# 💸 💰 🤑 Cache Saver: A Modular Framework for Efficient, Affordable, and Reproducible LLM Inference

**Nearchos Potamitis** [*1] **Lars Klein** [*2] **Chongyang Xu** [3] **Attreyee Mukherjee** [3] **Bardia Mohammadi** [3]
**Niket Tandon** [4] **Laurent Bindschaedler** [3] **Akhil Arora** [1]

## Abstract

Inference constitutes the majority of costs throughout the lifecycle of a large language model (LLM). While numerous LLM inference engines focusing primarily on low-level optimizations have been developed, there is a scarcity of non-intrusive client-side frameworks that perform high-level optimizations. In this paper, we introduce Cache Saver, a modular, plug-and-play, and asynchronous framework that facilitates high-level inference optimizations, thereby integrating cleanly into existing systems without requiring changes to the end-user application logic or the underlying LLM. The key novelty is a *namespace-aware list-valued cache* that ensures *statistical integrity* of LLM responses by generating *i.i.d.* responses within a namespace as well as ensuring *reproducibility*. Moreover, as a direct consequence of operating at a high level, Cache Saver supports both local and online models. We conduct extensive experiments with five representative state-of-the-art reasoning strategies, five diverse benchmark tasks, and three different LLMs. On average across all methods, tasks, and LLMs, Cache Saver reduces cost by $\simeq$ 25% and $CO_2$ by $\simeq$ 35%. Notably, Cache Saver excels in practical machine learning scenarios such as benchmarking across multiple methods or conducting ablation analysis of a specific method, obtaining substantial cost and carbon footprint reduction of $\simeq$ 60%. Cache Saver is publicly available at `https://github.com/au-clan/cachesaver`.

[*]Equal contribution [1]Aarhus University [2]EPFL [3]MPI-SWS [4]Microsoft Research. Correspondence to: Nearchos Potamitis <nearchos.potamitis@cs.au.dk>, Lars Klein <lars.klein@epfl.ch>, Akhil Arora <akhil.arora@cs.au.dk>.

*Proceedings of the 42nd International Conference on Machine Learning*, Vancouver, Canada. PMLR 267, 2025. 

## 1. Introduction

Large language models (LLMs) have taken the world by storm. Most mainstream web applications (e.g., Facebook, Twitter/X, WhatsApp, Reddit, Google/Bing Search, etc.) now offer some form of LLM-based companion or assistant. Not surprisingly, LLMs are estimated to account for $\simeq$ 2% of global electricity consumption and greenhouse emissions (Crawford, 2021), a figure projected to rise to $\simeq$ 8% by 2030 (IEA, 2025). Most costs in an LLM's lifecycle come from inference (Fu et al., 2024), which is expensive, especially with the rise of autonomous agents (Fan et al., 2022), test-time scaling (Muennighoff et al., 2025), multi-step reasoning strategies (Yao et al., 2024; Klein et al., 2024; Shinn et al., 2023; Hao et al., 2023), and native reasoning models (OpenAI, 2024). For instance, OpenAI's o3 costs $\simeq$1000$ per task on the ARC-AGI benchmark (Chollet et al., 2019).

**Existing work and challenges.** To address this issue, numerous LLM inference engines–most notably vLLM (Kwon et al., 2023c)–have been developed in recent years. However, owing to their focus on low-level optimizations such as efficient key-value (KV) caching and memory management (Kwon et al., 2023b; Ainslie et al., 2023; Park et al., 2025), it is non-trivial to leverage these engines for novel applications/LLMs without either modifying the application logic or the LLM or both. Notably, these engines cannot be used with online API-based LLMs. While recent work on client-side caching (Bang, 2023; Helicone; LangChain) is a step in the right direction, these approaches are typically limited to generic semantic matching, and *lack guarantees for reproducibility, statistical integrity*, or support for complex experimental workflows. Moreover, since in many practical application scenarios, such as stochastic sampling, uncertainty estimation, or ensuring policy diversity, reasoning strategies require *multiple independent responses to the same prompt*, a naïve KV cache that maps each unique prompt to a single LLM response is undesirable. Please see Appx. A for additional details on related works.

**Reuse potential.** To better illustrate the reuse potential, consider a math puzzle solved step-by-step using a tree search.

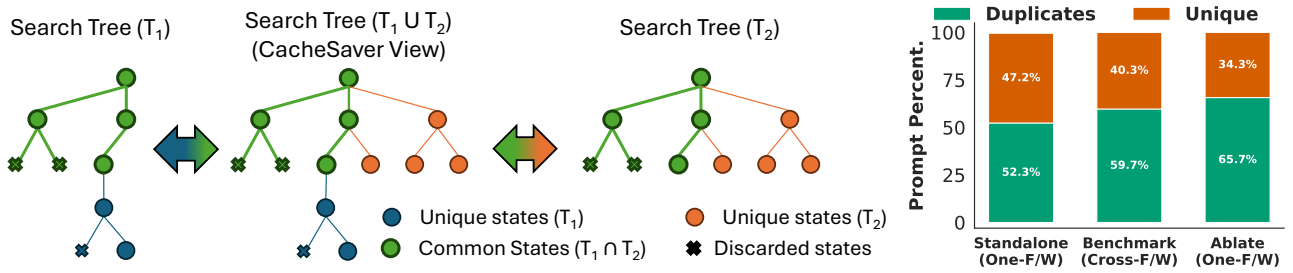


Figure 1: (Left) Search trees $T_1$ and $T_2$ corresponding to independent executions of two search strategies, and $T_1 \cup T_2$, the combined search tree as seen by CACHE SAVER recognizing the reuse potential through the common states. (Right) Analyzing the prompt redundancy for three different CACHE SAVER configurations across 5 representative SOTA methods and tasks. **On average, CACHE SAVER results in 21%, 36%, and 45% cost savings, respectively.**

Each node represents a state in the solution trajectory, and a prompt asks the LLM to suggest the next step. While a breadth-first search (BFS) may try to discover all child nodes of a current node by repeatedly calling the next-step prompt and aggregating suggestions, a depth-first search (DFS), by contrast, will commit to one path and go deep, expanding it repeatedly before backtracking. Both strategies use the same prompts and may visit similar regions of the state space. It is necessary for their samples to be independent within each search, so that repeated queries for the next step can discover branches in the search tree. However, to compare these two strategies fairly, it is important that they explore the same graph, i.e., when they encounter the same state, they receive the same samples. This is a form of statistical coupling and can be achieved by reusing samples across frameworks. Fig. 1(Left) shows for toy data how two tree-search strategies ($T_1$ and $T_2$) may be coupled to explore the same tree, and how the partial views onto the search space are assembled into a shared latent tree ($T_1 \cup T_2$) in the cache. Further, Fig. 1(Right) presents the real-world reuse potential by showcasing average prompt redundancy across 5 representative state-of-the-art (SOTA) methods and tasks. Notably, we observe **substantial duplicates $\simeq$50–65%** both during executions of a single method (Bars 1 and 3), which is expected owing to similarity across variations of a single method, and **across methods (Bar 2)**, which is an *interesting and novel finding* of our work.

**Present work (CACHE SAVER).** We implement CACHE SAVER (Fig. 2) as a set of pluggable, composable components that together form a modular request pipeline between the user and the LLM without imposing architectural constraints or design patterns. At its heart is a *fundamentally novel caching paradigm* operationalized using a *list-valued cache* and managed using the concept of *namespacing* to control how and where samples may be reused. Within a namespace, samples are guaranteed to be *i.i.d.*, whereas across namespaces, samples may be reused. An implementation of Fig. 1(Left) would rely on two namespaces "T1" and "T2". Beyond the novel cache, CACHE SAVER also substantially benefits by managing incoming queries with minimal redundancy and maximum efficiency (*batcher and deduplicator*), ensuring deterministic request handling and consistent output ordering for reproducibility (*reorderer*), and precise tracking of prompt-response mappings (*cacher*), enabling controlled experimentation and more reliable benchmarking across diverse configurations. On average, CACHE SAVER results in 21%, 36%, and 45% cost savings, respectively, across the three scenarios presented in Fig. 1(Right).

**Contributions.**

- We convincingly motivate (both intuitively and empirically) the existence of a substantial prompt reuse potential in both intra- and inter-framework settings (§ 1 and § 3).
- We propose CACHE SAVER, a modular, plug-and-play, and asynchronous framework for efficient and reproducible LLM inference. Additional advantages include low memory overhead owing to a disk resident cache and its extendibility: reasoning strategies can be added largely unchanged (§ 2).
- Powered by structured namespace-aware caching, ours is the first work to enable response reuse in LLMs without sacrificing the statistical integrity of the generative LLM (§ B.1).
- We conduct extensive experiments on five reasoning strategies[1] and benchmark tasks using three LLMs as base models. On average across all methods, tasks, and LLMs, CACHE SAVER reduces cost by $\simeq 25\%$ and $CO_2$ by $\simeq 35\%$. Notably, CACHE SAVER excels in performing practical tasks such as benchmarking or ablation analysis, obtaining substantial cost and carbon footprint reduction of $\simeq 60\%$.

## 2. CACHE SAVER

**Overview.** CACHE SAVER is a modular framework composed of three key modules: (1) a *Batcher* (§ B.2), (2) a *Deduplicator* (§ B.2), and (3) a *Cacher* (§ B.1). Fig. 2

[1] Single-step reasoning methods such as IO, CoT, CoT-SC, have no reuse potential as there are no repetitions, and as such, they cannot benefit from a framework like CACHE SAVER.

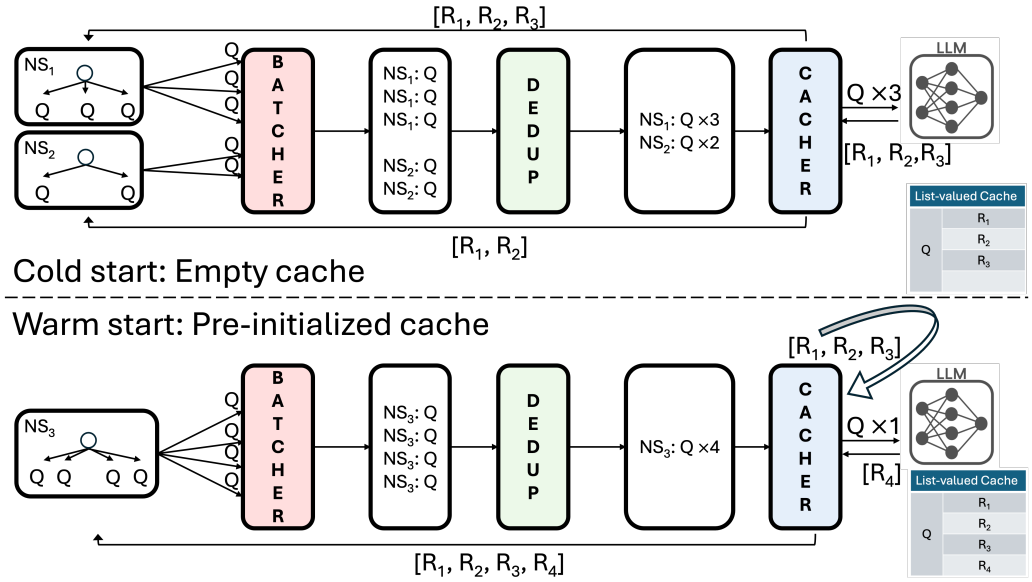


Figure 2: Overview of our CACHE SAVER framework: (Top) Cold-start and (Bottom) Warm-start.

presents an overview of its inner workings using a toy example, where three search strategies, independently exploring a dynamic search tree in their own namespace ($NS_1$ through $NS_3$), request responses for the same prompt $Q$. Fig. 2 (top) illustrates the *cold-start* scenario, where the cache is empty. The asynchronous *batcher* collates the five incoming requests into a single batch and passes it to the *deduplicator*, which groups identical prompts for each namespace and emits two aggregate requests, requesting three and two *i.i.d.* responses to $Q$ for $NS_1$ and $NS_2$, respectively. The *cacher*, which relies on a system of *asynchronous mutexes* to avoid redundant or overlapping requests, receives these two aggregate requests and sends a single request to the LLM, asking for three responses to $Q$. Finally, the LLM responses ($[R_1, R_2, R_3]$) are stored in the cache and used to resolve the requests from $NS_1$ and $NS_2$. In the *warm-start* scenario (Fig. 2 bottom), an aggregate request asking for four *i.i.d.* responses to $Q$ is triggered from $NS_3$, however, since three responses are already cached, the *cacher* asks for one additional response from the LLM and stores it in the cache. The four requests are then resolved by serving ($[R_1, R_2, R_3]$) from the cache along with the newly generated response ($R_4$). For additional details ((detailed method descriptions, pseudocode, practical considerations: cache eviction, consistency, etc.), please see Appx. B.

## 3. Experiments

We assess the effectiveness of CACHE SAVER through extensive experiments and analyses comprising 6 reasoning strategies, 5 benchmark tasks, and 3 LLMs. Additional details, e.g., implementation, hyperparameters, additional results, etc. are presented in Appx. C. The resources for reproducing our experiments are available at `https://github.com/au-clan/cachesaver`.

### 3.1. Setup

**Base model.** We use GPT4.1-Nano as the base model for the main results presented in this paper. To showcase the *generalizability* of our findings, we report results with other base models, namely, Llama4-Scout and Claude3.5-Haiku in the Appendix. While Llama4-Scout is run locally on a machine with 8 H200 GPUs, an AMD EPYC 9555 64-Core Processor, and 2TB of RAM, experiments with other LLMs were performed via API calls to their respective online platforms.

**Number of runs.** For § 3.2, 3.3, and § C.6.1, we run each experiment 10 times and report both mean and standard error of the evaluation metrics. For cost reasons[2], other experiments were only run once.

**Prompts.** To ensure a *fair* assessment of the benchmarked reasoning strategies, we *reuse* the prompts provided by the existing methods. For cases where there are no existing prompts, e.g., novel tasks or base LLMs, we adapt the original prompts provided by the methods For details, please see Appx. C.4.

**Tasks and data.** We conduct experiments on a judicious mix of 5 benchmark tasks that require a variety of reasoning, planning, and general problem-solving skills. Our

[2]Since we report results for multiple reasoning strategies, tasks, and base LLMs, costs blow up owing to a combinatorial explosion; thus, experiments crucial for the main takeaways were prioritized for multiple runs.

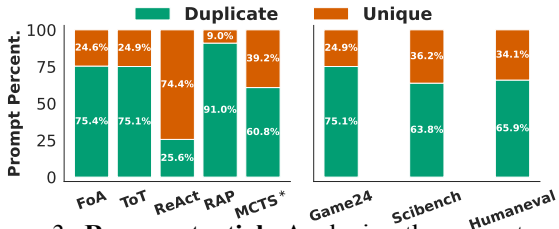

Figure 3: **Reuse potential:** Analyzing the prompt redundancy for (Left) each method by averaging over all tasks and (Right) each task by averaging over all methods.

tasks span diverse application domains: (1) mathematical reasoning: Game of 24 (Yao et al., 2024), (2) coding: HumanEval (Chen et al., 2021), (3) question answering: HotpotQA (Zhilin et al., 2018), (4) scientific reasoning: SciBench (Wang et al., 2024a), and (5) creative writing: Shakespearean Sonnet Writing (Suzgun & Kalai, 2024). For evaluation, we use the test sets as provided in the original benchmarks. For additional details, please see Appx. C.1.

**Reasoning strategies.** We conduct experiments with 5 representative SOTA reasoning strategies: (1) React (Yao et al., 2023), (2) ToT (Yao et al., 2024), (3) RAP (Hao et al., 2023), (4) ReST-MCTS (Zhang et al., 2024), and (5) FoA (Klein et al., 2024). We only include methods that have made their code available for at least one task benchmarked in this study. Thus, we exclude GoT (Besta et al., 2024), TouT (Mo & Xin, 2024), and RecMind (Wang et al., 2024b). Moreover, we exclude BoT (Yang et al., 2024), where although the code is available, an important resource (the meta-buffer) to reproduce their results is unavailable. We exclude LATS (Zhou et al., 2024) owing to its exorbitant cost footprint. Finally, owing to their *lack of reuse potential*, we exclude all *single-step* reasoning strategies such as IO prompting, CoT (Wei et al., 2022), CoT-SC (Wang et al., 2023), and AoT (Sel et al., 2024). For details, please see Appx. C.2.

**Evaluation metrics.** We assess the efficacy: *Quality*, efficiency: *Latency, Throughput, #Tokens, and Running Time*, and *cost*. For API-based LLMs, we report the cost (in USD), whereas for locally hosted LLMs, we report the energy consumption (in kWh) and the carbon footprint ($CO_2$ emissions in grams) measured using Carbontracker (Anthony et al., 2020). For details, please see Appx. C.3.

### 3.2. Basis for Cache Saver Effectiveness

Fig. 3 shows the reuse potential using GPT4.1-Nano as the base LLM by analyzing the percentage of duplicate prompts across all tasks and reasoning strategies benchmarked in this study. Results with Llama4-Scout are similar and are therefore presented in the Appendix. It is evident that overall $\simeq$ 50% prompts are duplicates, which implies that there exists a large overall reuse potential, which is not an artefact of a particular reasoning strategy or benchmark task or base LLM. Moreover, Fig. 3(Left) further shows that while all methods possess a similar number of duplicate prompts, React reports a substantially low reuse potential. This is largely expected as, despite being an iterative strategy, React only performs 2 retrials, which is consistent with conventions in the literature (Shinn et al., 2023). Thus, more retrials should result in a larger reuse potential. On the other hand, Fig. 3(Right) does not show any aberrations.

### 3.3. Statistical integrity of Cache Saver

Figs. 4(a)-(b) shows the quality across all tasks and reasoning strategies benchmarked in this study using GPT4.1-Nano and Llama4-Scout, respectively. We report the average and standard error over 10 independent runs. It is clear that the quality values with and without Cache Saver are statistically indistinguishable (overlapping intervals), which further provides strong empirical validation to our claim (by construction) regarding the statistical integrity of Cache Saver (§ B.1). At the same time, Figs. 4(c)-(d) portray substantial cost ($\simeq$25%) and carbon emission ($\simeq$35%) savings, respectively, with the biggest improvements achieved for RAP while the least for React, which is consistent with the findings from Fig. 3(Left). It is important to note that while the absolute values (e.g., cost in Fig. 4c) might appear small, which is just due to the extremely low cost of GPT4.1-Nano, we report relative percentage savings. In fact, we conducted the same experiment with GPT4.1, and found that RAP required 34.87 and 12.56 US$ with and without Cache Saver, respectively, roughly portraying a similar savings as for GPT4.1-Nano.

## 4. Discussion and Concluding Insights

### 4.1. Summary of Findings

**Reasoning strategies portray $\simeq$ 50% prompt redundancy.** We show, both intuitively (§ 1) and empirically (§ 3.2), the existence of substantial prompt reuse potential across five representative SOTA reasoning strategies and diverse benchmark tasks.

**Namespaced caching preserves statistical integrity of LLM responses.** We convincingly present the statistical correctness of Cache Saver by construction (§ B.1) and experiments (§ 3.3).

**Other advantages.** Cache Saver **saves up to 60%** cost and carbon emissions of LLM reasoning strategies across a variety of reasoning strategies and benchmarks. Moreover, Cache Saver does not possess any memory overhead, is plug-and-play, and easily extendible to new reasoning strategies or benchmark tasks.

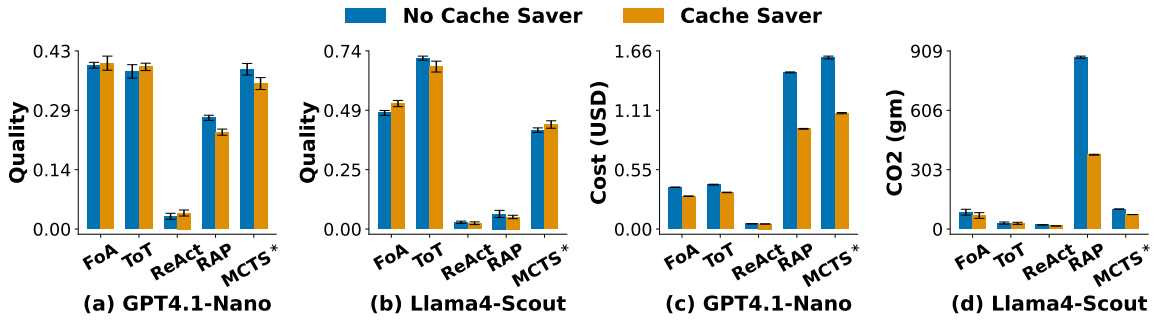


Figure 4: Comparing the (a)-(b) quality, (c) cost (US$), and (d) $CO_2$ (gm) for each method with and without CACHE SAVER, using GPT4.1-Nano (OpenAI API) and Llama4-Scout (deployed locally) as base LLMs, respectively.

### 4.2. Implications and Broader Impact

**Integrity in Experimentation.** We have introduced a principled approach to LLM evaluation by enforcing consistent seeding and input ordering, enabling statistically sound and reproducible experimentation.

**Environmental Efficiency.** We have demonstrated that avoiding redundant computation leads to substantial reductions in energy usage and carbon emissions, contributing to more sustainable LLM experimentation.

**Accessibility, Reproducibility, and Collaboration.** We have designed the cache to be publicly shareable with the broader ML community, particularly benefiting researchers conducting applied work with LLMs. This facilitates low cost follow-up studies, ensures reproducibility of benchmark results, and significantly accelerates new community-driven research by making shared resources easily accessible.

## Limitations

Currently, we evaluate each method in a plug-and-play fashion; however, the implementations could be further optimized to fully leverage the capabilities of the CACHE SAVER framework. Additionally, we could extend our workflows to optimize multi-step pipelines by optimizing and then leveraging the parallelizatino between independent steps.

While the CACHE SAVER framework is designed to be modular and compatible with existing inference engines, optimizations within our system may not always align with those in other frameworks. For example, caching strategies or batching heuristics used by CACHE SAVER may conflict with the parallelism, memory management, or scheduling decisions employed by underlying inference systems. At present, we do not focus on co-optimization across such system boundaries.

Our caching mechanism currently relies on exact matches, which limits reuse when inputs vary slightly. In future work, we aim to explore more advanced strategies. Fuzzy caching could support approximate or semantic matches to increase hit rates. Cascading caches across memory tiers (e.g., RAM, SSD, distributed) could help balance latency and cost. Sparsity-aware caching could store only the most relevant context fragments, especially for long prompts. Additionally, improving cache observability—through tools for hit/miss analysis, error tracing, and adaptive tuning—will be essential for enhancing performance and debugging.

Because the framework operates at the user level, our current focus is on high-level orchestration rather than hardware- or system-level optimization. Future work will explore low-level improvements such as a hardware-optimized batcher that groups requests by context length or cache affinity to improve GPU utilization. We also plan to implement prefetching strategies that proactively load or generate likely-needed cache entries, thereby reducing latency and improving responsiveness.

## Ethics statement

In our opinion, this work has no major ethical considerations. All the datasets and resources used in this work are publicly available and do not contain any private or sensitive information about human subjects. Moreover, we use standard and vetted benchmark datasets following the corresponding licensing and fair use terms and conditions. Finally, the research presented in this paper does not involve any interactions whatsoever with human subjects. That said, and similar to all other LLM-based research, our work does have a negative impact on the environment, in particular by contributing to the stark rise in greenhouse gas emissions and electricity consumption on account of generative AI models and tools. However, with CACHE SAVER, our work provides an explicit solution to mitigate the negative impact of LLM-based research on the environment by reducing the inference time, cost, and carbon footprint of LLM-based reasoning frameworks. Furthermore, all the resources required to reproduce the experiments in this paper are publicly available in a well-documented and organized GitHub repository.

## References


Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

Ainslie, J., Lee-Thorp, J., De Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Anthony, L. F. W., Kanding, B., and Selvan, R. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems, July 2020. arXiv:2007.03051.

Bang, F. Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pp. 212–218, 2023.

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., et al. Graph of thoughts: Solving elaborate problems with large language models. In *AAAI*, volume 38, pp. 17682–17690, 2024.

Chen et al. Evaluating large language models trained on code, 2021. arXiv eprint 2107.03374, cs.LG, `https://arxiv.org/abs/2107.03374`.

Chollet, F. et al. The arc agi benchmark, 2019. `https://arcprize.org/arc-agi`.

Chu, K., Liu, T., Li, Y., Yuan, P., and Zhang, W. Car: An efficient kv cache reuse system for large language model inference.

Crawford, K. *The Atlas of AI: Power, Politics, and the Planetary Costs of Artificial Intelligence*. Yale University Press, 2021. ISBN 9780300209570. URL `http://www.jstor.org/stable/j.ctv1ghv45t`.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Del Corro, L., Del Giorno, A., Agarwal, S., Yu, B., Awadallah, A., and Mukherjee, S. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference. *arXiv preprint arXiv:2307.02628*, 2023.

Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., and Anandkumar, A. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *NeurIPS: Datasets and Benchmarks Track*, 2022.

Fu, Z., Chen, F., Zhou, S., Li, H., and Jiang, L. Llmco2: Advancing accurate carbon footprint prediction for llm inferences. *arXiv preprint arXiv:2410.02950*, 2024.

ggml.ai. llama.cpp, 2023. URL `https://github.com/ggml-org/llama.cpp`.

Gill, W., Elidrisi, M., Kalapatapu, P., Ahmed, A., Anwar, A., and Gulzar, M. A. Meancache: User-centric semantic cache for large language model based web services. *arXiv preprint arXiv:2403.02694*, 2024.

Hao, S., Gu, Y., Ma, H., Hong, J. J., Wang, Z., Wang, D. Z., and Hu, Z. Reasoning with language model is planning with world model. In *EMNLP*, 2023.

Helicone. Helicone OS Caching. `https://docs.helicone.ai/features/advanced-usage/caching`.

Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M. W., Shao, Y. S., Keutzer, K., and Gholami, A. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

IEA. Ai is set to drive surging electricity demand from data centres, 2025. `https://tinyurl.com/iea-energy-2030`.

Iyengar, A., Kundu, A., Kompella, R., and Mamidi, S. N. A generative caching system for large language models. *arXiv preprint arXiv:2503.17603*, 2025.

Klein, L., Potamitis, N., Aydin, R., Gulcehre, C., West, R., and Arora, A. Fleet of agents: Coordinated problem solving with large language models using genetic particle filtering, 2024. URL `https://arxiv.org/abs/2405.06691`. arXiv ePrint 2405.06691, cs.CL, `https://arxiv.org/abs/2405.06691`.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *SOSP*, pp. 611–626, 2023a. URL `https://doi.org/10.1145/3600006.3613165`.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023b.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023c. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

LangChain. LangChain Caching. https://python.langchain.com/docs/integrations/llm_caching.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks. *CoRR*, 2021. URL https://arxiv.org/abs/2005.11401.

Li, B., Jiang, Y., Gadepally, V., and Tiwari, D. Llm inference serving: Survey of recent advances and opportunities. *arXiv preprint arXiv:2407.12391*, 2024.

Lightllm Team. Lightllm: A light and fast inference service for llm, 2023. URL https://github.com/ModelTC/lightllm.

Martin, N., Faisal, A. B., Eltigani, H., Haroon, R., Lamelas, S., and Dogar, F. Llmproxy: Reducing cost to access large language models. *arXiv preprint arXiv:2410.11857*, 2024.

MLC-AI. Mlc llm: Universal llm deployment engine with ml compilation, 2023. URL https://llm.mlc.ai/.

Mo, S. and Xin, M. Tree of uncertain thoughts reasoning for large language models. In *ICASSP*, pp. 12742–12746, 2024.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

Park, S., Jeon, S., Lee, C., Jeon, S., Kim, B.-S., and Lee, J. A survey on inference engines for large language models: Perspectives on optimization and efficiency, 2025. https://arxiv.org/abs/2505.01658.

Regmi, S. and Pun, C. P. Gpt semantic cache: Reducing llm costs and latency via semantic embedding caching. *arXiv preprint arXiv:2411.05276*, 2024.

Sel, B., Tawaha, A., Khattar, V., Jia, R., and Jin, M. Algorithm of thoughts: Enhancing exploration of ideas in large language models. In *ICML*, 2024.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, pp. 8634–8652, 2023.

Stojkovic, J., Zhang, C., Goiri, Í., Choukse, E., Qiu, H., Fonseca, R., Torrellas, J., and Bianchini, R. Tapas: Thermal- and power-aware scheduling for llm inference in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1266–1281, 2025.

Suzgun, M. and Kalai, A. T. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Triton. NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023.

Wang, X., Hu, Z., Lu, P., Zhu, Y., Zhang, J., Subramaniam, S., Loomba, A. R., Zhang, S., Sun, Y., and Wang, W. Scibench: evaluating college-level scientific problem-solving abilities of large language models. In *ICML*, 2024a.

Wang, Y., Jiang, Z., Chen, Z., Yang, F., Zhou, Y., Cho, E., Fan, X., Lu, Y., Huang, X., and Yang, Y. Recmind: Large language model powered agent for recommendation. In *NAACL-HLT (Findings)*, pp. 4351–4364, 2024b.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, pp. 24824–24837, 2022.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

Yang, L., Yu, Z., Zhang, T., Cao, S., Xu, M., Zhang, W., Gonzalez, J. E., and Cui, B. Buffer of thoughts: Thought-augmented reasoning with large language models. In *NeurIPS*, 2024.

Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S., and Jiang, J. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. *EuroSys'25*, 2025. URL https://arxiv.org/abs/2405.16444.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*, 2023.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*, 36, 2024.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zhang, D., Zhoubian, S., Hu, Z., Yue, Y., Dong, Y., and Tang, J. ReST-MCTS*: LLM self-training via process reward guided tree search. In *NeurIPS*, 2024.

Zheng, L., Yin, L., Xie, Z., Sun, C. L., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024a.

Zheng, Z., Ji, X., Fang, T., Zhou, F., Liu, C., and Peng, G. Batchllm: Optimizing large batched llm inference with global prefix sharing and throughput-oriented token batching. *arXiv preprint arXiv:2412.03594*, 2024b.

Zhilin, Y., Peng, Q., Saizheng, Z., Yoshua, B., William, C., Ruslan, S., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018. doi: 10.18653/v1/d18-1259. URL https://cir.nii.ac.jp/crid/1363388846142371200.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning, acting, and planning in language models. In *ICML*, 2024.

Zhu, H., Zhu, B., and Jiao, J. Efficient prompt caching via embedding similarity. *arXiv preprint arXiv:2402.01173*, 2024.

## A. Related Works

**Backend optimizations.** LLM serving engines optimize inference on the server side, with many optimizations focusing on efficient memory management and KV cache reuse (Kwon et al., 2023b; Del Corro et al., 2023; Hooper et al., 2024; Chu et al.; Ainslie et al., 2023; Zheng et al., 2024a; Yao et al., 2025; Park et al., 2025). PagedAttention treats the attention key-value cache like a virtual memory system, reducing fragmentation and allowing KV cache sharing within and across requests (Kwon et al., 2023b). KVQuant reduces memory usage by quantizing the KV cache (Hooper et al., 2024). Grouped Query Attention improves memory and computation efficiency by grouping queries to reduce redundant attention computations (Ainslie et al., 2023). CaR introduces a multi-tier caching system to facilitate the reuse and sharing of attention KV caches across different requests (Chu et al.). Additional optimizations target prompt batching and scheduling efficiency (Agrawal et al., 2023; Li et al., 2024; Stojkovic et al., 2025; Zheng et al., 2024b; Yu et al., 2022), as well as improvements in attention and decoding operations (Dao et al., 2022; Ye et al., 2025; Leviathan et al., 2023). For example, FlashAttention-2 enhances attention computation through improved parallelism and work partitioning (Dao et al., 2022). Most production-grade LLM serving systems, such as vLLM (Kwon et al., 2023c), Hugging Face Transformers (Wolf et al., 2020), and NVIDIA Triton Inference Server (Triton), incorporate several such optimizations. Other engines, including DeepSpeed-FastGen (Holmes et al., 2024), llama.cpp (ggml.ai, 2023), MLC LLM (MLC-AI, 2023), SGLang (Zheng et al., 2024a), and LightLLM (Lightllm Team, 2023), further specialize for low-latency decoding, efficient edge deployment, and server-side batching and memory management. Overall, these optimizations remain largely invisible to the practitioners and aim to reduce costs for the model providers. However, they may increase opacity, potentially diminish the quality of results, and the cost savings are rarely passed on to the end user.

**Client-side caching.** Client-side caching addresses the challenges of reducing redundant requests, minimizing latency, and lowering operational costs. While Retrieval-Augmented Generation (RAG) methods reduce the need for LLMs to generate answers from scratch by retrieving relevant information from external sources (Lewis et al., 2021), their primary focus is not caching. Instead, we examine various client-side caching systems that aim to optimize LLM performance. Helicone implements a key-based caching mechanism by storing LLM responses at the edge using Cloudflare Workers (Helicone). GPTCache is an open-source semantic cache that transforms queries and LLM responses into embeddings, conducting similarity searches to retrieve cached responses (Bang, 2023). Developer frameworks like LangChain offer a framework-specific caching layer that supports both exact and semantic matching (LangChain). GPT Semantic Cache utilizes semantic embedding caching by storing query embeddings in in-memory storage systems, such as Redis (Regmi & Pun, 2024). MeanCache introduces a user-centric semantic caching system that preserves privacy through a learned federated similarity model (Gill et al., 2024). GenerativeCache goes beyond traditional caching by synthesizing new responses from multiple cached entries (Iyengar et al., 2025). LLMProxy is a proxy service that implements a single endpoint for applications, supporting model selection, context management, and caching (Martin et al., 2024). In summary, client-side caching solutions for large language models (LLMs) are commonly used, primarily addressing generic workloads through semantic caching (Zhu et al., 2024).

**Key differences.** Existing client-side solutions often lack guarantees for statistical integrity and reproducibility. To the best of our knowledge, by employing a *namespace-aware list-valued cache*, Cache Saver is the only framework that enables response reuse without sacrificing the statistical integrity of the generative LLM.

## B. Cache Saver: Additional Details

The pseudocode for different modules of the Cache Saver framework are presented in Algorithms 1–4. Additionally, each building block of Cache Saver is described below.

### B.1. The Cacher

As motivated in Fig. 1 (§ 1), the prompts issued by internal reasoning steps of multi-step reasoning strategies are highly repetitive. Moreover, in many practical application scenarios, such as stochastic sampling, uncertainty estimation, or ensuring policy diversity, reasoning strategies require multiple independent responses to the same prompt. A naïve KV cache that maps each unique prompt to a single LLM response is undesirable in such scenarios.

Our cacher circumvents the aforementioned concerns by employing a *list-valued cache* (Appx. C.4 shows a snapshot of the cache on real-world tasks) that maintains a sequence of responses for each unique prompt. Additionally, to enable response

Table 1: Sample allocation: (Left) Independent vs. (Right) CACHE SAVER-coupled experiments.

| Experiment | Observed Samples (Independent) | Observed Samples (Coupled) |
|---|---|---|
| $E_1$ ($\text{NS}_1$) | $Z_1, Z_2, Z_3, \dots, Z_{n_1}$ | $Z_1, Z_2, Z_3, \dots, Z_{n_1}$ |
| $E_2$ ($\text{NS}_2$) | $Z_{n_1+1}, Z_{n_1+2}, \dots, Z_{n_1+n_2}$ | $Z_1, Z_2, Z_3, \dots, Z_{n_2}$ |
| $E_3$ ($\text{NS}_3$) | $Z_{n_1+n_2+1}, Z_{n_1+n_2+2}, \dots, Z_{n_1+n_2+n_3}$ | $Z_1, Z_2, Z_3, \dots, Z_{n_3}$ |
| $\vdots$ | $\vdots$ | $\vdots$ |

reuse without sacrificing the statistical integrity of the generative model (in this case, an LLM), we introduce the concept of *namespaced caching*. This implies that all responses to a given prompt are independent within a namespace, whereas responses may be reused across namespaces. In other words, responses can never be reused within a namespace. Revisiting the example presented in Fig. 2, let's say that the search strategy in $\text{NS}_1$ now requires two additional responses to $Q$. The cache contains four responses ($[R_1, R_2, R_3, R_4]$) to $Q$, of which the first three have already been used once in $\text{NS}_1$; thus, they cannot be reused. Following namespaced caching, one new response $R_5$ will be obtained by the LLM, and $[R_4, R_5]$ will be used to resolve the request. Formally, namespaced caching is achieved via a *stochastic coupling of LLM responses*.

**Sample reuse through stochastic coupling.** Consider an LLM as a probabilistic oracle. Given a prompt $p$ and a parameter set $\theta$ (e.g., sampling temperature, top-$p$ threshold, etc.), LLM responses follow a probability distribution $\rho_{p,\theta}$. We assume that the pair $(p, \theta)$ fully parameterizes this distribution.

Initially, consider a scenario with a series of experiments $E_k$ (corresponding to namespaces $\text{NS}_k$), each independently requesting samples drawn *i.i.d.* from $\rho_{p,\theta}$. Formally, let $(Z_i)_{i=1}^{\infty}$ denote an infinite sequence of random variables with $Z_i \overset{\text{iid}}{\sim} \rho_{p,\theta}$. Each experiment independently accesses distinct samples leading to a non-overlapping partitioning of the infinitie sequence.

However, in practice, generating independent samples for each experiment is inefficient. CACHE SAVER explicitly couples these experiments by using a shared prefix of the same infinite sequence $(Z_i)_{i=1}^{\infty}$. Intuitively, this corresponds to using a *shared random seed* across all experiments and caching seeded random samples. Within each experiment $E_k$, samples are *i.i.d. by construction*, directly inherited from the infinite sequence $(Z_i)$. For two experiments $E_i$ and $E_j$, let $m = \min(n_i, n_j)$. Their first $m$ samples coincide almost surely: $f_i(Z)_k = f_j(Z)_k = Z_k \ \ \forall k \leq m$.

Table 1 shows a side-by-side comparison of the two sampling strategies. The cacher (Alg. 1) ensures a shared seed for all coupled random variables. The first evaluation of a random variable sets its value across all experiments. Race conditions between experiments are resolved by the use of a dynamically generated asynchronous mutex table.

### B.2. Beyond Caching

**Batcher.** CACHE SAVER's position between the reasoning strategy and the LLM allows for a range of additional, transparent optimizations. As shown in Fig. 2, we extend the caching layer into a pipeline of modular and composable building blocks. The first of these is a batching layer (Alg. 2), which uses an *asynchronous producer-consumer queue* to collect incoming requests and group them into batches. The batching mechanism is governed by two tunable parameters: a timeout and a batch size, which together control the trade-off between responsiveness and throughput. Under light load, small batches pass through quickly with minimal delay; under heavy load, larger batches form naturally, improving efficiency. This batching is immediately beneficial for *local models*, where it allows better hardware utilization. In combination with other CACHE SAVER building blocks, it also improves efficiency and reproducibility in online API settings.

**Reordering Requests for Reproducible Results.** The order in which requests are resolved in asynchronous computing is non-deterministic and can therefore change results even across identical runs. To ensure reproducibility, CACHE SAVER includes a reordering module (Alg. 3) that ensures a deterministic order within each batch. Requests are sorted by a stable identifier before being passed to the LLM and reordered back to their original positions after the LLM responds. The asynchronous reorderer can guarantee reproducible request resolution in scenarios where the asynchronous batcher is able to group all in-flight requests into a single batch.

**Deduplicator.** Many LLM inference engines support *efficient same-input, multiple-response* use cases, which are enabled by optimizations such as paged attention, prefix prompt caching, input sharing, etc.. Such optimizations are afforded by most online platforms (e.g., OpenAI, Anthropic, etc.), allowing users to request multiple *i.i.d.* samples for a given prompt, charging for input tokens only once, and returning a list of completions; and can be enabled for local deployments via inference engines such as vLLM (Kwon et al., 2023a). Overall, these optimizations incentivize grouping identical requests to reduce redundant input processing. CACHE SAVER's deduplication module (Alg. 4) takes advantage of this by identifying requests within a batch that share the same prompt, parameters, and namespace. These requests are merged into a single LLM call with an aggregated sample count, reducing both cost and latency.

**Algorithm 1** CACHER

**Global State:**
Cache $\mathcal{C} : (p, \theta) \mapsto [Z_1, Z_2, \dots]$, initially empty
Usage counters $\mathcal{U} : (E_k, p, \theta) \mapsto u \in \mathbb{N}_0$, initially 0
Mutex table $\mathcal{M} : (p, \theta) \mapsto$ *async* lock

**Asynchronous Procedure** RequestSamples($r$)
  **Input:** Request $r = (E_k, p, \theta, n, \mathsf{id})$
  **Output:** Future $f$ resolving to $(Z_{u+1}, \dots, Z_{u+n})$
$f \leftarrow$ new Future
key $\leftarrow (p, \theta)$
**if** key $\notin \mathcal{M}$ **then**
  $\mathcal{M}$[key] $\leftarrow$ new async mutex
**end if**
**async acquire** $\mathcal{M}$[key]
  **if** key $\notin \mathcal{C}$ **then**
    $\mathcal{C}$[key] $\leftarrow$ [ ]
  **end if**
  **if** $(E_k, p, \theta) \notin \mathcal{U}$ **then**
    $\mathcal{U}[(E_k, p, \theta)] \leftarrow 0$
  **end if**
  $u \leftarrow \mathcal{U}[(E_k, p, \theta)]$
  $Z \leftarrow \mathcal{C}$[key]
  $n_{\text{fresh}} \leftarrow \max(0,\ u + n - |Z|)$
  **if** $n_{\text{fresh}} > 0$ **then**
    Draw fresh samples $(Z_{|Z|+1}, \dots, Z_{|Z|+n_{\text{fresh}}}) \overset{\text{iid}}{\sim} \rho_{p,\theta}$
    Extend: $\mathcal{C}$[key] $\leftarrow Z + (Z_{|Z|+1}, \dots, Z_{|Z|+n_{\text{fresh}}})$
    $Z \leftarrow \mathcal{C}$[key]
  **end if**
  Extract samples: $(Z_{u+1}, \dots, Z_{u+n})$
  Resolve $f$ with value $(Z_{u+1}, \dots, Z_{u+n})$
  Update usage: $\mathcal{U}[(E_k, p, \theta)] \leftarrow u + n$
**release** $\mathcal{M}$[key]
**return** $f$

### B.3. Cache management: storage and eviction

Since retrieving cached responses from the disk is many orders of magnitude faster than issuing a new LLM query, especially for remote APIs, CACHE SAVER persists all cache entries to the disk by default. Unlike typical in-memory caches, where storage is a limiting factor, here storage is cheap and recomputation is costly, so no advanced eviction policy is necessary for most use cases. That said, should eviction be required (e.g., due to storage constraints), the simplest approach is to evict all responses associated with a given prompt and parameter set. When the evicted prompt is encountered again, the cache miss results in requests to the underlying LLM, and the newly generated responses are stored in the cache. This preserves the cache's guarantee of *i.i.d.* samples within each namespace.

**Algorithm 2** ASYNC BATCHER

**Global State:** Queue $\mathcal{Q} \leftarrow [\,]$
**Parameters:** batch size $N$, timeout $\delta$, overflow flag overflow

**Asynchronous Procedure** RequestSamples($[r_1, \dots, r_m]$)
  **Input:** List of $m$ requests $r_i = (E_k^i, p^i, \theta^i, n^i, \mathsf{id}^i)$
  **Output:** List of futures $[f_1, \dots, f_m]$, where each $f_i$ resolves to $n^i$ samples
**for** $i = 1$ to $m$ **do**
  $t \leftarrow$ current time
  $f_i \leftarrow$ new Future
  enqueue $(r_i, t, f_i)$ into $\mathcal{Q}$
**end for**
**return** $[f_1, \dots, f_m]$

**Asynchronous Background Task** BatchWorker()
**while true do**
  $(r_1, t_1, f_1) \leftarrow$ *async* $\mathcal{Q}$.get()
  Initialize: $\mathcal{B} \leftarrow [r_1]$, $\mathcal{F} \leftarrow [f_1]$
  $t_{\mathsf{start}} \leftarrow t_1$
  **while true do**
    $\tau \leftarrow \delta -$ (current time $- t_{\mathsf{start}}$)
    **if** $|\mathcal{B}| \geq N$ **or** $\tau \leq 0$ **then break**
    **try**
      $(r_i, t_i, f_i) \leftarrow$ *async* $\mathcal{Q}$.get() with timeout $\tau$
      Append $r_i$ to $\mathcal{B}$, $f_i$ to $\mathcal{F}$
    **catch** timeout: break
  **end while**
  **if** overflow **then**
    **while** $\mathcal{Q}$ is not empty **do**
      $(r_i, t_i, f_i) \leftarrow \mathcal{Q}$.get_nowait()
      Append $r_i$ to $\mathcal{B}$, $f_i$ to $\mathcal{F}$
    **end while**
  **end if**
  responses $\leftarrow$ model.batch_request($\mathcal{B}$)
  **for** $i = 1$ to $|\mathcal{B}|$ **do**
    resolve $f_i \leftarrow$ responses[$i$]
  **end for**
**end while**

### B.4. Cache consistency

To ensure cache consistency in concurrent asynchronous environments, CACHE SAVER utilizes per-key mutex locks. When a request is received, a unique hash-based key is generated, and a corresponding mutex is initialized if no other request holds the mutex. Alternatively, if a different request currently holds the mutex, it is retrieved from that request once it is freed. This mutex guards access to the critical section where the cache, potentially updated with fresh model responses, is read, and usage counters are incremented. By serializing access to cache entries on a per-key basis, the implementation prevents race conditions and ensures accurate tracking of response usage. At the same time, requests for different keys proceed concurrently, maintaining overall efficiency.

### B.5. Batcher: Handling batch overflows

To improve efficiency in scenarios where multiple clients issue identical requests concurrently, CACHE SAVER includes an option that allows the batch to overflow. When enabled, this mechanism allows additional requests to be added to a batch

**Algorithm 3** ASYNC REORDERER

**Asynchronous Procedure** RequestSamples($[r_1, \dots, r_m]$)
  **Input:** List of $m$ requests $r_i = (E_k^i, p^i, \theta^i, n^i, \mathsf{id}^i)$
  **Output:** List of futures $[f_1, \dots, f_m]$, each resolving to $n^i$ samples
Create futures: $f_i \leftarrow$ new Future for $i = 1$ to $m$
Let $\mathcal{R} \leftarrow [r_1, \dots, r_m]$ and $\mathcal{F} \leftarrow [f_1, \dots, f_m]$
Compute sorted indices: $\mathcal{I}_{\text{sorted}} \leftarrow$ sort indices of $\mathcal{R}$ by $\mathsf{id}^i$
Reorder requests and futures:
  $\mathcal{R}_{\text{sorted}} \leftarrow \mathcal{R}[\mathcal{I}_{\text{sorted}}]$
  $\mathcal{F}_{\text{sorted}} \leftarrow \mathcal{F}[\mathcal{I}_{\text{sorted}}]$
**asynchronously**:
  responses $\leftarrow$ await model.request($\mathcal{R}_{\text{sorted}}$)
  **for** $i = 1$ to $m$: resolve $\mathcal{F}_{\text{sorted}}[i] \leftarrow$ responses[$i$]
**return** $[f_1, \dots, f_m]$

**Algorithm 4** ASYNC DEDUPLICATOR

**Asynchronous Procedure** RequestSamples($[r_1, \dots, r_m]$)
  **Input:** List of $m$ requests $r_i = (E_k^i, p^i, \theta^i, n^i, \mathsf{id}^i)$
  **Output:** List of futures $[f_1, \dots, f_m]$, each resolving to $n^i$ samples
Initialize:
  $\mathcal{K} \leftarrow \{\}$ {Deduplication key $\mapsto (E_k, p, \theta)$}
  $\mathcal{N} \leftarrow \{\}$ {Key $\mapsto$ total requested samples}
  $\mathcal{M} \leftarrow \{\}$ {Key $\mapsto$ list of (future, count) pairs}
**for** $i = 1$ to $m$ **do**
  $r_i = (E_k^i, p^i, \theta^i, n^i, \mathsf{id}^i)$
  $k_i \leftarrow \text{Hash}(E_k^i, p^i, \theta^i)$
  $f_i \leftarrow$ new Future
  Append $(f_i, n^i)$ to $\mathcal{M}[k_i]$
  $\mathcal{K}[k_i] \leftarrow (E_k^i, p^i, \theta^i)$
  $\mathcal{N}[k_i] \leftarrow \mathcal{N}[k_i] + n^i$
**end for**
**for** each key $k$ **do**
  $(E_k, p, \theta) \leftarrow \mathcal{K}[k]$
  $n_{\text{total}} \leftarrow \mathcal{N}[k]$
  **asynchronously**:
    responses $\leftarrow$ await model.request($E_k, p, \theta, n_{\text{total}}$)
    resolve all futures in $\mathcal{M}[k]$ using responses, in order
**end for**
**return** $[f_1, \dots, f_m]$

even after the specified batch size has been reached, but only if they are *duplicates*, i.e., they share the same input (prompt and decoding parameters) as an existing request in the batch. Overall, this enables the *deduplicator* module of CACHE SAVER to work at full efficiency, improving the overall performance of the CACHE SAVER pipeline.

### B.6. Practical considerations

CACHE SAVER is built from modular building blocks, allowing users to flexibly compose different pipelines that combine caching, batching, deduplication, and reordering in any order or subset, depending on their application needs. In particular, the choice of namespace assignment gives precise control over the trade-off between strict sample independence (one namespace per benchmark) for unbiased benchmarking and maximal cost savings (one namespace per benchmark puzzle or $N$ namespaces per benchmark, where $N$ is the number of puzzles in the benchmark) for large-scale ablations or production

deployments. For example, users can construct pipelines with full independence across benchmarks or enable aggressive reuse within a benchmark as appropriate.

## C. Additional Experimental Details

### C.1. Detailed Task Descriptions

#### C.1.1. Game of 24

The Game of 24 is a math puzzle where players are given four numbers and must use each of them exactly once, along with the basic arithmetic operations (+, –, ×, ÷), to form an expression that evaluates to 24.

Our benchmark includes 1,362 such puzzles collected from 4nums.com, organized in ascending order of difficulty. Each puzzle provides four input numbers, and the goal is to generate a valid equation that results in 24. Following the approach of ToT (Yao et al., 2024), we designate puzzles numbered 901 to 1000 as our test set.

#### C.1.2. SciBench

SciBench (Wang et al., 2024a) is a scientific reasoning benchmark designed to evaluate college-level problem-solving abilities across subjects such as mathematics, physics, and chemistry. Each task presents an open-ended problem that requires multi-step reasoning, domain-specific knowledge, and advanced computations, including calculus and differential equations. Problems are drawn from widely used textbooks and university exams.

Following the approach of ReST-MCTS (Zhang et al., 2024), we sampled 109 problems spanning different subjects to form the test set. Quality is measured using an *accuracy* metric, defined as the proportion of problems correctly solved according to the official solutions (exact matching).

#### C.1.3. HumanEval

HumanEval (Chen et al., 2021) is a code generation benchmark where participants are given natural language docstrings and must generate Python functions that correctly implement the described behavior. Each problem includes a hidden test suite used to verify functional correctness.

Following the setup from Reflexion (Shinn et al., 2023), the benchmark consists of 100 programming tasks in the test set. We evaluate performance using the *pass@1* metric, which measures the proportion of problems solved correctly on the first attempt.

#### C.1.4. HotpotQA

HotpotQA (Zhilin et al., 2018) is a large-scale question answering benchmark that tests an agent's ability to perform multi-hop reasoning across multiple documents. Multi-step approaches, such as ToT, are permitted to interact with an API that enables document retrieval and targeted information lookup.

Following prior work (Zhou et al., 2024; Shinn et al., 2023), we evaluate on a set of 100 randomly selected questions. The quality of a response is judged based on *exact match* (EM) with the oracle answer.

#### C.1.5. Shakespearean Sonnet Writting

Shakespearean Sonnet Writing (Suzgun & Kalai, 2024) is a creative generation task where the goal is to compose a 14-line sonnet adhering to the classic rhyme scheme "ABAB CDCD EFEF GG". Each sonnet must include three provided words verbatim.

Following Suzgun et al. (Suzgun & Kalai, 2024), we randomly sampled 50 datapoints to form the test set. Quality is measured using an *accuracy* metric, which reflects the proportion of sonnets that both satisfy the rhyme scheme and include all three required words exactly as given.

### C.2. Detailed Descriptions of Reasoning Strategies

Multi-step reasoning strategies can be broadly grouped into two high-level categories: (1) structured reasoning and (2) iterative reasoning, which capture the major design paradigms in the space of LLM reasoning. To systematically assess the effectiveness of CACHE SAVER in contemporary multistep reasoning settings, we pick three and two representative state-of-the-art (SOTA) methods from the two categories, respectively, that incorporate distinct algorithmic choices and reasoning dynamics. Specifically, we pick ToT and FoA from structured reasoning, and React, RAP, and ReST-MCTS from iterative reasoning. The five representatives are described below.

• **Tree of Thoughts (ToT):** Decomposes the problem into multiple chains of thoughts, organized in a tree structure. Thought evaluation and search traversal algorithms are utilized to solve the problem (Yao et al., 2024).

• **Fleet of Agents (FoA):** Decomposes the problem into multiple chains of thoughts. Employs a genetic-type particle filtering approach to navigate through dynamic tree searches to solve the problem (Klein et al., 2024).

• **React:** is a reasoning method that interleaves reasoning (thought generation) and acting (taking environment-interacting actions) to solve problems interactively. Each action's output informs subsequent reasoning, enabling adaptive and dynamic problem-solving (Yao et al., 2023).

• **Reasoning via Planning (RAP):** is a reasoning framework that equips Large Language Models (LLMs) with an internal world model and employs Monte Carlo Tree Search (MCTS) for strategic exploration of reasoning paths. RAP repurposes the LLM to simulate future states and evaluate potential actions, enabling deliberate planning and improved problem-solving performance (Hao et al., 2023)

• **ReST-MCTS:** is a reasoning method that employs a modified Monte Carlo Tree Search (MCTS) algorithm, guided by a process reward model (PRM), to explore and evaluate reasoning paths. In this work, we only evaluate in-context reasoning strategies; thus, we utilize only the MCTS* component of ReST-MCTS, excluding the self-training aspects involving policy and reward model updates (Zhang et al., 2024).

### C.3. Detailed Descriptions of Evaluation Metrics

- **Quality** indicates how well a reasoning strategy performs on a benchmark task. It depends on the task type and can be measured by accuracy, score, or success rate. The exact definition of the quality metric is provided with the description of each benchmark task in Appx. C.1.

- **Latency** measures how long it takes for an LLM to respond to a request. For each reasoning strategy and benchmark task, we report the average latency, i.e., the average time required per call.

- **Throughput.** Let each datapoint in a benchmark task be referred to as a puzzle. For each reasoning strategy, we report the number of puzzles solved per second. Overall, throughput reflects a system's ability to process multiple tasks concurrently.

- **#Tokens** denote the total number of tokens used by a reasoning strategy to solve a benchmark task. We report the total, including both the input tokens sent to the model and the output tokens in the response generated by an LLM.

- **Time** refers to the wall clock time, i.e., the total time taken by a reasoning strategy to solve a benchmark task from start to finish, including all processing and waiting times.

- **Cost (API-based LLMs only)** denotes the total monetary cost (in USD) of executing a reasoning strategy on a benchmark task when using an LLM via an API call to an online platform. We compute the cost based on the number of tokens processed and the platform provider's pricing.

- **Energy consumption and Carbon footprint (Local LLMs only)** measures the total energy consumption (in kWh) and the estimated $CO_2$ emissions (in grams) to run a reasoning strategy for a benchmark task. We measure these quantities using Carbontracker (Anthony et al., 2020).

### C.4. Implementation Details

C.4.1. Platforms, Model checkpoints, and Prices

The GPT models were accessed through the OpenAI API while Claude models were accessed through the Anthropic API. For our local experiments, our models were deployed using the vLLM (Kwon et al., 2023c) inference engine.

To compute the costs of the online experiments, we used the current model prices indicated by the corresponding platform. The specific models snapshot used in this work, along with their respective prices, are presented in 2. Note that Llama4-Scout was run locally, and thus, the cost is listed as N/A.

Table 2: Base LLM snapshot prices. OpenAI and Anthropic prices for each model used during the implementation of the project.

| | US$ per 1M prompt tokens | US$ Per 1M completion tokens |
|---|---|---|
| **GPT4.1-Nano-2025-04-14** | 0.10 | 0.40 |
| **Claude3.5-Haiku** | 0.8 | 4 |
| **Llama4-Scout-17B-16E-Instruct** | N/A | N/A |

C.4.2. Model configurations

Generation parameters specified when making calls to any of the models used throughout this project. These parameters were not defined by us, but by the implementation where the respective prompts were introduced. However, as newer models were used for this study, we only adjusted the maximum allowed completion tokens as needed to ensure compatibility and successful completion of responses.

C.4.3. Cache Saver Hyperparameters

The Cache Saver framework itself has two hyperparameters: (1) batch size and (2) timeout of the asynchronous batcher. The batch size should be chosen large enough to allow the deduplicator to identify and group duplicate prompts efficiently. At the same time, too large a batch size can trigger timeouts. We use Cache Saver in a setup where many tasks, frameworks, configurations, etc. are evaluated asynchronously and choose *a batch size of 300 and a timeout of 2 seconds*.

Importantly, when running a set of experiments at the same time, tuning the batch size carefully can lead to further cost savings. The batch size for the asynchronous batcher may be larger than batch sizes that can be processed within the memory constraints of a local model or the rate limits of a third-party API to a model hosted at an online platform. Slicing a batch into optimal chunks is a downstream task and should be implemented in a model-specific wrapper.

C.4.4. Prompts

Due to the large number of methods and tasks presented in this paper, including all corresponding prompts would be impractical within the main text. Therefore, we provide a comprehensive collection of all prompts used in our experiments on our GitHub repository: `https://github.com/au-clan/cachesaver/blob/main/prompts.md`.

Table 3: Generation parameters specified when making requests to a base LLM.

| | **max_tokens** | **temperature** | **top_p** | **stop** |
|---|---|---|---|---|
| **Game of 24** | 200 | 0.7 | 1 | Null |
| **SciBench** | 300 | 0.7 | 1 | Null |
| **HumanEval** | 200 | 0.7 | 1 | Null |
| **HotpotQA** | 300 | 0.7 | 1 | Null |
| **Sonnet Writing** | 800 | 1.0 | 1 | Null |

### C.5. Additional Results

C.5.1. Cache Saver for practical applications

In this experiment, we evaluate Cache Saver's ability to support practical machine learning applications, namely, A1: tuning hyperparameters, A2: performing ablation analysis of a reasoning strategy, and A3: benchmarking multiple reasoning strategies to identify the best. In all cases, we use GPT4.1-Nano as the base LLM, three benchmark tasks, namely "Game of 24", "HumanEval", and "SciBench", and report average cost (with and without Cache Saver) and the marginal cost (with Cache Saver), which represents the additional cost of adding a new method (A3) or a new hyperparameter configuration (A1). The results are presented in Fig. 5.

**A1: Hyperparameter tuning.** For A1, we tune the hyperparameters for ToT (Yao et al., 2024) by conducting a grid-search over tree-width: [1, 3, 5], tree-depth: [2, 3, 4], and #evaluations of the value prompt: [1, 2, 3]. We find that Cache Saver reports substantial performance improvements: *6x lower cost, tokens, and latency* and *7x higher throughput*. While average (with CS, yellow bar) presents the realistic setting of executing the full experiment, the marginal (with CS, green bar) is also valuable as it presents the added cost of incorporating a new variation in the experiment.

**A2: Ablation analysis.** For A2, we analyze the three major variations of the FoA (Klein et al., 2024) algorithm, in particular, by removing the (1) selection phase, backtracking mechanism, and resampling strategy. Here, Cache Saver only obtains a *2.5x* performance improvement. We hypothesize that running the same method with a different hyperparameter configuration should result in a more similar reasoning strategy when compared to running a different variation of a method, which offers a plausible explanation for the differences in the observed performance improvements.

**A3: Benchmarking.** For A3, we evaluate all the structured reasoning strategies benchmarked in our study, i.e., ToT (Yao et al., 2024), GoT (Besta et al., 2024), and FoA (Klein et al., 2024). Here, Cache Saver obtains a *2x* (slightly lower than A2) performance improvement. The hypothesis remains similar: the potential for reuse is even lower when considering entirely different reasoning strategies than variations of a specific reasoning strategy. That said, a cross-framework reuse potential is an interesting and novel finding in its own right.

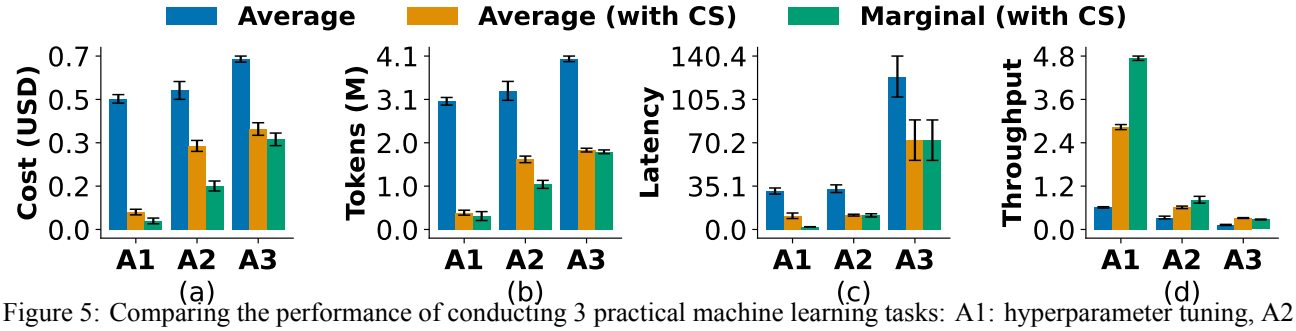


Figure 5: Comparing the performance of conducting 3 practical machine learning tasks: A1: hyperparameter tuning, A2: ablation analysis, and A3: benchmarking, with and without Cache Saver using GPT4.1-Nano as base LLM.

### C.6. Analyses

C.6.1. Impact of Namespace Size

§ B.1 formalized the relationship between the namespace configuration and statistical integrity of an experiment. Let $n$ and $N$ be the number of datapoints in a namespace and a benchmark, respectively. We define namespace fraction (NF) to be $n/N$, which naturally lies between $[0, 1]$ as $n$ is upper-bounded by $N$. Here, we analyze the impact of NF on the cost and carbon footprint (Fig. 6 presents the results). Recall that NF=1 corresponds to the statistically correct configuration (§ B.1), whereas NF=0 corresponds to the maximum cost-saving configuration. We find that the cost and carbon footprint increase with increasing NF; however, interestingly, we noticed that the quality still remains statistically indistinguishable across all NF values. While all the main results reported in the paper correspond to NF=1, in practical settings, NF could even be set to a lower value to increase cost savings even further.

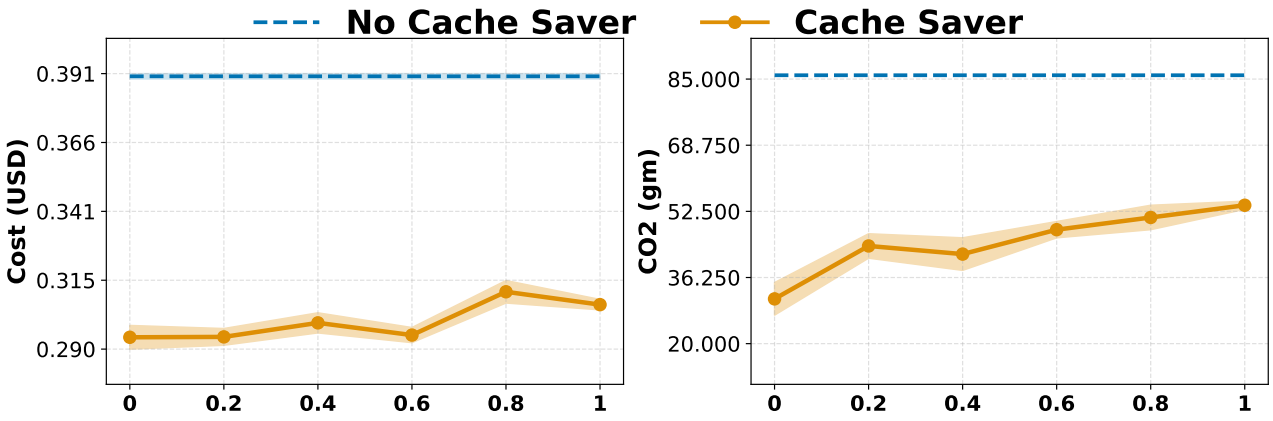


Figure 6: **Namespace impact:** Analyzing the impact on cost as a function of the percentage of puzzles sharing the same namespace for GPT-4.1-Nano (Left, cost in USD) and LLaMA4-Scout (Right, cost in estimated $CO_2$ emissions).

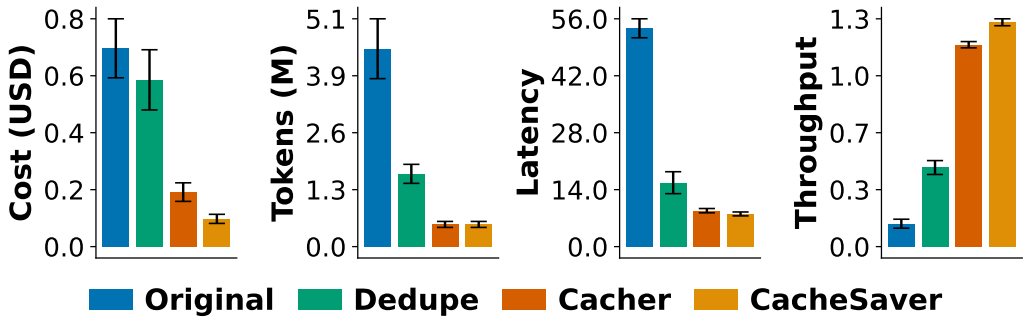


Figure 7: Ablation analysis to study the impact of the *deduplicator* and *cacher* modules of CACHE SAVER on the performance of FoA (Klein et al., 2024) in the Game of 24 task with GPT4.1-Nano as the base LLM.

### C.6.2. CACHE SAVER Ablation Analysis

Fig. 7 presents the impact of the *deduplicator* and *cacher* of CACHE SAVER. We report performance metrics: (1) without CACHE SAVER, (2) with only *deduplicator*, (3) with only *cacher*, and (4) with CACHE SAVER. We find that the *cacher* contributes the most to the improvements seen by CACHE SAVER, followed by the *batcher*. This result shows the strength of our novel caching paradigm, while also highlighting the practical improvements obtained by other CACHE SAVER modules.

### C.6.3. Existing optimizations vs. CACHE SAVER

Finally, we study how much CACHE SAVER saves over and above existing platform-specific optimizations (such as KV caching, paged attention, etc.) as already provided by OpenAI or vLLM. Table 4 presents the results. We find that while existing optimizations lead to savings of $\simeq$3.3% on average, CACHE SAVER alone obtains $\simeq$20%. However, when both optimizations are switched on, the overall improvement reported by CACHE SAVER alongside existing optimizations is $\simeq$14%. This shows that these optimizations likely collide with each other, which is expected; however, CACHE SAVER still reports a substantial overall improvement. That said, studying co-optimization strategies constitutes as future work.

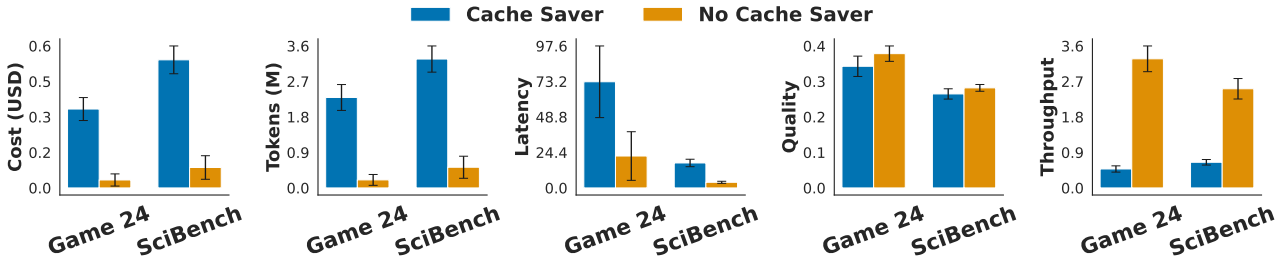


Figure 8: OpenAI: Hyperparameter tuning for Tree of Thoughts across tasks -> Need to add HumanEval

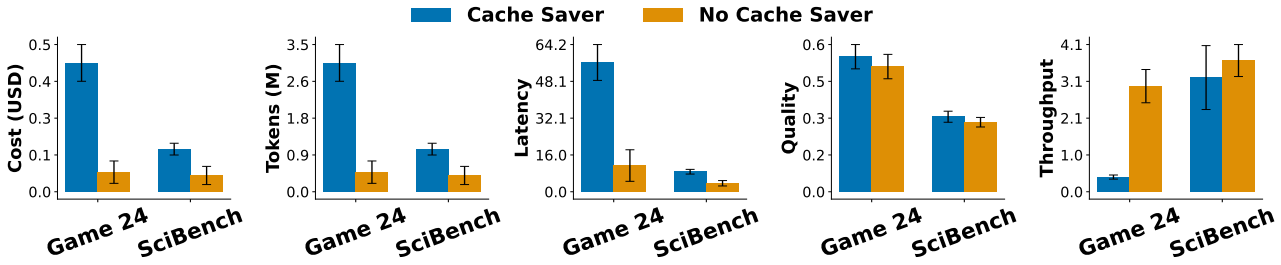


Figure 9: Llama: Hyperparameter tuning for Tree of Thoughts across tasks -> Need to add HumanEval

Table 4: Analyzing the impact of existing optimizations and CACHE SAVER on the cost (in US$) of ToT (Yao et al., 2024) across Game of 24, SciBench, and HumanEval using GPT-4.1-Nano as the base LLM. Values indicate the percentage reduction in cost relative to the original.

| Task | Existing | CacheSaver | Both |
|---|---|---|---|
| Game 24 | 7.7% | 30.8% | 25.5% |
| SciBench | 0.0% | 18.2% | 6.1% |
| HumanEval | 2.2% | 12.1% | 7.7% |
| Average | 3.3% | 20.4% | 13.1% |

| Cache retrievals (%) | ToT | FoA | Average |
|---|---|---|---|
| Game of 24 | 50.39 | 43.04 | 46.71 |
| Mini Crosswords | 1.98 [3] | 52.78 | 27.38 |
| Average | 26.19 | 47.91 | 37.04 |

Table 5: Percentage of prompt responses retrieved from the cache instead of being generated by the LLM during the intra-framework namespace sharing experiment. By allowing each puzzle instance to reuse responses produced by others (with one-time usage constraints), CacheSaver significantly reduces redundant generation, highlighting its effectiveness in promoting efficiency through controlled response sharing.

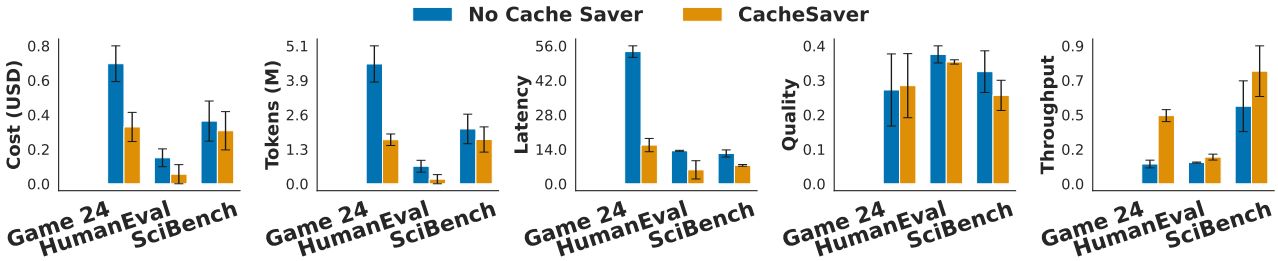


Figure 10: OpenAI: Ablation analysis for Fleet of Agents across tasks

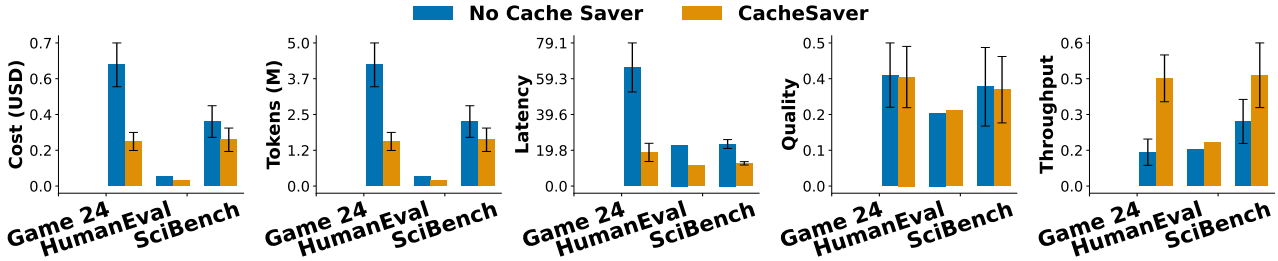


Figure 11: Llama: Ablation analysis for Fleet of Agents across tasks

C.6.4. Hyperparameter tuning

C.6.5. Ablation Analysis

C.6.6. Benchmarking

C.6.7. Impact of Namespace size

C.6.8. Optimization interference

[3]The best performing implementation of Mini Crosswords for ToT is using its Depth-First Search variation. As a result, by definition all the states that the algorithm traverses through are unique, and because of this, most prompts are as well. Since the prompts are unique, minimal reuse is possible and by extension the percentage of cache retrievals is nominal in this experiment.

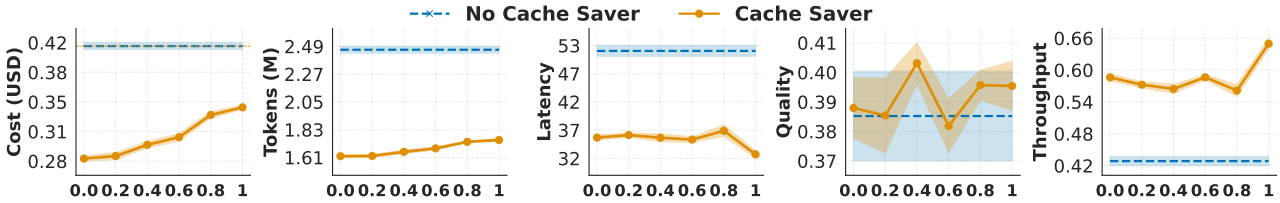


Figure 12: **Namespace impact:** Analyzing the effect of the percentage of puzzles sharing the same namespace on the Game of 24 task, for the Tree of Thoughts method.

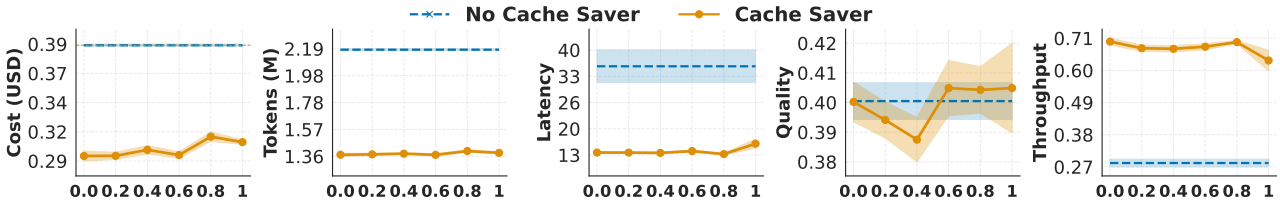


Figure 13: **Namespace impact:** Analyzing the effect of the percentage of puzzles sharing the same namespace on the Game of 24 task, for the Fleet of Agents method.