent learning controller with deterministic byte enforcement (Section 4). Extended profiling details appear in Appendix B.

3 Online Sequential Decision Formulation

The four challenges in Section 2 motivate the formulation below. We consider online heterogeneous-cache control under a non-stationary, task-driven workload, modeled as a hard-constrained partially observable sequential decision process. Uncertain reuse and hidden task progress determine what the controller can observe; delayed latency consequences determine its objective; the shared memory budget restricts every cache transition.

3.1 Task-Driven Dynamics, Objective, and Constraints

An agent executes a sequence of tasks, each unfolding into requests u_t, $t = 1, 2, \ldots$, as intermediate results determine its next step. The request mix, locality, and reuse costs can change with task progress; we do not assume independent requests or a fixed request distribution. At step t, the host serves u_t using the inherited cache, observes new candidates and service feedback, and decides what to retain for later steps. Each retention decision is made after the current request completes and before the next begins, when all resident entities are available for eviction. Its observed history $\mathcal{H}_t$ contains past and current requests, measured costs, cache contents, and previous decisions, but not the agent's remaining plan or future requests.

Let $\mathcal{Q} = \{1, \ldots, Q\}$ denote fixed cache classes. At step t, the system observes resident entities $\mathcal{E}_t^q$ and online candidates $\mathcal{G}_t^q$ for each class q; each resident entity k carries a charge $m_k^q > 0$, the bytes the manager accounts to it (Section 3.3). The decision $b_t = \{(\mathcal{A}_t^q, \mathcal{D}_t^q)\}_{q \in \mathcal{Q}}$ admits candidates $\mathcal{A}_t^q \subseteq \mathcal{G}_t^q$ and evicts residents $\mathcal{D}_t^q \subseteq \mathcal{E}_t^q$. Invalidated entries $\mathcal{I}_t^q$ leave lookup immediately and serve no new hit; the eviction set satisfies $\mathcal{I}_t^q \subseteq \mathcal{D}_t^q$. The cache updates via $\mathcal{E}_{t+1}^q = (\mathcal{E}_t^q \setminus \mathcal{D}_t^q) \cup \mathcal{A}_t^q$, where an evicted entry leaves $\mathcal{E}_t^q$, and releases its charge, only when its physical reclamation completes.

Given hit indicators $\mathbf{x}_t$ for request u_t, the step latency $L_t = g(u_t, \mathbf{x}_t) + c_t(b_t)$ integrates service work g and management cost c_t. Overlapping benefits are inherently captured by g, as higher-level reuses can bypass low-level execution work altogether. Our goal is to learn a causal policy $\pi(b_t \mid \mathcal{H}_t)$

minimizing long-run average latency under a retained-memory budget B:

$$\min_{\pi} \quad \limsup_{T\to\infty} \frac{1}{T}\mathbb{E}_{\pi}\left[\sum_{t=1}^{T} L_t\right] \tag{1a}$$

$$\text{s.t.} \quad M_{t+1} = \sum_{q\in\mathcal{Q}} \sum_{k\in\mathcal{E}_{t+1}^{q}} m_k^q \le B, \quad \forall t. \tag{1b}$$

Here M_{t+1} sums the charges of every resident entity. B therefore bounds retained, charged cache memory, not host memory, which also holds the temporary contexts of in-flight cold acquisitions and the host and agent runtime (Appendix C.1). Proposition 1 shows that the manager's rules maintain (1b) on every execution path.

3.2 A Resource-Level Control Interface

Directly optimizing entity-level decisions b_t is difficult for two reasons. The set of candidate entities changes online, so the input and output dimensions of a learned policy are unstable, and enumerating valid retention subsets over J candidates yields an $O(2^J)$ search space. We therefore introduce a fixed-dimensional resource-level control interface that decouples allocation policy from entity management. At control epoch d, the policy emits:

$$a_d = (\boldsymbol{\alpha}_d, \boldsymbol{\theta}_d), \qquad \boldsymbol{\alpha}_d \in \Delta^{Q-1}, \quad \boldsymbol{\theta}_d \in \mathbb{R}_+^Q, \tag{2}$$

where $\boldsymbol{\alpha}_d$ specifies the proportional memory quota allocation $B_d^q = \alpha_d^q B$ across classes ($\sum_q \alpha_d^q = 1$), and $\boldsymbol{\theta}_d$ defines class-specific selectivity admission thresholds. The manager enforces each applied class quota separately; unused capacity is not borrowed across classes. A deterministic, rule-based cache manager combines a_d with current entity metrics to make feasible entity-level admission and eviction decisions b_t. The action space has $2Q-1$ degrees of freedom independent of entity cardinality, although entity-level management still depends on the cache population. Temporal credit assignment and partial observability are left to the history-aware controller in Section 4, and the manager's scoring and eviction rules are given in Section 4.1.

3.3 Instantiation in MCP Agents

In MCP, $\mathcal{Q} = \{c, o\}$ represents warm execution contexts and reusable tool outputs. MCPool operates at the host without modifying servers. Request $u_t = (s_t, z_t, \xi_t)$ identifies a server, tool, and normalized arguments. The resident sets are warm contexts $\mathcal{C}_t = \mathcal{E}_t^c$ and cached outputs $\mathcal{O}_t = \mathcal{E}_t^o$. With valid context and result hits $x_t^c, x_t^o \in \{0, 1\}$, latency is

$$L_t = \ell_t^{\text{route}} + (1 - x_t^o)\big[(1 - x_t^c)\ell_t^c + \ell_t^o\big] + \ell_t^p, \tag{3}$$

where ℓ_t^{route} is the routing latency, ℓ_t^c is the context-acquisition latency, ℓ_t^o is the tool-execution latency alone, and ℓ_t^p is the control and cache-management overhead. A context hit avoids acquisition; a result hit bypasses both acquisition and execution. This is the overlapping-benefit instance of g above. A miss may create retention candidates, and future task steps determine whether admitting them pays off. Thus MCP supplies concrete entities, latency measurements, and validity rules for the same sequential resource-control problem. Memory charges include retained context footprints or serialized result bytes, with detailed accounting rules in Appendix C.1. To avoid stale results, highly time-sensitive tools are marked *ineligible* for result reuse, while the remaining tools are *eligible*.

4 Hard-Constrained Recurrent Reinforcement Learning

We use recurrent proximal policy optimization (RPPO) to select the resource controls in Section 3.2. Its components address the four challenges in Section 2: observed access and cost feedback supplies estimates when future reuse is unknown; a recurrent state combines observations under partial observability; interval returns propagate delayed latency consequences; and a deterministic manager enforces the hard budget during both training and execution.

4.1 Resource Actions and Deterministic Execution

The controller updates its controls every W consecutive tool requests; $W = 8$ means one update per eight requests, with the request count continuing across task boundaries. A single action a_d governs these W requests, indexed by $t \in [t_d, t_{d+1})$. For the two-class MCP system ($Q = 2$), the action $a_d = (\beta_d, \theta_d^c, \theta_d^o)$, where $\beta_d \in [0, 1]$ and $\theta_d^c, \theta_d^o \geq 0$, specifies the memory allocation ratio and class-specific selectivity thresholds, yielding class quotas $B_d^c = \beta_d B$ and $B_d^o = (1-\beta_d)B$. The actor samples unconstrained Gaussian variables $(z_d^\beta, z_d^c, z_d^o)$ and maps them into this action domain by

$$\beta_d = \sigma(z_d^\beta), \qquad \theta_d^c = \mathrm{softplus}(z_d^c), \quad \theta_d^o = \mathrm{softplus}(z_d^o). \tag{4}$$

This parameterization guarantees the range constraints of the action; the manager then enforces the retained-memory budget.

Entity Value Density. Upon arrival of candidate entity k of class q, a deterministic manager evaluates its log-scaled *value density* $V_{t,k}^q$, based on reuse-probability-weighted potential latency savings normalized by footprint and reference latency:

$$V_{t,k}^q = \log\left(1 + \frac{\widehat{p}_{t,k}^q \widehat{b}_{t,k}^q}{(m_{t,k}^q/B)L_{\mathrm{ref}}}\right), \tag{5}$$

where $\widehat{p}_{t,k}^q$ is the estimated reuse probability, $\widehat{b}_{t,k}^q$ estimates the maximum potential latency saving from one reuse of entity k, and L_{ref} is the reference latency. For MCP, a context can serve multiple calls on the same server and saves setup latency ℓ^c, whereas a result serves an identical query and saves both setup and execution latency $\ell^c + \ell^o$. The score favors high potential savings per unit of memory, while the RL reward uses actual latency from (3) to account for overlapping benefits. Detailed estimation procedures are deferred to Appendix C.

Feasible Admission and Eviction Execution. The class-specific admission threshold θ_d^q controls retention selectivity. After each request completes, candidate k is admitted only if $V_{t,k}^q \geq \theta_d^q$ and its charge fits within its own class quota, after evicting lower-valued residents of that class in ascending $V_{t,k}^q$ order if needed; neither class borrows the other's unused quota. On each β_d update between requests, the manager evicts residents in ascending value within each class until the new quotas are met, then applies the split and admission thresholds. Under the accounting assumptions stated in Appendix C.2, the manager maintains $M_t \leq B$ after every event for any sequence of policy actions.

4.2 Partial Observability and Recurrent History

The cache manager operates under partial observability because the agent's internal task plan, upcoming reasoning branches, and future tool requests remain hidden from the host system. A static snapshot of instantaneous hit rates or cache occupancies cannot determine whether a workload is entering or leaving a temporal locality phase. We thus model the control path as a partially observable Markov decision process (POMDP) (Kaelbling et al., 1998) $\mathcal{M} = \langle \mathcal{X}, \mathcal{O}, \mathcal{A}_F, P, \Omega, r, \gamma \rangle$, where $\mathcal{X}$ and $\mathcal{O}$ are the latent state and observation spaces, $\mathcal{A}_F$ is the feasible action space, P and Ω are the transition and observation models, r is the reward, and γ is the discount factor. At control epoch d, the host system constructs a compact, fixed-dimensional observation vector o_d to characterize the system state without depending on fluctuating entity populations:

$$o_d = [\rho_d^c, \rho_d^o, \bar{h}_d^c, \bar{h}_d^o, \eta_d^c, \eta_d^o, \bar{\ell}_d^c, \bar{\ell}_d^o, \bar{m}_d^c, \bar{m}_d^o, \bar{V}_d^c, \bar{V}_d^o, \omega_d^c, \omega_d^o, \zeta_d^o]. \tag{6}$$

For each class $q \in \{c, o\}$, o_d guides policy adjustments across four core dimensions: i) *Utilization (ρ_d^q)* reflects memory pressure and available capacity; ii) *Efficiency ($\bar{h}_d^q$)* measures hit rates to evaluate quota effectiveness; iii) *Demand & Cost ($\eta_d^q, \bar{\ell}_d^q, \bar{m}_d^q, \bar{V}_d^q$)* captures arrival volume, miss penalties, candidate footprints, and value density (Eq. (5)) to inform threshold θ_d^q adjustments; and iv) *Stability & Safety (ω_d^q, ζ_d^o)* tracks memory churn to suppress thrashing, alongside the fraction of eligible results to prevent unutilizable quota allocation. Here, $\bar{V}_d^q$ averages the log-scaled candidate scores in (5); admission thresholds θ_d^q operate on the same log scale.

To mitigate the partial observability of each instantaneous observation, a Gated Recurrent Unit (GRU) aggregates history, the previously applied control a_{d-1}, and latency returns into a latent recurrent state h_d:

$$h_d = \mathrm{GRU}_\phi\big(h_{d-1}, [o_d, a_{d-1}, r_d]\big). \tag{7}$$

This recurrent representation h_d enables the controller to infer hidden workload trends and locality shifts across execution intervals (Cho et al., 2014; Hausknecht & Stone, 2015). The recurrent state persists across task boundaries within the same workload sequence, together with the retained cache, and resets only when an independent sequence begins.

4.3 Interval Feedback and PPO Optimization

Let $\mathcal{R}_d$ contain the W consecutive requests served under action a_d. After these requests, the action receives a normalized latency reward:

$$r_{d+1} = -\frac{1}{W L_{\mathrm{ref}}} \sum_{t \in \mathcal{R}_d} L_t. \tag{8}$$

Task boundaries neither close this request group nor reset recurrent history or value bootstrapping. Only the end of an independent sequence can leave fewer than W requests; their reward uses the same denominator $W L_{\mathrm{ref}}$, so each request retains the same weight.

For training, we maximize the discounted surrogate $J_R(\pi) = \mathbb{E}_\pi[\sum_{d \geq 0} \gamma^d r_{d+1}]$ over manager-executed policies. This supplies delayed feedback for the latency objective in (1); fixed-W normalization avoids overweighting short terminal groups, although temporal discounting still distinguishes the training surrogate from the long-run per-request average. A critic $V_\psi(h_d)$ and GAE (Schulman et al., 2016) assign later latency consequences to earlier controls:

$$\delta_d = r_{d+1} + \gamma V_\psi(h_{d+1}) - V_\psi(h_d), \qquad \widehat{A}_d = \sum_{l \geq 0} (\gamma \lambda_{\mathrm{GAE}})^l \delta_{d+l}. \tag{9}$$

The return has no capacity penalty: infeasible admissions are already excluded by execution. With likelihood ratio $w_d(\theta) = \pi_\theta(a_d \mid h_d)/\pi_{\theta_{\mathrm{old}}}(a_d \mid h_d)$, PPO maximizes

$$\mathcal{L}_{\mathrm{clip}}(\theta) = \widehat{\mathbb{E}}_d\Big[\min\Big(w_d(\theta)\widehat{A}_d, \mathrm{clip}(w_d(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_d\Big)\Big]. \tag{10}$$

The critic fits return targets and entropy regularization supports exploration (Schulman et al., 2017). Updates use ordered recurrent subsequences; the complete loss and procedure are in Appendix D. Every sampled action passes through the same deterministic manager, so parameter updates cannot violate the retained-memory bound of Proposition 1.

5 Evaluation

We evaluate MCPool[1] through four research questions: **RQ1:** How effective is MCPool compared with existing caching approaches? **RQ2:** How does MCPool behave under different resource constraints? **RQ3:** Which design components are important to MCPool's effectiveness? **RQ4:** How robust is MCPool under different workload and control settings?

5.1 Experimental Setup

Workloads and Models. We evaluate MCPool using gpt-5.4-mini (default for detailed analyses), claude-sonnet-5, and gpt-5.5. Workloads are derived from MCP-Bench (Wang et al., 2025)[2], encompassing 28 MCP servers and 250 tools. For RL policy optimization, we construct separate training and evaluation workloads from MCP-Bench dataset, consisting of 250 training tasks and 50 held-out evaluation tasks, respectively. The default memory budget is 320 MiB, with controls updated every $W = 8$ consecutive tool requests across task boundaries. Further settings and hyperparameters appear in Appendix E.

Baselines and Metrics. We benchmark MCPool against five representative methods: (1) ResultOnly: Adapts ToolCaching Zhai et al. (2026) with group-level UCB1 admission and value-aware LRU eviction; (2) ContextOnly: Exclusively retains initialized execution contexts; (3) Unified-LRU: Manages contexts and results in a single cache under a unified LRU order; (4) ValueGreedy: Dynamically evicts items based on a value-density heuristic; (5) Static-Control: Fixes static memory allocations and admission thresholds.

[1] Our implementation: `https://anonymous.4open.science/r/MCPool-release-23D5/`

[2] `https://github.com/Accenture/mcp-bench`

We evaluate performance via mean and P95 task latency speedups (normalized against Static-Control), memory-budget violation rates (% of observations exceeding the budget limit), and token-reduction factors. Task performance is evaluated by an external LLM judge, Gemini-3.7-Flash, which assesses the degree of task completion. (Details in Appendix E).

Table 1: End-to-end performance comparison across foundation models. Mean and P95 speedups and token-reduction factors are relative to Static-Control, with larger factors indicating better performance. Task performance is scored on a 1–10 scale.

Models	**Methods**	**Speedup**		**Violation**	**Token reduction**	**Performance**
		Mean	P95	(%)		
gpt-5.4-mini	ContextOnly	1.48×	1.60×	39.39	0.96×	7.30
	ResultOnly	1.10×	2.59×	0.00	0.99×	7.01
	Unified-LRU	1.25×	2.63×	54.04	1.03×	7.22
	Static-Control	1.00×	1.00×	0.00	1.00×	**7.55**
	ValueGreedy	1.09×	1.40×	37.66	1.09×	7.11
	MCPool	**2.38**×	**2.62**×	**0.00**	**1.95**×	7.35
claude-sonnet-5	ContextOnly	0.95×	0.94×	38.10	1.01×	**7.54**
	ResultOnly	0.98×	1.02×	0.00	1.10×	7.23
	Unified-LRU	0.97×	1.01×	53.70	1.11×	7.45
	Static-Control	1.00×	1.00×	0.00	1.00×	7.31
	ValueGreedy	0.99×	0.99×	52.19	1.02×	7.42
	MCPool	**1.34**×	**1.18**×	**0.00**	**1.42**×	7.53
gpt-5.5	ContextOnly	1.08×	1.03×	39.47	0.99×	8.24
	ResultOnly	0.93×	0.82×	0.00	0.95×	8.06
	Unified-LRU	0.92×	0.98×	52.48	1.01×	8.09
	Static-Control	1.00×	1.00×	0.00	1.00×	8.23
	ValueGreedy	1.00×	1.20×	51.28	1.00×	8.12
	MCPool	**1.54**×	**1.35**×	**0.00**	**1.06**×	**8.27**

5.2 Main Results: End-to-End Performance (RQ1)

As shown in Table 1, MCPool delivers consistent performance improvements across all foundation models, achieving mean speedups of $1.34\times$–$2.38\times$ while maintaining a strict 0.00% memory-budget violation rate.

Execution Speedup and Model Dependency. On gpt-5.4-mini, MCPool achieves a $2.38\times$ mean speedup (38.0% lower mean latency than ContextOnly) and a $2.62\times$ P95 tail speedup, matching the tail acceleration of unconstrained heuristics like Unified-LRU ($2.63\times$). Speedup magnitudes vary on claude-sonnet-5 ($1.34\times$) and gpt-5.5 ($1.54\times$), reflecting model-dependent differences in tool invocation density and trajectories. Notably, heuristic baselines often yield sub-unitary speedups ($0.92\times$–$0.99\times$) on larger models due to cache management overheads under sparse reuse.

Token Consumption Savings. Beyond execution latency, MCPool consistently reduces prompt and context token consumption across all three models, by $1.95\times$, $1.42\times$, and $1.06\times$ on gpt-5.4-mini, claude-sonnet-5, and gpt-5.5, corresponding to savings of 48.7%, 29.6%, and 5.6% over Static-Control, respectively. On gpt-5.4-mini, token usage is also 44.0% lower than ValueGreedy. These savings come from result reuse, which avoids repeated tool invocations and the associated observations and interaction context that would otherwise be added to subsequent LLM calls.

Deterministic Budget Compliance. Unconstrained baselines—ContextOnly, Unified-LRU, and ValueGreedy—severely violate spatial boundaries, exhibiting high violation rates between 37.66% and 54.04%. While ResultOnly avoids violations (0.00%) by foregoing context caching entirely, it yields inferior latency gains. In contrast, MCPool pairs a learned policy for joint allocation and admission control with a deterministic manager that strictly enforces the byte budget at every cache event, achieving optimal performance with zero violations.

Importantly, efficiency gains in MCPool do not come at the cost of task execution accuracy. Across completed and successfully scored executions, MCPool maintains benchmark scores comparable to

or higher than baselines: 7.35 vs. 7.55 (Static-Control) on gpt-5.4-mini, 7.53 vs. 7.54 (ContextOnly) on claude-sonnet-5, and a top-tier score of 8.27 on gpt-5.5.

5.3 In-Depth Analysis

5.3.1 Temporal and latency breakdown analysis.

Figure 4 illustrates the cumulative speedup trajectory across processing request sequences. After initial warm-up fluctuations, MCPool sustains a prominent 2.5×–2.7× speedup through the middle phase and converges to a stable 1.66× end-to-end speedup, whereas ValueGreedy consistently lags near the baseline. This trajectory tracks the evolving distribution of reuse opportunities across task boundaries, confirming that MCPool delivers substantial cumulative runtime savings even as workload locality shifts. Table 2 further breaks down the per-request latency across system processing stages under MCPool. Cold starts account for only 1.60% of the remaining latency, with up to 1.75× speedup over ResultOnly. Physical tool execution accounts for 96.97% of the remaining latency, despite a 2.13× speedup over Static-Control, indicating that most remaining latency lies in actual tool execution rather than context setup. Routing contributes only 0.01%, with a 1.14× speedup over ValueGreedy, while policy inference and cache management account for 1.41%, with a 2.48× speedup over ResultOnly. Overall, MCPool reduces both context setup and redundant tool execution while keeping its control overhead small.

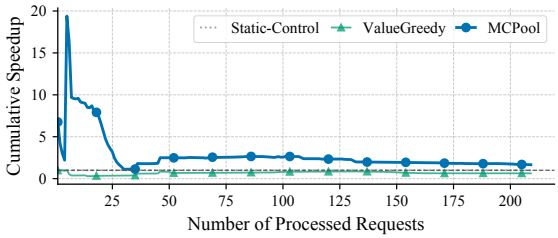

Figure 4: Cumulative speedup over processed requests.

Table 2: Per-request latency breakdown.

Stage	Up to	Share (%)
Routing	1.14×	0.01
Cold start	1.75×	1.60
Tool execution	2.13×	96.97
Control	2.48×	1.41
Overall	**1.52×**	**100.00**

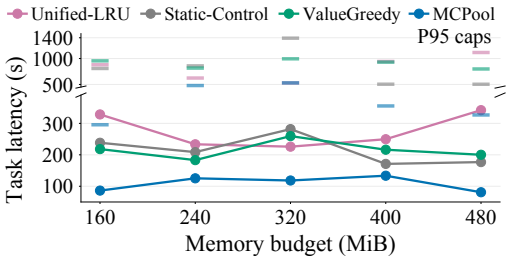

Figure 5: Task latency across memory budgets; caps indicate P95 latency.

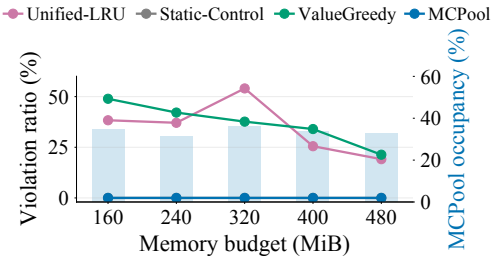

Figure 6: Memory-budget violation ratio and average occupancy across budgets.

5.3.2 Memory Budget Sensitivity (RQ2)

Across the entire 160–480 MiB sweep in Fig. 5, MCPool consistently maintains the lowest mean task latency. The P95 latency caps also show lower tail latency than Static-Control across these budgets, with up to a 2.73× speedup. As the memory budget expands, average memory occupancy shows a steady upward trend (Fig. 6). Importantly, despite the increasing occupancy, MCPool remains within the strict memory budget across all memory settings. In contrast, baseline heuristics like ValueGreedy exhibit high violation rates throughout the sweep, confirming that expanding physical memory alone cannot guarantee budget compliance unless paired with deterministic, byte-aware enforcement.

5.3.3 Ablation Analysis (RQ3)

Figure 7 compares MCPool with fixed allocation (FixedSplit), fixed admission thresholds (FixedThreshold), no recurrent history (FF), and Static-Control (Static in the plot). MCPool achieves

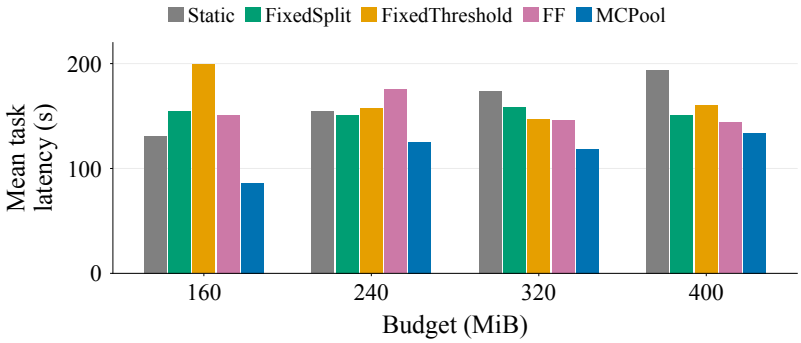

Figure 7: Mean task latency for MCPool and its ablations across memory budgets.

the lowest mean task latency across all tested budgets, with 1.23×–1.52× speedups over Static-Control. Admission provides the largest gain at 160 MiB, whereas allocation contributes the most at 320 MiB, where MCPool reduces latency by 25.4% relative to FixedSplit. Their varying importance across budgets highlights the benefit of MCP's jointly adapting allocation, admission, and recurrent workload information. Detailed results are in Appendix F.2.

Table 3: Mean task latency (s) across different orders.

Order	Static-Control	ValueGreedy	MCPool
Original	281.78	259.52	**213.93**
Order 1	277.51	333.68	**203.17**
Order 2	322.39	219.91	**174.84**
Order 3	318.31	341.92	**176.33**
Order 4	251.45	215.88	**186.78**

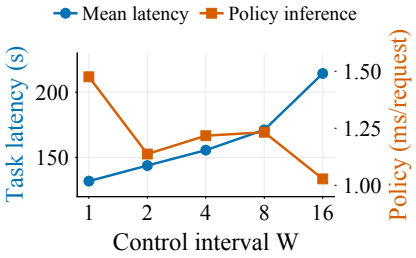

Figure 8: Control-frequency trade-off.

5.3.4 Sensitivity to Request Ordering and Control Frequency (RQ4)

Request Order Permutation Robustness. Table 3 evaluates policy resilience across the original task-arrival sequence and four reordered versions. MCPool consistently secures the lowest mean latency across all sequences (174.84–213.93 s), exhibiting minimal variance across order permutations. In contrast, baseline heuristics demonstrate extreme instability, ValueGreedy's latency fluctuates wildly between 215.88 s and 341.92 s, reversing its relative ranking against Static-Control depending on the request sequence. MCPool's recurrent state tracks changing arrival distributions in real time, ensuring robust performance advantages regardless of request ordering.

Control Frequency: Requests per Update (W). Figure 8 shows how the control interval W affects execution efficiency. As W increases from 1 to 16, mean task latency generally rises from 131.9 s to 214.3 s, while per-request policy inference overhead decreases overall from 1.48 ms to 1.03 ms. This reflects a trade-off between control overhead and responsiveness: larger intervals amortize policy inference over more requests but adapt more slowly to workload changes, whereas smaller intervals respond more frequently and achieve lower task latency at a modest inference cost.

6 Conclusion

We studied joint retention of MCP contexts and tool results as a partially observable sequential decision problem with coupled reuse benefits and a shared memory budget. MCPool combines recurrent control of memory allocation and admission thresholds with deterministic retained-memory enforcement, separating workload adaptation from capacity compliance. Across three foundation models on MCP-Bench, it achieves 1.34×–2.38× mean speedups and 5.6%–48.7% token savings over Static-Control, with zero observed budget violations and comparable task-performance scores among completed, successfully scored executions. Ablations show that the relative contributions of allocation, admission, and recurrent history vary with memory capacity, supporting their joint use. This framework could also extend to other settings where heterogeneous reusable states compete for limited memory, including LLM serving and serverless execution.

AI USE STATEMENT

Large Language Models (LLMs) were used solely to aid in the writing and polishing of the manuscript. LLMs, specifically ChatGPT, were employed exclusively as writing assistants in the preparation of this manuscript. Their role was limited to improving the presentation quality of the text, including tasks such as rephrasing sentences, correcting grammar, enhancing readability, and improving the overall flow of exposition. The use of LLMs was confined to linguistic refinement, and they were not involved in generating, verifying, or shaping any scientific ideas.

All research contributions, including the formulation of research questions, algorithmic design, and experimental studies, were conceived and executed entirely by the authors. Their contribution was restricted to stylistic and grammatical adjustments, with no bearing on the substance of the research.

The authors retain full responsibility for the entire content of this work, including any text improved with LLM assistance. We have carefully ensured that the usage of LLMs complies with ethical standards and does not introduce plagiarism, fabrication, or other forms of scientific misconduct.

REPRODUCIBILITY STATEMENT

The problem formulation and control method are described in Sections 3 and 4. Additional design and policy details are provided in Appendices C and D. Experimental settings and additional results are provided in Appendices E and F. Our implementation is available in an anonymous repository at `https://anonymous.4open.science/r/MCPool-release-23D5/`.

REFERENCES

Arash Ahmadi, Sarah Sharif, and Yaser M. Banad. MCP Bridge: A lightweight, LLM-agnostic RESTful proxy for Model Context Protocol servers. *arXiv preprint arXiv:2504.08999*, 2025.

Anthropic. Introducing the Model Context Protocol. Anthropic blog post, `https://www.anthropic.com/news/model-context-protocol`, 2024.

Farhana Begum, Craig Scott, Kofi Nyarko, Mansoureh Jeihani, and Fahmi Khalifa. Hierarchical caching for agentic workflows: A multi-level architecture to reduce tool execution overhead. *Machine Learning and Knowledge Extraction*, 8(2):30, 2026.

Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, 1997.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014.

Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. Model Context Protocol (MCP) at first glance: Studying the security and maintainability of MCP servers. *ACM Transactions on Software Engineering and Methodology*, 2026.

Matthew Hausknecht and Peter Stone. Deep recurrent Q-learning for partially observable MDPs. In *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*, pp. 29–37, 2015.

Jianmin Hu, Minxian Xu, Sa Wang, Chong Ma, Min Shen, Kejiang Ye, Lin Qu, and Chengzhong Xu. SwiftCache: Efficient LLM serving for multi-turn conversations with heterogeneous KV cache sharing. *arXiv preprint arXiv:2606.16135*, 2026.

Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

Jingyi Jia and Qinbin Li. AutoTool: Efficient tool selection for large language model agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 40, pp. 31265–31273, 2026. doi: 10.1609/aaai.v40i37.40389.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998. doi: 10.1016/S0004-3702(98)00023-X.

Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. RL-Cache: Learning-based cache admission for content delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML (NetAI)*, pp. 57–63, 2019.

Varad Kulkarni, Vaibhav Jha, Nikhil Reddy, Anand Eswaran, Praveen Jayachandran, and Yogesh Simmhan. Optimizing FaaS platforms for MCP-enabled agentic workflows. *arXiv preprint arXiv:2601.14735*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2026. URL https://arxiv.org/abs/2511.02230v7.

Anas Mohamed, Kaizan Haque, Azal Ahmad Khan, Chetan Sharma, Shuwen Ge, and Ali Anwar. Workload-aware caching for multi-agent systems. *arXiv preprint arXiv:2607.20495*, 2026.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8), 2025.

Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

Abhishek Vijaya Kumar, Bhaskar Kataria, Byungsoo Oh, Emaad Manzoor, and Rachee Singh. TVCache: A stateful tool-value cache for post-training LLM agents. *arXiv preprint arXiv:2602.10986*, 2026.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. MCP-Bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers. *arXiv preprint arXiv:2508.20453*, 2025.

Yi Zhai, Dian Shen, Junzhou Luo, and Bin Yang. ToolCaching: Towards efficient caching for LLM tool-calling. *arXiv preprint arXiv:2601.15335*, 2026.

Rui Zhang, Chaeeun Kim, Shaoting Feng, Kuntai Du, Yuhan Liu, Yi Zhong, Cheng-Wei Ching, Junchen Jiang, and Liting Hu. Learning agent execution for KV-cache management in agentic serving. *arXiv preprint arXiv:2608.14624*, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

A Related work

MCP and Tool-Call Caching. Recent work reduces repeated tool execution through caching. ToolCaching combines bandit-based admission with value-aware eviction of tool outputs (Zhai et al., 2026). TVCache supports stateful tool-value reuse during agent post-training through execution-prefix matching, with selective sandbox snapshots for state reconstruction (Vijaya Kumar et al., 2026). MCPool focuses on jointly allocating retained bytes to live MCP contexts and eligible tool results, with learned class-specific admission thresholds. Table 4 separates reuse targets from the specific shared-byte control studied here.

Table 4: Comparison of reuse targets and resource control. **Context**: retaining initialized MCP server processes or connections across requests. **Result**: tool or intermediate workflow outputs. Checks/crosses indicate modeled/not-modeled reuse targets. **Joint**: adaptive allocation and admission of live contexts and results under one retained-byte budget; "–" means this specific control is not reported. **Memory limit**: the resource bound used by each design; entry and sandbox counts are not byte-level limits.

Methods	**Reusable state**			**Resource control**		
	Context	**Result**	**KV**	**Approach**	**Joint**	**Memory limit**
ToolCaching (Zhai et al., 2026)	✗	✓	✗	Bandit/value	–	Cache entries
TVCache (Vijaya Kumar et al., 2026)	✗	✓	✗	Prefix/snapshot	–	Sandbox count
Workload-aware (Mohamed et al., 2026)	✗	✓	✗	DAG/cost score	–	Result-cache capacity
SwiftCache (Hu et al., 2026)	✗	✗	✓	GPU sharing	–	GPU memory
Continuum (Li et al., 2026)	✗	✗	✓	Cost-based TTL	–	GPU memory
CacheScout (Zhang et al., 2026)	✗	✗	✓	Online transitions	–	GPU KV blocks
MCPool (ours)	✓	✓	✗	**Recurrent RL**	✓	**Shared retained bytes**

Caching for Heterogeneous Execution Instances. Recent agent-serving caches make reuse decisions using workload structure, cost, and memory demand. Workload-aware caching combines recomputation cost, DAG dependencies, and agent frequency to retain intermediate results under capacity constraints (Mohamed et al., 2026). SwiftCache lets co-located heterogeneous models use spare GPU memory for KV storage; it shares storage capacity, not interchangeable KV tensors between models (Hu et al., 2026). Continuum retains KV state across tool execution using cost-based TTLs that account for reloading and queueing penalties (Li et al., 2026). CacheScout learns agent transitions online and uses predicted reuse to guide KV eviction and prefetching (Zhang et al., 2026). These are close precedents for history-aware retention and heterogeneous resource management beyond classical recency policies. However, they primarily manage a single type of cached entity or place such entities across heterogeneous memory resources. MCPool instead jointly manages heterogeneous reusable entities, including live MCP contexts and materialized tool results, under a unified retained-memory budget. These entities differ in their lifecycles, reuse granularities, and latency benefits. Its fixed-dimensional controller selects class-level allocation and admission, while a deterministic manager enforces validity and hard retained-memory capacity after each request. The main distinction is the joint management of heterogeneous reuse entities under a shared memory budget, rather than heterogeneous caching or online reuse prediction itself.

B Profiling Details

Repeated context initialization. Figure 9 shows that, in the profiled workload, increasing the number of switches from 1 to 19 increases context initializations from 2 to 20, while end-to-end latency rises from roughly 10 to 26 s. Repeated cold starts can therefore accumulate along a request sequence, motivating context retention beyond a single invocation.

Memory cost of warm contexts. The server-level measurements in Fig. 3 show that context footprints vary substantially across servers. The displayed examples range from approximately 70 MB for Met Museum to 180 MB for BioMCP. Figure 10 complements these individual measurements with the aggregate memory required to keep increasing numbers of servers warm. The full 28-server set requires over 1.85 GB, illustrating why retaining every context is incompatible with the smaller shared-cache budgets in our evaluation. The profiling figures use MB; experimental cache budgets use MiB. Together, these profiles expose the retention trade-off: reusing warm contexts avoids repeated acquisition, but keeping more MCP servers warm increases the aggregate memory footprint.

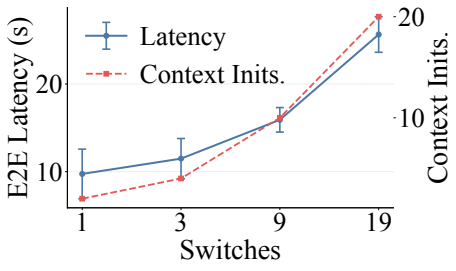

Figure 9: End-to-end latency and context initializations as the number of switches increases.

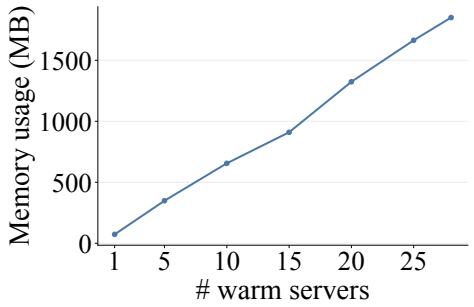

Figure 10: Aggregate memory usage as the number of warm servers increases.

C Additional Design Details

Smoothed reuse estimate. The manager maintains statistics for observed entities. For class q and entity k, the smoothed reuse estimate is

$$\widehat{p}^{q}_{t,k} = \frac{n^{q}_{t,k,\text{reuse}} + \alpha}{n^{q}_{t,k,\text{access}} + \alpha + \beta_0}, \tag{11}$$

where $\alpha, \beta_0 > 0$ are pseudocounts and accesses are counted independently of cache hits.

Log-scaled value density. The footprint denominator can give small tool results much larger raw value densities than server contexts. We apply the natural-log transform $\log(1+v)$ to the raw density v in (5), compressing this dynamic range while preserving zero and the ordering used for eviction. Admission compares the transformed score directly with the softplus threshold, and the observation averages transformed scores, rather than taking the logarithm of their average. This reduces scale disparity in both threshold comparisons and recurrent inputs; it does not equalize class distributions or guarantee selective admission at initialization.

C.1 Memory Accounting and Reclamation

The retained-memory budget B bounds the sum of charges of resident entities; it is not a bound on host memory, which additionally holds the temporary contexts of in-flight cold acquisitions and the host and agent runtime. The manager applies the following rules.

1. *Charge scope.* A context's charge covers its resident memory as measured by the host plus its index entry; a result's charge covers its serialized bytes plus its index entry. The manager's fixed-size statistics (reuse counters, smoothed estimates, per-class observations) are runtime memory

outside B, as is a newly created context until it is admitted; a rejected context is closed after use. For a remote HTTP server, only the locally retained client and session state is charged; the server process runs elsewhere and its memory is outside B.

2. *Conservative charge.* A context's charge is fixed at admission as its measured footprint plus a headroom, and the manager re-measures the context after the request completes. If the footprint exceeds the charge, the manager raises the charge to the new footprint plus headroom, records a charge overrun, pauses admissions to the class, and evicts entries in ascending value until the class is within its quota. A result's charge is its serialized size and does not change.
3. *Sequential decisions.* Cache decisions execute after the current request completes and before the next begins; all resident entities are available for eviction, and each admission checks its charge against the remaining class quota before proceeding.
4. *Logical removal before physical reclamation.* Evicting or invalidating an entity removes it from lookup immediately. Its charge is released only after physical reclamation completes: the context process has terminated, its connection has been closed, or the cached result has been freed. Until then the entity remains in $\mathcal{E}_t^q$ and in M_t. The manager completes reclamation before admitting replacements or applying a new split.

C.2 Retained-Memory Feasibility

Proposition 1 (Pathwise retained-memory feasibility)**.** *Let $M_t^q = \sum_{k \in \mathcal{E}_t^q} m_k^q$ and assume (i) initially $M_t^q \le B_d^q$ for every class; (ii) each charge is at least the entity's retained footprint while it is resident; (iii) cache decisions execute sequentially after the current request completes and before the next begins, when every resident entity is available for eviction; and (iv) evicted or invalidated entities remain charged until reclamation completes. Under the manager's rules (class-wise eviction to the new quotas before applying a split, quota-checked admission within the candidate's class, and no cross-class borrowing), $M_t^q \le B_d^q$ for every class, and hence $M_t \le B$, after every manager event and for any sequence of policy actions.*

Feasibility is thus a property of the manager, not of learned policy quality; a charge overrun, which violates (ii), is detected at re-measurement and repaired by eviction (Appendix C.1).

Proof. Let B^q denote the applied quota of class q, with $\sum_q B^q = B$. We show by induction on manager events that $M_t^q \le B^q$ holds for every class after each event; $M_t \le B$ follows by summation. The invariant holds initially by assumption (i). Suppose it holds before an event.

Admission. A candidate k of class q is admitted only if $M_t^q + m_k^q \le B^q$; sequential processing (iii) ensures that this check uses the current occupancy, and m_k^q covers the entity's retained footprint (ii). No other class changes, so the invariant holds afterwards.

Eviction. An evicted entity leaves lookup at once and leaves $\mathcal{E}^q$ only when its reclamation completes; its charge is counted until then (iv) and no admission uses that charge before it is released. Hence M^q never increases through eviction.

Quota change. For each new class quota $\tilde{B}^q$, the manager evicts residents in ascending value until $M^q \le \tilde{B}^q$. Every resident is available for eviction under (iii), so this requires at most reclaiming all residents of that class. The manager completes these evictions before applying the new split and resuming admissions; occupancy remains within the old quotas during eviction and within the new quotas after the switch.

Invalidation. An invalidated entry leaves lookup and is evicted as above, remaining charged until reclamation completes (iv); M^q does not increase.

Re-measurement and reclamation. The manager re-measures context footprints after request completion. If a footprint exceeds its charge, assumption (ii) has been violated and the manager repairs the violation as in Appendix C.1. Reclamation releases a charge and decreases M^q.

Each case preserves $M^q \le B^q$. The argument uses only the manager's rules and assumptions (i)–(iv); it does not depend on how the policy chooses a_d, whether by a recurrent, feed-forward, static, or random actor. Since $\sum_q B^q = B$, assumption (i) also gives $M_t \le B$ initially. □

D Additional Policy Details

D.1 Training Objective and Recurrent Updates

The critic fits the return target $\widehat{R}_d$ using

$$\mathcal{L}_V = \widehat{\mathbb{E}}_d \left[(V_\psi(h_d) - \widehat{R}_d)^2 \right]. \tag{12}$$

Together with the PPO surrogate in (10), the complete training loss is

$$\mathcal{L} = -\mathcal{L}_{\text{clip}} + \kappa_V \mathcal{L}_V - \kappa_H \widehat{\mathbb{E}}_d \left[\mathcal{H}(\pi_\theta(\cdot \mid h_d)) \right], \tag{13}$$

where κ_V and κ_H weight value estimation and entropy regularization. GAE assigns later latency feedback to earlier resource controls (Schulman et al., 2016; 2017). The discounted interval return is the training surrogate; it is not generally identical to the long-run per-request average latency in (1).

Training preserves the temporal order of control transitions. Recurrent updates use contiguous subsequences and their initial hidden states, with padding excluded from the loss. Task boundaries remain nonterminal transitions: recurrent state and GAE credit assignment continue across tasks within a workload sequence. Online access and cost estimates continue to update from observed requests. The same deterministic manager enforces the retained-memory budget during training and evaluation.

D.2 Control Procedure

Algorithm 1 summarizes continuous resource control across task arrivals within a workload sequence.

Algorithm 1 Joint Allocation and Admission Control

Input: Policy π_θ, critic V_ψ, manager, budget B, request count W

Initialize observations, recurrent state, and previous feedback at the start of an independent sequence.
for each control epoch d **do**
  Obtain $a_d = (\beta_d, \theta_d^c, \theta_d^o)$ using (4).
  Between requests, evict residents class-wise in ascending value to meet the new quotas, then apply the split and admission thresholds (Section 4.1).
  for the next W requests, continuing across tasks, or until the sequence ends **do**
    Serve an eligible result hit; otherwise acquire a context and execute the tool.
    Record L_t and update observed access and cost statistics.
    After the request completes, score candidates and apply eligibility, threshold, and quota rules within each candidate's class.
  end for
  Form o_{d+1} and reward r_{d+1} using (8).
  Update recurrent history using (7); preserve it across task boundaries.
  if training and the rollout is full **then**
    Compute GAE targets and update actor and critic using (13).
  end if
end for

E Detailed Experimental Settings

Models, Workloads, and Phase Compositions. Main comparison experiments evaluate three foundation models (gpt-5.4-mini, claude-sonnet-5, and gpt-5.5), with gpt-5.4-mini serving as the default for in-depth analyses. All source tasks are derived from MCP-Bench (Wang et al., 2025) (28 MCP servers, 250 tools) and execute live against real endpoints under a default retained-cache budget of 320 MiB and a control interval of $W = 8$ requests. Training and evaluation workloads utilize strictly disjoint source-task sets:

- **Training Workload (250 arrivals, 52 tasks):** Synthesized via four structured rounds into six distinct operational phases: 30 exact-repeat, 120 short-switching, 30 initial long-gap anchor, 30 anchor revisit, 31 distractor, and 9 volatile-data arrivals (negative cases for result reuse).
- **Evaluation Workload (50 arrivals, 26 held-out tasks):** Generated by hashing representative task records using seed `2027` and cycling until 50 arrivals are reached (24 tasks occur twice; 2 occur once). The trace spans 27 single-server, 11 two-server, and 12 three-server arrivals across 26 unique server combinations and 24 target servers.

While each evaluation sequence starts with an empty cache, retained session state, recurrent neural history, and request counters persist across consecutive task boundaries.

Task Orderings, Locality, and Scaling Workloads. Order experiments preserve the same 50-arrival evaluation multiset while permuting sequence arrival order across five configurations: Order 0 (original), Orders 1–2 (shuffled via seeds `2028` and `2029`), Order 3 (sorted by target server combination), and Order 4 (grouped into server-combination buckets and drawn round-robin). All baselines receive identical arrival permutations, and internal tool dependencies are strictly preserved. To thoroughly evaluate performance bounds, two synthetic workload suites are constructed:

- **Locality Stress Tests (250 arrivals each):** (1) *Bursty Reuse*: 5 hot tasks executed in 50 blocks of 5 consecutive repetitions; (2) *Hotspot Shift*: abruptly switching from one 5-task hot set at arrival 125 to another; and (3) *Low Result Reuse*: cycling through 26 test tasks while appending a unique marker to each arrival's description to alter input contexts.
- **Entity Scaling Workloads (500 arrivals each):** (1) *Context Scaling*: covers 8, 16, or 24 distinct servers using 3, 7, or 14 distinct tasks; and (2) *Result Scaling*: cycles through 16, 32, or 64 query topics over a two-server combination. These counts define macro query diversity, whereas actual result-cache keys depend on granular tool names and arguments.

Performance Evaluation Judge. Task completion quality is evaluated using an independent judge, gemini-3.7-flash, with one evaluation per completed task execution. The evaluator assesses the complete execution trace—including initial prompts, tool specifications, execution summaries, intermediate steps, and final outputs—and assigns objective scores on a 1–10 scale across two core dimensions: Task Fulfillment (constraint satisfaction and goal achievement) and Grounding (factuality relative to retrieved tool evidence). Reported performance scores reflect the arithmetic mean over successfully scored arrivals, with repeated arrivals contributing individually.

Table 5: RL hyperparameters and default resource-control settings.

Parameter	Value	Parameter	Value
Learning rate	3×10^{-4}	Discount γ	0.99
GAE λ	0.95	Rollout transitions	32
Sequence length	8	PPO epochs	3
Clip range	0.2	Entropy coefficient	0.01
Value coefficient	0.5	Maximum gradient norm	0.5
GRU hidden dimension	64	Target KL	0.05
EWMA coefficient	0.1	Reference latency	1 second
Control interval	8 requests	Cache budget	320 MiB

E.1 RL Hyperparameters

Table 5 lists the training and control settings. Each PPO update uses 32 control transitions, recurrent subsequences of length eight, and up to three optimization epochs, with target-KL early stopping.

F Supplementary Experiments

F.1 Reward Convergence

Figure 11 shows the latency reward over training decision steps. The moving average approaches zero after the initial fluctuations and remains near that level for most later steps, with occasional downward excursions.

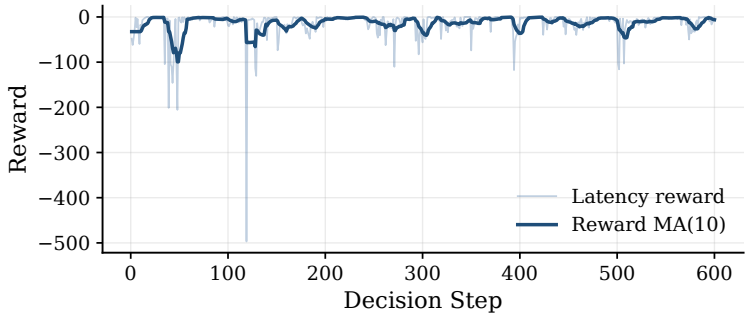

Figure 11: Latency reward during training and its 10-step moving average.

Table 6: Detailed latency and token ablation across memory budgets (MiB). Speedup uses Static-Control as the baseline at each budget; token reduction compares each variant's token count with MCPool's.

Budget	Variant	Latency (s)	Tokens/task	Speedup	Token reduction
160	**MCPool**	86.25	30,830.17	1.52×	1.00×
	FF	151.17	30,400.50	0.86×	0.99×
	FixedSplit	154.51	31,104.00	0.85×	1.01×
	FixedThreshold	199.16	30,819.33	0.66×	1.00×
	Static-Control	130.75	22,702.67	1.00×	0.74×
240	**MCPool**	125.27	19,713.17	1.23×	1.00×
	FF	175.34	29,984.83	0.88×	1.52×
	FixedSplit	151.07	22,193.00	1.02×	1.13×
	FixedThreshold	157.49	30,392.33	0.98×	1.54×
	Static-Control	154.23	35,170.80	1.00×	1.78×
320	**MCPool**	118.33	21,512.90	1.47×	1.00×
	FF	146.50	30,580.83	1.19×	1.42×
	FixedSplit	158.70	31,324.17	1.09×	1.46×
	FixedThreshold	147.47	29,441.50	1.18×	1.37×
	Static-Control	173.76	31,680.00	1.00×	1.47×
400	**MCPool**	133.53	21,446.56	1.45×	1.00×
	FF	143.92	30,801.50	1.35×	1.44×
	FixedSplit	151.21	21,887.00	1.28×	1.02×
	FixedThreshold	160.40	32,495.17	1.21×	1.52×
	Static-Control	193.82	31,117.17	1.00×	1.45×

F.2 Detailed Component Ablation

Table 6 reports mean task latency and tokens per task for gpt-5.4-mini on the 50-arrival evaluation workload with $W = 8$. FixedSplit fixes the memory allocation, FixedThreshold fixes admission thresholds, and FF removes recurrent history. Latency speedup is Static-Control's latency divided by the listed variant's latency at the same budget; values above $1\times$ indicate faster execution than Static-Control. Token reduction is the variant's token count divided by MCPool's token count, with values above $1\times$ indicating fewer tokens for MCPool.

MCPool achieves the lowest mean task latency at all four budgets, with $1.23\times$–$1.52\times$ speedups over Static-Control, consistent with Fig. 7. The largest latency reduction among the three component ablations comes from adaptive admission at 160 MiB (56.7% relative to FixedThreshold) and recurrent history at 240 MiB (28.6% relative to FF). At 320 MiB, adaptive allocation provides the largest gain (25.4% relative to FixedSplit); at 400 MiB, admission again contributes the most (16.8% relative to FixedThreshold). This changing ranking supports jointly adapting allocation, admission, and recurrent workload information across memory budgets.

The token results show joint latency and token savings at 240–400 MiB. At 240 MiB, MCPool uses 19,713.17 tokens per task, reducing usage by 11.2%–35.1% relative to the three ablated variants and 44.0% relative to Static-Control. At 320 and 400 MiB, it uses 26.9%–31.3% and 2.0%–34.0% fewer tokens, respectively, than the three ablated variants. Across these budgets, restoring all components therefore improves latency without increasing token usage. At 160 MiB, token usage remains within 1.4% of the ablated variants, but is 35.8% higher than Static-Control's. Token savings thus depend on the memory budget, while MCPool maintains the lowest latency across all tested budgets.

F.3 Sensitivity to Workload Locality

Figure 12 shows that MCPool sustains approximately $1.3\times$ speedup over Static-Control in all three workloads, whereas ValueGreedy falls below the baseline under bursty reuse and provides smaller gains in the other two conditions. The advantage under a hotspot shift is consistent with adapting retention to changing demand; the gain under low result reuse also supports the value of retaining reusable contexts when exact-result hits are scarce. These aggregate results show robustness across locality conditions.

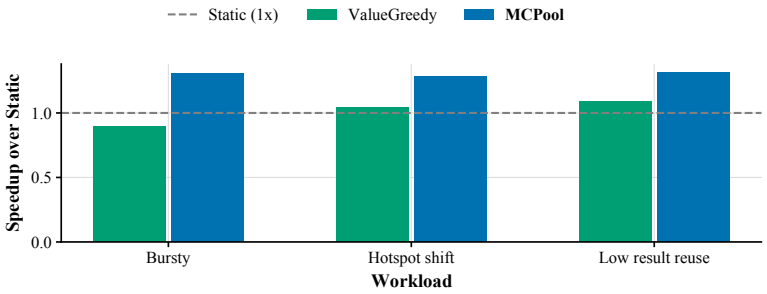

Figure 12: Speedup relative to Static-Control (Static in the plot) under bursty reuse, a hotspot shift, and low result reuse.

F.4 Robustness to Entity Population

Table 7 varies the number of target servers or query topics in the fixed-budget setting. The reported budget is 320 MiB.

Table 7: Mean task latency (s) as the entity population varies. Bold marks the lowest observed latency in each row.

Entity type	Count	MCPool	Static-Control	ValueGreedy
Servers	8	**142.01**	142.94	148.21
Servers	16	**107.02**	115.11	119.03
Servers	24	**114.69**	118.44	138.77
Query topics	16	35.46	**33.39**	33.43
Query topics	32	**33.21**	36.57	34.37
Query topics	64	**33.22**	33.29	33.26

For context scaling, MCPool achieves the lowest latency at all three server counts, reducing latency by 0.6%–7.0% over Static-Control and 4.2%–17.4% over ValueGreedy. The advantage over Val-

ueGreedy is largest with 24 servers, where more contexts compete for the same retained-memory budget. The non-monotonic latency across server counts also indicates that entity population alone does not determine workload difficulty; reuse patterns and context acquisition costs remain important.

The benefit is less consistent when scaling query diversity. MCPool performs best at 32 topics, reducing latency by 9.2% over Static-Control and 3.4% over ValueGreedy. At 64 topics, the three methods differ by less than 0.1 s, indicating little benefit from adaptive control in this setting. At 16 topics, MCPool is 6.2% slower than Static-Control and 6.1% slower than ValueGreedy. This suggests that dynamic control is most useful when the workload provides meaningful retention choices; when those choices are limited or the policies make similar retention decisions, its advantage can diminish or disappear. Overall, MCPool is robust to increasing context populations, while its gains under result-entity scaling depend more strongly on the available reuse opportunities.