Finding the Sweet Spot for Small Models in Large Language Models

Anonymous ACL submission

Abstract

Large language models (LLMs) are increasingly used for program-aided reasoning, agentic decision making, and structured task execution, but these settings often incur substantial inference cost. Many such requests share similar computational structures while differing in variables, constraints, or contexts, creating opportunities for program-level caching. Since program caches need to reapply reusable computation logic to new requests, their key steps often involve lightweight and structured operations such as variable extraction, program binding, and generation acceleration, which are well suited for small models. We propose an inference optimization framework centered on reusable program caches. The framework converts Program-of-Thoughts (PoT)-style programs from one-time reasoning artifacts into reusable cache objects, and reuses the same small model for two roles: semantic variable extraction on the cache-hit path and speculative drafting during target-LLM generation. Experiments on shopping-style request datasets, WebShop, Formula, and CodeTAT-QA show that our proposed framework reduces inference latency and improves effective cache reuse while preserving comparable or better task quality than existing caching and generation baselines, achieving up to about 3.1× latency speedup; in parallel serving experiments, it improves throughput by about 2.8× over PoT-style methods. These results suggest that the sweet spot for small models in large-model inference systems lies not in solving complex tasks independently, but in performing lightweight, structured, and verifiable auxiliary operations.

1 Introduction

Large Language Models (LLMs) have demonstrated strong capabilities in complex reasoning, code generation, tool use, and agentic task execution. This progress has been closely associated with scaling model size, training data, and computation (Brown et al., 2020; Kaplan et al., 2020; Hoffmann et al., 2022). Recent systems continue this trend through both large dense models and sparse Mixture-of-Experts architectures, such as Llama 3 and DeepSeek-V3 (Grattafiori et al., 2024; DeepSeek-AI, 2024). However, larger target LLMs also introduce higher inference latency, memory pressure, and serving cost. Many program-aided reasoning tasks, structured task executions, and agentic workflows involve related requests that differ in request-specific variables, constraints, or contexts, yet share similar computational structures, action templates, or variable relations. Therefore, instead of generating a new program for every request, the system can potentially reuse computation logic from previously generated programs.

We focus on where small models can provide value in this reusable-program setting. Unlike response-level caching, reusable program caching does not simply return a previous answer; rather, it must adapt existing program logic to new inputs through intermediate operations between the new request and the cached program. These operations are typically well-bounded, low-cost, and easier to constrain than full task solving. At the same time, small models are cheaper to invoke, faster to respond, and suitable for local structured tasks, making them a natural fit for such auxiliary operations rather than for independently solving the entire complex task.

Existing LLM inference optimization methods reduce inference cost from different directions. Program-aided reasoning methods, including Program-of-Thoughts (PoT) and Program-aided Language Models (PAL), translate natural-language problems into executable programs and improve reliability on computation-heavy tasks (Chen et al., 2023b; Gao et al., 2023). However, they typically require the target LLM to generate a new program for every request. More importantly, these one-time programs often entangle computation logic with request-specific variables, and may

Table 1: Qualitative comparison of representative inference optimization methods. Resource Util. refers to reducing target-LLM computation.

Method	Acc.	Speed	Resource Util.
PoT	✓	×	×
PAL	✓	×	×
Speculative Decoding	–	✓	–
Speculative Sampling	–	✓	–
GPTCache	–	✓	✓
GenCache	–	✓	✓
Ours	✓	✓	✓

even hardcode concrete values or conditions into the generated code. Therefore, they are difficult to directly cache and reuse for new requests from the same task family. Cache-based reuse methods, such as GPTCache and GenCache, can reduce repeated inference, but their quality depends on whether cache matching and variable binding are reliable (Bang, 2023; Chakraborty et al., 2025). Speculative Decoding and Speculative Sampling accelerate a single target-model generation through draft-and-verify mechanisms, but they do not directly exploit computational homogeneity across requests (Leviathan et al., 2023; Chen et al., 2023a).

Representative directions include PoT and PAL for program-aided reasoning, Speculative Decoding and Speculative Sampling for draft-and-verify acceleration, and GPTCache and GenCache for cache-based reuse. Table 1 summarizes these methods from the perspectives of task accuracy, speed, and resource utilization. PoT-style methods mainly improve accuracy on complex computation tasks, but often sacrifice speed and resource utilization. Speculative decoding methods mainly improve generation speed, while their resource-utilization benefit is conditional because they introduce an additional draft model. Cache-based reuse can improve speed and resource utilization, but its accuracy depends on the reliability of cache hits. In contrast, our goal is to organize program reasoning, cache reuse, and small-model acceleration within a unified inference framework.

To address this gap, we propose an LLM inference optimization framework centered on reusable program caches. The key idea is to convert PoT-style programs from one-time reasoning artifacts into parameterized cache objects, thereby decoupling computation logic from request-specific data. Each cache entry contains a variable extraction template and a parameterized program: the template specifies which variables should be extracted from a new request, while the program stores reusable computation logic across requests. For cache-hit requests, the system uses a small model to extract semantic variables and binds them to the cached program for execution. For cache-miss requests and cache generation, the target LLM generates new templates and programs, while the same small model serves as a speculative drafter to reduce generation cost.

This design assigns the small model to lightweight, structured operations that are central to reusable program caching. Semantic variable extraction allows cached programs to adapt to new requests that are semantically related but structurally different. Speculative drafting reduces the cost of target-LLM calls that remain necessary during cache-miss inference and cache construction. In this way, the small model supports reusable program caching not by solving the full task independently, but by performing auxiliary operations that are easier to verify and cheaper to execute.

We evaluate Ours on the shopping-style request datasets introduced by GenCache (Chakraborty et al., 2025), WebShop, Formula from FinLoRA (Wang et al., 2025), and CodeTAT-QA from BizBench (Krumdick et al., 2024), and further analyze its robustness under long contexts and parallel serving. On a financial reasoning task with stable computational structures, Ours preserves PoT-style-level accuracy while achieving about 3.1× latency speedup. In parallel serving experiments, it achieves about 2.8× throughput improvement over PoT-style. These results suggest that the sweet spot for small models in large-model inference systems lies not in solving complex tasks independently, but in performing lightweight, structured, and verifiable auxiliary operations.[1] The main contributions are summarized as follows:

- We propose an LLM inference optimization framework centered on reusable program caches. The framework converts PoT-style programs into parameterized cache objects for cross-request reuse, decouples computation logic from request-specific data, and unifies program reasoning, cache reuse, and generation acceleration in one inference system.
- We design a dual-role small-model reuse

[1] An anonymized implementation and experiment artifacts are available at https://anonymous.4open.science/r/cache-31E4.

mechanism. The same small model serves as a semantic variable extractor on the cache-hit path, enabling cached programs to adapt to structurally different but semantically related requests, and as a speculative drafter during target-LLM generation, reducing the cost of cache-miss inference and cache construction.

- We conduct extensive experiments across shopping-style request datasets, WebShop, Formula, and CodeTAT-QA, with additional analysis under long-context inputs and parallel serving. The results demonstrate the effectiveness of Ours in reducing inference latency and improving cache reuse while maintaining competitive task quality.

2 Related Work

Program-aided reasoning. Program-aided reasoning improves LLM reliability on complex reasoning tasks by using explicit intermediate reasoning or executable programs (Wei et al., 2022; Chen et al., 2023b; Gao et al., 2023; Yao et al., 2023). Chain-of-Thought prompting exposes intermediate reasoning steps, while Program of Thoughts and PAL translate natural-language problems into executable code that can be handled by external interpreters (Wei et al., 2022; Chen et al., 2023b; Gao et al., 2023). ReAct further extends this idea to interactive settings by interleaving reasoning traces with task-specific actions (Yao et al., 2023). Although these methods improve accuracy, they usually generate a new program for each request, causing repeated latency and token cost when requests share the same computation pattern but differ only in variables or surface forms.

Speculative decoding. Speculative decoding reduces autoregressive generation latency through a draft-and-verify paradigm, where a smaller or faster model proposes candidate tokens and the target LLM verifies them in parallel (Leviathan et al., 2023; Chen et al., 2023a). Recent variants improve this paradigm with different verification structures, drafting mechanisms, self-speculation, and lookahead decoding (Miao et al., 2024; Cai et al., 2024; Li et al., 2024; Zhang et al., 2024; Fu et al., 2024). These methods accelerate single target-model generation, but they do not directly exploit reusable computation structures across related requests, and some variants require additional draft models, auxiliary heads, or model-specific modifications.

Cache-based reuse. Cache-based reuse reduces redundant LLM inference by reusing previous outputs, semantically similar queries, or model states (Bang, 2023; Gill et al., 2024; Gim et al., 2024; Chakraborty et al., 2025). Existing methods include response-level and semantic caches, as well as model-state caches that reuse prompt segments or attention states (Bang, 2023; Gill et al., 2024; Gim et al., 2024). GenCache is closer to our setting because it studies generative reuse for structurally similar prompts and addresses the limitations of exact matching and purely semantic caching under request variations (Chakraborty et al., 2025). However, program-aided reasoning requires not only matching similar requests, but also constructing reusable programs or templates and reliably extracting and binding variables from new inputs, making robust program-level reuse challenging.

Unified small-model reuse. PoT-style reasoning, speculative decoding, and GenCache-style caching optimize LLM inference from program generation, generation acceleration, and structural reuse, respectively (Chen et al., 2023b; Leviathan et al., 2023; Chen et al., 2023a; Chakraborty et al., 2025). However, these mechanisms are often studied separately, leaving underexplored how they can be unified and where small models provide the highest leverage. We argue that PoT-style programs can serve not only as one-time reasoning artifacts but also as reusable cache objects for computation-oriented tasks. Under this view, small models can support both semantic variable extraction for reliable cache execution and speculative drafting for efficient cache construction.

3 Method

3.1 Overview

We propose an LLM inference optimization framework centered on reusable program caches, as shown in Figure 1. For each input request, the framework first performs request routing. If a matched group has a valid program cache, the request enters the cache-hit branch, where a reusable small model extracts variables and an executable cached program produces the output. Otherwise, the request enters the cache-miss branch, where the target LLM generates an answer, code, or program, with Speculative Decoding (SpecDec) used to reduce generation cost. When a request group accumulates enough examples, the framework attempts to construct a new program cache for future

reuse.

The framework follows a reuse-oriented design. Target-LLM-generated programs are converted from one-time reasoning artifacts into executable cache objects, and the same small model is reused for semantic variable extraction on the cache-hit path and speculative drafting during target-LLM generation. In this way, the small model supports reusable program caching through two lightweight operations: semantic variable extraction, which determines cache usability, and speculative drafting, which reduces the cost of remaining target-LLM generation.

3.2 Routing and Program Cache

Because many computation-oriented requests share executable structures, the framework routes each request before invoking the target LLM. Let $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$ denote the existing request groups. For an input request x, we use a semantic encoder $E(\cdot)$ to obtain its embedding $\mathbf{e}_x = E(x)$. We use `all-MiniLM-L6-v2` as the semantic encoder for semantic similarity search (Reimers and Gurevych, 2019; Wang et al., 2020). The system computes a semantic matching score $S(x, G_i)$ between x and each group G_i, where $S(\cdot)$ uses embedding similarity.

The system selects the group with the highest matching score, $i^* = \arg\max_i S(x, G_i)$. If $S(x, G_{i^*}) < \tau$, where τ denotes the semantic matching threshold, the request is sent to the cache-miss branch. Otherwise, the system checks whether G_{i^*} has a valid program cache.

We define a program cache entry as $C_i = (T_i, P_i)$. Here, T_i denotes a variable extraction template that specifies which variables the small model should extract and what format they should follow. P_i denotes an executable program that takes the extracted variables as input and produces the final output. If group G_{i^*} does not have a cache entry, the request enters the cache-miss branch; otherwise, it enters the cache-hit branch.

3.3 Cache-Hit Branch

A cached program can be reused only when the system correctly identifies and binds the required variables in a new request. Since surface-pattern-based extraction in GenCache-style reuse may fail for semantically similar but structurally different requests (Chakraborty et al., 2025), we use the small model for semantic variable extraction to improve the robustness of cached-program reuse.

For a matched cache entry $C_{i^*} = (T_{i^*}, P_{i^*})$, the system feeds the input request x and the extraction template T_{i^*} into the reusable small model M_s, which outputs a structured variable set $\mathbf{v} = M_s(x, T_{i^*})$. The variable set $\mathbf{v}$ may contain domain-specific entities, constraints, or numerical parameters depending on the template. The system then checks whether $\mathbf{v}$ contains all required fields and satisfies the expected types and formats. If the check fails, the request falls back to the cache-miss branch.

If the extraction is valid, the system binds $\mathbf{v}$ to the executable program P_{i^*} and returns $y = P_{i^*}(\mathbf{v})$. Thus, the small model performs a lightweight but high-leverage auxiliary task: it extracts the variables required for safe program execution, while the cached program preserves the reusable computation logic. The cache-hit path can also batch variable extraction under concurrent serving; implementation details are provided in Appendix A.1.

3.4 Cache-Miss Branch

The cache-miss branch handles requests that cannot be safely served by the Program Cache, including cases where no matching group exists, the matched group has no generated cache, variable extraction fails, or the task requires new reasoning. For an input request x in this branch, the target LLM M_t generates an answer, code, or executable program. We use the same small model M_s as the speculative drafter during target-LLM invocation. Following the draft-and-verify mechanism in SpecDec, the small model proposes candidate tokens or steps, and the target LLM verifies, corrects, or continues the generation (Leviathan et al., 2023; Chen et al., 2023a).

The output form depends on the task. For computation-oriented or structured reasoning tasks, the target LLM can generate code or a program whose execution produces the final result. For tasks that do not require execution, it can directly generate a natural-language answer or an agent action. This fallback branch preserves the reasoning capability of the target LLM while reducing the cost of necessary target-LLM calls through speculative drafting.

3.5 Cache Generation

The framework generates program caches so that target-LLM outputs do not remain one-time reasoning artifacts. For a request group G_i, let $\mathcal{D}_i = \{(x_j, y_j)\}_{j=1}^{m_i}$ denote its accumulated examples,

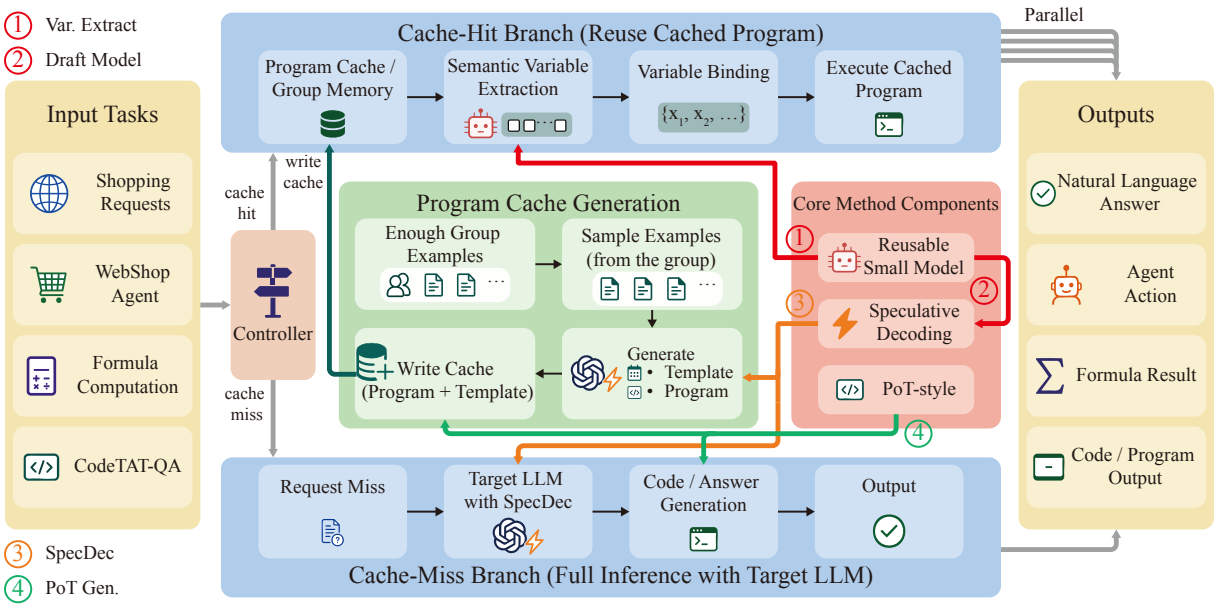


Figure 1: Overview of the proposed inference framework. The numbered arrows denote: (1) semantic variable extraction by the reusable small model, (2) the same small model as the speculative drafter, (3) SpecDec-accelerated target-LLM calls, and (4) PoT-style program generation for answers and cache construction.

where x_j is a request and y_j is the corresponding output. Each group has a dynamic cache generation threshold ν_i. When $m_i \geq \nu_i$, the system triggers cache generation for group G_i.

During one cache generation attempt, the system samples examples from $\mathcal{D}_i$ for generation and validation. The generation examples prompt the target LLM to infer the shared structure of the group and generate a cache entry $C_i = (T_i, P_i)$. The target-LLM call that generates T_i and P_i also uses M_s as the speculative drafter. The generated cache entry is then evaluated on validation examples: for each validation request, the small model extracts variables according to T_i, and the program P_i executes with the extracted variables. Let r_i denote the validation pass rate and α denote the required pass-rate threshold.

If $r_i \geq \alpha$, the system writes $C_i = (T_i, P_i)$ into the Program Cache, so future requests routed to G_i can reuse this cache entry through the cache-hit branch. If cache generation fails, the system retries up to R times and uses failure summaries as reflection for subsequent attempts. If repeated failures indicate that the group lacks a stable shared computational structure, the group is marked as uncacheable and future requests in this group continue to use the cache-miss branch.

For groups that remain potentially cacheable, we adopt exponential backoff to avoid repeatedly spending target-LLM calls on groups that are not ready for reliable cache construction. If group G_i fails to generate a valid cache under the current threshold ν_i, the system increases the next trigger threshold by updating $\nu_i \leftarrow \beta\nu_i$, where $\beta > 1$ denotes the backoff coefficient. Detailed procedures for cache validation, retry, reflection, uncacheable-group marking, and backoff are provided in Appendix A.2.

4 Experiments

4.1 Experimental Setup

We implement an API-based evaluation framework that wraps different inference methods as OpenAI-compatible `/v1/chat/completions` services. Each method receives the same input format, and a unified runner sends requests, records request-level logs, saves cache states and task outputs, and computes evaluation metrics with the same scripts. This setup allows us to compare Direct LLM, PoT-style code generation, ExactCache, GPTCache, GenCache, Ours, and SpecDec variants under a shared serving and evaluation pipeline.

We use Qwen3-32B as the target LLM and Qwen3-1.7B as the reusable small model (Yang et al., 2025). We use Qwen3-1.7B without task-specific fine-tuning. The small model serves two roles: semantic variable extraction in the cache-hit branch and speculative drafting during target LLM invocation. For request-group matching, we use `all-MiniLM-L6-v2` as the semantic en-

coder, following sentence embedding and compact transformer models for semantic similarity search (Reimers and Gurevych, 2019; Wang et al., 2020).

For methods that support SpecDec, we evaluate both non-SpecDec and SpecDec variants. The non-SpecDec variants directly invoke Qwen3-32B for generation, while the SpecDec variants use Qwen3-1.7B as the draft model. The main tables report representative +SpecDec variants for readability, while the complete +SpecDec results are provided in Appendix A.6. Except for the parallel experiments, all experiments run in a single-request setting. For hyperparameters related to semantic matching, cache generation, validation, retry, and backoff, we keep the settings fixed within each task to ensure fair comparison across methods.

4.2 Datasets

We evaluate on five groups of tasks. We use the Shopping-Full and Shopping-Struct request datasets introduced by GenCache (Chakraborty et al., 2025). Shopping-Full contains 10,136 shopping-style requests, while Shopping-Struct contains 10,000 structurally perturbed requests that preserve the same semantic intent but change the prompt surface form. WebShop evaluates agent-style shopping behavior in a simulated e-commerce environment with real product data and natural-language goals (Yao et al., 2022). For financial reasoning, we use Formula from FinLoRA (Wang et al., 2025) and CodeTAT-QA from BizBench (Krumdick et al., 2024). Formula focuses on formula construction and calculation over XBRL financial data, whereas CodeTAT-QA involves more diverse table-and-text reasoning. We further construct length-controlled Formula variants with average input lengths near 1K, 2K, 4K, and 8K tokens to test robustness under long contexts. Detailed dataset construction and filtering procedures are provided in Appendix A.3.

4.3 Baselines and Metrics

We compare Direct LLM, PoT-style code generation, ExactCache, GPTCache, GenCache (Chakraborty et al., 2025), and Ours. For GenCache, we use a faithful reproduction of the original regex/matcher-based program-cache pipeline and adapt only task-specific input/output interfaces. For methods that involve target-LLM generation, we also evaluate +SpecDec variants. PoT-style is used only for Formula and CodeTAT-QA, where executable program generation is a natural baseline. For Shopping-Full, Shopping-Struct, Formula, and CodeTAT-QA, we report answer accuracy; for WebShop, we report average reward. Across tasks, Lat. denotes average end-to-end request latency. For cache-based methods, Hit denotes cache hit rate and Hit Acc. denotes the accuracy of cache-hit outputs. For parallel experiments, we additionally report throughput and wall-clock time per request. Additional baseline definitions and metric details are provided in Appendix A.4.

4.4 Main Results

Table 2 and Table 3 report the main results. Overall, Ours improves the latency–reuse trade-off while maintaining competitive task quality. On Shopping-Full and Shopping-Struct, it achieves high Hit and Hit Acc. with lower latency than Direct LLM, showing that small-model variable extraction enables reliable program reuse under request variations. GPTCache also obtains high Hit but poor quality, indicating that full-prompt semantic similarity alone is insufficient for safe reuse, while GenCache has limited reuse under structural perturbations. On financial reasoning tasks, Ours improves accuracy over Direct LLM through PoT-style executable reasoning and reduces latency through program reuse. More detailed analysis is provided in Appendix A.7.

The financial reasoning tasks further show the advantage of converting PoT-style programs into reusable cache objects. On Formula, PoT-style improves Acc. from 75.92 to 94.69 but increases Lat. from 0.237s to 2.015s. Ours + SpecDec preserves PoT-style-level accuracy, achieving 94.19 Acc., while reducing Lat. to 0.648s with 87.16 Hit and 93.76 Hit Acc. Compared with the reproduced GenCache baseline, which reaches 94.52 Acc. but has 0.00 Hit on Formula, Ours obtains effective cache reuse rather than relying mainly on fallback generation. On CodeTAT-QA, where group-level computation patterns are less stable, Ours still serves more than half of the requests through cache, but with a slight accuracy drop, indicating that reusable program caching is most effective when tasks contain stable computation structures.

SpecDec mainly helps when target-LLM generation is long enough to amortize draft-and-verify overhead, such as PoT-style code generation, cache-miss inference, and cache construction. This supports our design choice of reusing the same small model not as a direct replacement for the target

Table 2: Main results on Shopping-Full, Shopping-Struct, and WebShop. Acc. denotes accuracy, Reward denotes average WebShop reward, Lat. denotes average end-to-end latency, and Hit/Hit Acc. denote cache hit rate and cache-hit accuracy.

Method	**Shopping-Full**				**Shopping-Struct**				**WebShop**		
	Acc.	Lat.	Hit	Hit Acc.	Acc.	Lat.	Hit	Hit Acc.	Reward	Lat.	Hit
Direct LLM	92.71	0.915	–	–	97.93	0.913	–	–	0.5728	1.4070	–
Direct LLM + SpecDec	**92.73**	0.419	–	–	**97.94**	0.475	–	–	0.5762	1.1364	–
ExactCache	92.72	0.972	0.00	–	97.93	0.971	0.00	–	**0.5832**	1.4515	0.00
GPTCache	8.11	0.108	91.80	0.02	17.18	0.150	89.17	7.13	0.0531	0.1942	89.33
GenCache	78.51	0.197	94.89	77.68	96.91	3.252	0.10	0.00	0.5600	0.7264	70.52
Ours	91.91	0.256	**96.80**	91.89	96.65	0.249	**98.03**	96.62	0.5763	0.6598	74.22
Ours + SpecDec	92.15	**0.214**	96.35	**92.13**	96.66	**0.228**	**98.03**	**96.64**	0.5760	**0.5797**	74.22

Table 3: Main results on Formula and CodeTAT-QA. Acc. denotes answer accuracy, Lat. denotes average end-to-end latency, and Hit/Hit Acc. denote cache hit rate and cache-hit accuracy.

Method	**Formula**				**CodeTAT-QA**			
	Acc.	Lat.	Hit	Hit Acc.	Acc.	Lat.	Hit	Hit Acc.
Direct LLM	75.92	0.237	–	–	49.17	0.258	–	–
Direct LLM + SpecDec	75.80	0.314	–	–	49.24	0.472	–	–
PoT-style	**94.69**	2.015	–	–	71.53	2.447	–	–
PoT-style + SpecDec	94.56	1.564	–	–	71.69	1.880	–	–
ExactCache	94.64	2.056	0.00	–	**71.98**	2.550	0.08	100.00
GPTCache	94.56	0.858	63.45	93.69	42.54	0.485	83.03	35.73
GenCache	94.52	3.678	0.00	–	71.69	3.232	0.00	–
Ours	94.19	0.847	84.14	93.54	71.25	1.849	56.05	68.00
Ours + SpecDec	94.19	**0.648**	**87.16**	**93.76**	70.41	**1.648**	**57.48**	**71.79**

Table 4: Ablation study on Formula. Cache Gen. reports successful cache generations over attempted cache generations.

Method	Acc.	Lat.	Hit	Hit Acc.	Cache Gen.
Direct LLM	75.92	0.237	–	–	–
PoT-style	**94.69**	2.015	–	–	–
PoT-style + SpecDec	94.56	1.564	–	–	–
GenCache	94.52	3.678	0.00	–	5/133
Ours	94.19	0.847	84.14	93.54	23/38
Ours + SpecDec	94.19	**0.648**	**87.16**	**93.76**	**24/38**

LLM, but as both a semantic variable extractor and a speculative drafter.

4.5 Ablation Study

Table 4 reports the ablation study on Formula, the clearest setting for analyzing program-aided reasoning, cache generation, and cached-program execution.

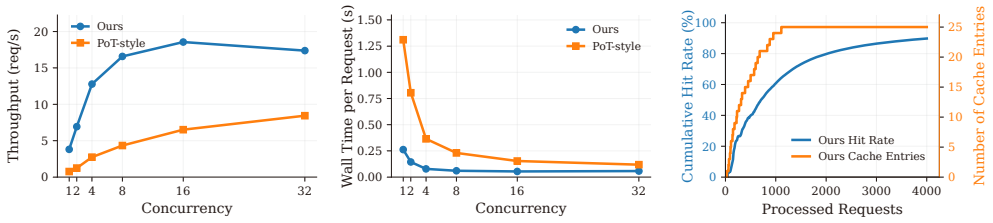
The ablation results show how the three components support Ours. PoT-style code generation provides accurate executable reasoning but incurs high latency. GenCache reaches PoT-style-level accuracy but generates few valid caches and obtains 0.00 Hit, meaning that its accuracy mainly comes from fallback code generation. In contrast, Ours increases Cache Gen. from 5/133 to 23/38, obtains 84.14 Hit with 93.54 Hit Acc., and reduces Lat. to 0.847s. Adding SpecDec further reduces Lat. to 0.648s. These results show that small-model semantic extraction turns generated programs into reusable caches, while speculative drafting reduces the remaining target-LLM cost.

4.6 Parallel Performance

We evaluate parallel performance on Formula by comparing Ours with PoT-style. As shown in Figure 2, Ours achieves higher throughput and lower wall-clock time per request across concurrency levels. At concurrency 16, it reaches 18.56 req/s, compared with 6.51 req/s for PoT-style, giving about 2.85× throughput improvement. This advantage comes from replacing repeated target-LLM program generation with variable extraction and cached-program execution after cache warm-up. Detailed throughput curves and cache warm-up dynamics are provided in Appendix A.8.

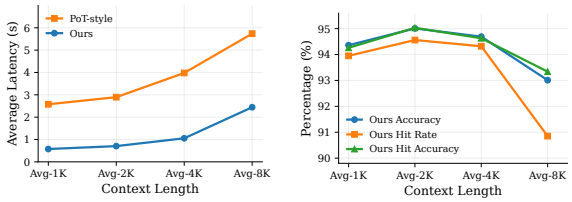
4.7 Length Robustness

Figure 3 evaluates whether program caching remains effective as the input context becomes longer. Ours consistently outperforms PoT-style across Avg-1K to Avg-8K Formula settings, with latency speedups of 4.49×, 4.11×, 3.77×, and 2.35×, respectively. Although the speedup decreases at Avg-



(a) Throughput. (b) Wall time per request. (c) Cache warm-up dynamics.

Figure 2: Parallel performance and cache warm-up on Formula. (a) Throughput under different concurrency levels. (b) Wall-clock time per request under different concurrency levels. (c) Cache warm-up dynamics at concurrency 8.



(a) Avg. latency. (b) Quality.

Figure 3: Length robustness on Formula.

8K, Ours still maintains 93.01% accuracy, 90.85% cache hit rate, and 93.34% cache-hit accuracy, showing that reusable program caching remains effective under long contexts when the task has stable formula-level computation structures. Detailed analysis is provided in Appendix A.9.

5 Conclusion

We present an LLM inference optimization framework centered on reusable program caches. Instead of treating program-aided reasoning, cache reuse, and speculative decoding as separate mechanisms, our framework organizes them around a unified reuse principle: programs generated by the target LLM become executable cache objects, and the small model is reused for both semantic variable extraction and speculative drafting. This design allows the system to avoid repeated target-LLM program generation on cache-hit requests while reducing the cost of necessary target-LLM calls during cache-miss inference and cache construction.

Experiments on shopping-style request datasets, WebShop, Formula, and CodeTAT-QA show that our framework reduces inference latency and improves effective cache reuse while preserving competitive task quality compared with exact matching, semantic response caching, and generative program caching baselines. The results further show that reusable program caches remain effective under longer contexts and concurrent requests when tasks contain stable computation patterns. Overall, our findings suggest that the sweet spot for small models in large-model inference systems lies not in solving complex tasks independently, but in performing lightweight, structured, and verifiable auxiliary operations.

Limitations

Ours has three main limitations. First, it is most effective when requests share reusable structured patterns. Although small-model variable extraction improves reuse for semantically similar but structurally different requests, the framework remains less suitable for free-form dialogue, creative writing, or tasks with high prompt diversity. This is consistent with prior generative caching work, which also scopes caching to structurally similar repetitive tasks rather than free-form chatbot interactions (Chakraborty et al., 2025).

Second, the method depends on both the target LLM and the small model. The target LLM must induce a valid extraction template and executable program from grouped examples, while the small model must reliably extract variables from new requests. If the group pattern is unstable, the generated program is incomplete, or the input contains long contexts and ambiguous variables, cache generation or cache-hit execution can become unreliable.

Third, program caching may propagate errors when the system constructs an incorrect cache or routes a request to an unsuitable cache entry. Prior work also notes this risk and recommends strict guardrails for generative caching (Chakraborty et al., 2025). We reduce such failures with validation, fallback execution, failure reflection, exponential backoff, and uncacheable-group marking, but these mechanisms cannot eliminate all erroneous reuse.

References

Fu Bang. 2023. 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)*, pages 212–218, Singapore. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 5209–5235. PMLR.

Sarthak Chakraborty, Suman Nath, Xuchao Zhang, Chetan Bansal, and Indranil Gupta. 2025. Generative caching for structurally similar prompts and responses. In *Advances in Neural Information Processing Systems*, volume 38, pages 4598–4625.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023a. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023b. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of LLM inference using lookahead decoding. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 14060–14079. PMLR.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Waris Gill, Mohamed Elidrisi, Pallavi Kalapatapu, Ammar Ahmed, Ali Anwar, and Muhammad Ali Gulzar. 2024. MeanCache: User-centric semantic cache for large language model based web services. *arXiv preprint arXiv:2403.02694*.

In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems*. MLSys.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, and 1 others. 2022. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 30016–30030.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Michael Krumdick, Rik Koncel-Kedziorski, Viet Dac Lai, Varshini Reddy, Charles Lovering, and Chris Tanner. 2024. Bizbench: A quantitative reasoning benchmark for business and finance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8309–8332, Bangkok, Thailand. Association for Computational Linguistics.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286. PMLR.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 28935–28948. PMLR.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating generative large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3982–3992. Association for Computational Linguistics.

Dannong Wang, Jaisal Patel, Daochen Zha, Steve Y. Yang, and Xiao-Yang Liu. 2025. Finlora: Benchmarking lora methods for fine-tuning llms on financial datasets. *arXiv preprint arXiv:2505.19819*.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems*, volume 33, pages 5776–5788.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, volume 35, pages 20744–20757.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2024. Draft & verify: Lossless large language model acceleration via self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11263–11282, Bangkok, Thailand. Association for Computational Linguistics.

A Additional Experimental Details

A.1 Cache-Hit Execution Details

The cache-hit branch is designed to reuse cached programs without invoking the target LLM. Given a matched cache entry $C_{i^*} = (T_{i^*}, P_{i^*})$, the small model receives the input request x together with the variable extraction template T_{i^*}. The template specifies the required variable names, expected types, and output format. The small model then returns a structured variable set $\mathbf{v} = M_s(x, T_{i^*})$.

Before executing the cached program, the system validates the extracted variable set. The validation checks whether all required fields specified by T_{i^*} are present, whether the extracted values satisfy the expected types and formats, and whether they can be safely bound to the program inputs. If any required field is missing, if a type constraint is violated, or if the output format cannot be parsed, the extraction is marked as invalid and the request falls back to the cache-miss branch.

When validation succeeds, the system binds $\mathbf{v}$ to the executable program P_{i^*} and returns $y = P_{i^*}(\mathbf{v})$. This design separates semantic variable extraction from deterministic program execution: the small model handles the flexible natural-language interface, while the cached program preserves the reusable computation logic.

The cache-hit path also supports concurrent serving. Multiple variable extraction requests can be batched and sent to the small model together. After extraction, cached programs are executed independently for different requests. This batching reduces small-model overhead under concurrent workloads, while program execution avoids repeated target-LLM calls. This implementation is the basis for the parallel serving behavior analyzed in the main experiments.

A.2 Cache Generation and Failure Handling Details

For each request group G_i, the system maintains an accumulated example set $\mathcal{D}_i = \{(x_j, y_j)\}_{j=1}^{m_i}$ and a dynamic cache generation threshold ν_i. When $m_i \geq \nu_i$, the group becomes eligible for cache generation. In each cache generation attempt, the system samples examples from $\mathcal{D}_i$ and splits them into generation examples and validation examples. The generation examples are used to prompt the target LLM to infer the shared computation pattern of the group and produce a cache entry $C_i = (T_i, P_i)$.

The generated template T_i should describe the variables that must be extracted from future requests, including their names, expected types, and output structure. The generated program P_i should implement the reusable computation logic shared by the group and take the extracted variables as input. The target-LLM call used for cache generation is also accelerated by SpecDec, where the same small model M_s acts as the speculative drafter.

After a candidate cache entry is generated, the system evaluates it on validation examples. For each validation request, the small model extracts variables according to T_i, the program P_i executes with the extracted variables, and the system checks whether the output satisfies the expected answer or output format. The validation pass rate is denoted as r_i. If $r_i \geq \alpha$, where α is the required pass-rate

threshold, the cache entry is accepted and written into the Program Cache.

If validation fails, the system retries cache generation up to R times. After each failed attempt, the target LLM summarizes the failure reasons. Typical failure reasons include incomplete variable definitions, inconsistent program logic, incorrect output format, invalid variable binding, or poor coverage of the validation examples. The failure summary is used as reflection to guide the next cache generation attempt, so later attempts can revise the extraction template or program logic according to previous errors.

If all R attempts fail, the system records the failure summaries and reuses them when the group triggers cache generation again. When repeated failures indicate that the group lacks a stable shared computational structure, the group is marked as uncacheable. For an uncacheable group, the system disables future cache generation attempts and sends future requests in the group to the cache-miss branch. This prevents the framework from repeatedly spending target-LLM calls on groups that are unlikely to yield reliable program caches.

For groups that are not marked as uncacheable but still fail to produce a valid cache, the system applies exponential backoff. If group G_i fails to generate a valid cache under the current threshold ν_i, the next trigger threshold is increased as

$$\nu_i \leftarrow \beta \nu_i,$$

where $\beta > 1$ is the backoff coefficient. This mechanism allows difficult groups to accumulate more examples before the next cache generation attempt and reduces wasted target-LLM computation during repeated unsuccessful cache construction.

A.3 Dataset Construction Details

Shopping request datasets. We use the Shopping-Full and Shopping-Struct request datasets introduced by GenCache (Chakraborty et al., 2025). Shopping-Full contains 10,136 shopping-style requests. Each example includes a user instruction, the complete prompt sent to the model, and a reference action or answer. This dataset is used to evaluate whether different caching methods can reuse recurring shopping-style request patterns under the original prompt format.

Shopping-Struct is designed by GenCache to test robustness under structural perturbations. Starting from Shopping-Full, GenCache constructs rewritten versions that modify surface realization, field order, prompt format, or partial descriptions while aiming to preserve the same user intent. The dataset retains rewritten examples whose answers are judged to be equivalent to the original reference answer. This setting evaluates whether a cache method can reuse computation across prompts that are semantically equivalent but structurally different.

Table 5: Example of a Shopping-Struct perturbation.

Field	Example
Source instruction	i want a x-large and machine washable lazy tuxedo t-shirt, and price lower than 261 dollars
Rewritten instruction	Could you help me find a lazy tuxedo t-shirt in x-large, machine washable, and under 261 dollars?
Reference action	`Search(keywords=lazy tuxedo t-shirt x-large machine washable, max_price=261)`
Equivalence check	Qwen3-32B judges that the rewritten request preserves the same shopping intent and the same maximum-price constraint.

Table 5 gives a real example from Shopping-Struct. The rewritten instruction changes the surface form of the request, but the search intent and price constraint remain unchanged. The example is retained because the generated action is equivalent to the original reference action.

WebShop. WebShop is a simulated e-commerce environment with real product data and crowd-sourced instructions (Yao et al., 2022). Each episode requires an agent to search, inspect products, navigate pages, and select an item according to a natural-language goal. We evaluate 50 episodes using fixed product data, search indices, and task goals. Since WebShop outputs are mainly search, click, navigation, and selection actions, we do not evaluate PoT-style code generation on this task. We compare Direct LLM, ExactCache, GPTCache, GenCache, Ours, and applicable SpecDec variants.

Financial reasoning tasks. We use Formula from FinLoRA (Wang et al., 2025) and CodeTAT-QA from BizBench (Krumdick et al., 2024). Formula contains Formula Construction and Formula Calculation tasks. These tasks are based on financial formulas and XBRL financial data: Formula Construction requires constructing financial formulas from relevant information, while Formula Calculation requires substituting numerical values into formulas and computing the final result. Formula is well suited for evaluating reusable program caches, because requests from the same formula group often share computation logic but differ in

Table 6: Example of a Formula variant.

Field	Example
Formula group	Modigliani–Miller Theorem
Formula	$V_L = V_U + T_c D$
Seed question	Assuming a company without any debt is valued at $950,000, what would the value be should it decide to leverage $150,000 at a 32% corporate tax rate?
Rewritten question	Compute the value of a levered firm given an unlevered value of $950,000, debt of $150,000, and a corporate tax rate of 32%.
Reference answer	998000.0

variable values. The original Formula dataset contains relatively few examples, so we expand it by using each original example as a seed and modifying non-critical fields, rephrasing questions, and changing input order while preserving the core formula logic and final answer. We use the locally deployed Qwen3-32B model without SpecDec to filter examples whose answers remain consistent with the seed.

Table 6 shows a real Formula example after rewriting. The example belongs to the Modigliani–Miller Theorem group. Although the question is rephrased, the underlying computation is unchanged: the model must substitute the unlevered firm value, tax rate, and debt into $V_L = V_U + T_c D$ and return the same numerical answer. Such examples allow us to test whether a cache method can learn a reusable computation pattern for a formula group rather than memorize a single prompt.

CodeTAT-QA is a more diverse financial table-and-text question answering task. Because the original data has weaker group-level regularities, we first cluster the dataset with `all-MiniLM-L6-v2` sentence embeddings and semantic similarity (Reimers and Gurevych, 2019; Wang et al., 2020). We then select large clusters to form a cacheable subset and randomly shuffle the selected examples. This construction tests whether reusable program caches remain useful when the task contains more complex financial reasoning and less regular request groups.

Length-controlled Formula datasets. To evaluate robustness under longer contexts, we construct four Formula variants with average input lengths near 1K, 2K, 4K, and 8K tokens. We add extra context, table fields, irrelevant text, or weakly related financial descriptions while preserving the original question logic, formula structure, and final answer. These datasets allow us to compare how Direct LLM, PoT-style code generation, and Ours behave as context length increases. PoT-style methods must generate a new program over the full input context for every request, whereas cache-hit requests in Ours only require variable extraction and cached program execution.

A.4 Baseline and Metric Details

Baselines. Direct LLM directly invokes the target model to generate the final answer. PoT-style denotes a program-generation-and-execution baseline, following the general idea of program-aided reasoning methods such as Program-of-Thoughts and PAL (Chen et al., 2023b; Gao et al., 2023). In this baseline, the target LLM generates executable code or a program, and the system executes it to obtain the final answer. We apply PoT-style only to Formula and CodeTAT-QA, where program execution is a natural fit.

ExactCache reuses cached responses only when the complete prompt exactly matches a previous prompt. GPTCache performs semantic response caching based on prompt embeddings, following the broader idea of embedding-based cache retrieval (Bang, 2023). GenCache generates reusable program caches for structurally similar requests (Chakraborty et al., 2025). Ours also uses reusable program caches, but differs by parameterizing cached programs and using a small model to extract semantic variables on the cache-hit path.

For methods that involve target-LLM generation, we additionally evaluate +SpecDec variants based on speculative drafting (Leviathan et al., 2023; Chen et al., 2023a). In Ours + SpecDec, the same small model serves two roles: it extracts semantic variables for cache-hit requests and acts as the speculative drafter during target-LLM generation. For Formula and CodeTAT-QA, cache-miss requests use code generation and execution, and successful generated programs can become candidates for future cache construction.

A.5 GenCache Reproduction and Stability

We use GenCache as the closest program-cache baseline and implement it as a faithful reproduction of the original pipeline (Chakraborty et al., 2025). The reproduced baseline preserves the original regex/matcher-based variable extraction, request grouping, program-cache generation, cache validation, and fallback execution logic. For each task, we only adapt the input/output interface so that GenCache receives the same task prompts and is evaluated with the same response format as the other methods. ExactCache, GPTCache, GenCache, and Ours are all served through the same

Table 7: Stability of the reproduced GenCache baseline on Shopping-Full.

Setting	Runs	Acc.	Hit Acc.	Cache Gen.	Avg. Lat.
B1, non-SpecDec	4	80.58	80.09	27.41	0.069–0.197
B3, SpecDec	4	58.97	57.77	26.71	0.040–0.087

API wrapper and evaluated with the same request-level logging and correctness scripts.

This task adaptation is necessary because the evaluated tasks differ in output format: Shopping-Full and Shopping-Struct use shopping-style actions or answers, WebShop uses agent observations and actions, and Formula and CodeTAT-QA use program-generation or answer-checking interfaces. These adapters do not change GenCache's core cache construction or variable extraction mechanism. Therefore, the GenCache results should be interpreted as the performance of its original reuse mechanism under these task interfaces, rather than as a weakened baseline.

Table 7 summarizes repeated runs of the original GenCache branch on Shopping-Full. In the non-SpecDec setting (B1), cache-hit accuracy remains within 77.68–82.79%, while overall accuracy ranges from 78.51% to 83.13%. In the SpecDec setting (B3), cache-hit accuracy ranges from 42.59% to 71.25%, and overall accuracy ranges from 45.18% to 71.94%. Thus, although SpecDec reduces latency for the reproduced GenCache branch, it also introduces substantial variance in cache-hit correctness under this setting.

The same reproduced GenCache implementation also struggles to construct stable reusable caches on structurally perturbed or program-execution-oriented tasks. On Shopping-Struct, GenCache obtains only 0.10% Hit without SpecDec and 1.46% Hit with SpecDec. On Formula, GenCache reaches high answer accuracy through the fallback code-generation path, but its Hit is 0.00. These results do not indicate that GenCache is an invalid baseline; rather, they show that the original regex/matcher-based reuse mechanism is brittle when prompts are structurally perturbed or when reusable programs require robust semantic variable extraction.

Metrics. For Shopping-Full, Shopping-Struct, Formula, and CodeTAT-QA, we use answer accuracy as the quality metric. For WebShop, we report average reward, following the environment's task-level evaluation protocol (Yao et al., 2022). Lat. denotes average per-request end-to-end latency in seconds, measured from request submission to response return.

For cache-based methods, Hit denotes the fraction of requests served by cache. Hit Acc. denotes the fraction of correct outputs among cache-hit requests. This metric is important because high cache hit rate alone does not guarantee reliable reuse: a method may frequently retrieve semantically similar but nonequivalent examples, leading to high Hit but low Hit Acc. and poor overall task quality.

For parallel experiments, Throughput denotes the number of completed requests per second. Wall-clock time per request denotes total wall-clock time divided by the number of completed requests and reflects throughput-side system efficiency under concurrent serving. It differs from Lat., which measures the average end-to-end latency of individual requests. For cache warm-up curves, we use the number of processed requests as the x-axis, which makes cache construction and cache-hit dynamics comparable across concurrency settings with different total running times.

A.6 Full SpecDec and Non-SpecDec Results

Table 8 reports the complete non-SpecDec and SpecDec results on Shopping-Full, Shopping-Struct, and WebShop. Table 9 reports the complete +SpecDec variants on the financial reasoning tasks. The main text shows representative rows to keep the tables compact.

A.7 Detailed Main Result Analysis

On Shopping-Full, Ours substantially reduces latency while maintaining task quality close to Direct LLM. With SpecDec, the framework reaches 92.15 Acc. and 0.214s Lat., compared with 92.71 Acc. and 0.915s Lat. for Direct LLM. Thus, the result should not be interpreted as improving the target LLM's intrinsic accuracy. Instead, the key evidence is that most requests are served by cache (96.35 Hit) and that these cache-hit outputs remain reliable (92.13 Hit Acc.), indicating that reusable programs can replace repeated target-LLM generation with only a small quality difference on Shopping-Full.

Shopping-Struct stresses robustness under structural perturbations. GenCache obtains high task quality on this dataset, but its Hit is only 0.10, indicating that it mainly falls back to target-LLM generation rather than reusing cache entries. In contrast, Ours with SpecDec achieves 98.03 Hit and 96.64 Hit Acc., while reducing Lat. from Direct LLM's 0.913s to 0.228s. This comparison directly supports the role of semantic variable extraction: it

Table 8: Complete non-SpecDec and SpecDec results on Shopping-Full, Shopping-Struct, and WebShop.

Method	Shopping-Full				Shopping-Struct				WebShop		
	Acc.	Lat.	Hit	Hit Acc.	Acc.	Lat.	Hit	Hit Acc.	Reward	Lat.	Hit
Direct LLM	92.71	0.915	–	–	97.93	0.913	–	–	0.5728	1.4070	–
ExactCache	92.72	0.972	0.00	–	97.93	0.971	0.00	–	0.5832	1.4515	0.00
GPTCache	8.11	0.108	91.80	0.02	17.18	0.150	89.17	7.13	0.0531	0.1942	89.33
GenCache	78.51	0.197	94.89	77.68	96.91	3.252	0.10	0.00	0.5600	0.7264	70.52
Ours	91.91	0.256	96.80	91.89	96.65	0.249	98.03	96.62	0.5763	0.6598	74.22
Direct LLM + SpecDec	92.73	0.419	–	–	97.94	0.475	–	–	0.5762	1.1364	–
ExactCache + SpecDec	92.70	0.584	0.00	–	97.94	0.527	0.00	–	0.5762	1.1629	0.00
GPTCache + SpecDec	8.11	0.073	91.80	0.02	17.16	0.096	89.17	7.11	0.0531	0.1667	89.33
GenCache + SpecDec	59.09	0.040	98.95	58.67	95.99	1.852	1.46	23.29	0.5680	0.6298	66.40
Ours + SpecDec	92.15	0.214	96.35	92.13	96.66	0.228	98.03	96.64	0.5760	0.5797	74.22

Table 9: Complete non-SpecDec and SpecDec results on Formula and CodeTAT-QA.

Method	Formula				CodeTAT-QA			
	Acc.	Lat.	Hit	Hit Acc.	Acc.	Lat.	Hit	Hit Acc.
Direct LLM	75.92	0.237	–	–	49.17	0.258	–	–
PoT-style	94.69	2.015	–	–	71.53	2.447	–	–
ExactCache	94.64	2.056	0.00	–	71.98	2.550	0.08	100.00
GPTCache	94.56	0.858	63.45	93.69	42.54	0.485	83.03	35.73
GenCache	94.52	3.678	0.00	–	71.69	3.232	0.00	–
Ours	94.19	0.847	84.14	93.54	71.25	1.849	56.05	68.00
Direct LLM + SpecDec	75.80	0.314	–	–	49.24	0.472	–	–
PoT-style + SpecDec	94.56	1.564	–	–	71.69	1.880	–	–
ExactCache + SpecDec	94.64	1.650	0.00	–	72.03	1.720	0.08	100.00
GPTCache + SpecDec	94.56	0.700	63.45	93.69	42.36	0.331	83.03	35.57
GenCache + SpecDec	94.56	2.873	0.00	–	71.67	3.042	0.00	–
Ours + SpecDec	94.19	0.648	87.16	93.76	70.41	1.648	57.48	71.79

allows the system to reuse cached programs even when requests preserve semantic intent but change surface structure.

GPTCache serves as an instructive counterexample. It reaches high Hit on Shopping-Full and Shopping-Struct, but its task quality collapses. This shows that full-prompt semantic similarity alone is insufficient for safe reuse: similar prompts may still require different variable bindings or different actions. Ours avoids this failure mode by extracting task variables before executing a cached program, which leads to high Hit and high Hit Acc. simultaneously.

On WebShop, Ours maintains the same level of task reward as Direct LLM while reducing latency. With SpecDec, the framework reaches 0.5760 average reward with 0.5797s latency, compared with 0.5728 reward and 1.4070s latency for Direct LLM. GPTCache again shows that high cache hit rate is not enough: despite 89.33 Hit, its reward drops to 0.0531. The identical Hit of Ours and Ours + SpecDec is expected because SpecDec changes the target-LLM generation path but not the cache matching policy. ExactCache obtains a slightly higher reward in this 50-episode sample, but its Hit is 0.00; therefore, the result reflects fallback to the Direct LLM path rather than effective exact-cache reuse.

On Formula, PoT-style substantially improves accuracy over Direct LLM, increasing Acc. from 75.92 to 94.69, but it also increases latency from 0.237s to 2.015s. Ours preserves PoT-style-level accuracy while reducing latency through program reuse. Ours + SpecDec achieves 94.19 Acc. with 0.648s latency, giving a large reduction compared with PoT-style while maintaining 87.16 Hit and 93.76 Hit Acc. This indicates that cached programs can serve as reusable reasoning artifacts for formula computation tasks.

The comparison with GenCache is central. On Formula, GenCache reaches 94.52 Acc., but its Hit is 0.00. Thus, its accuracy mainly comes from the fallback code-generation path rather than from cache reuse. Ours, by contrast, achieves high Hit and high Hit Acc., showing that small-model semantic variable extraction makes the program cache executable for future requests. A similar pattern appears on CodeTAT-QA: GenCache again has

0.00 Hit, while Ours + SpecDec reaches 57.48 Hit and 71.79 Hit Acc.

CodeTAT-QA is more challenging than Formula because it involves table-and-text reasoning and weaker group-level regularities. Ours + SpecDec obtains slightly lower accuracy than PoT-style + SpecDec, but reduces latency from 1.880s to 1.648s while serving more than half of the requests through cache. The slight accuracy drop on CodeTAT-QA suggests that program caching requires sufficiently stable group-level computation patterns to fully preserve PoT-style performance.

SpecDec is useful mainly when target-LLM generation is long enough to amortize draft-and-verify overhead. In the direct-output setting, Direct LLM + SpecDec can be slower than Direct LLM: on Formula, latency increases from 0.237s to 0.314s, and on CodeTAT-QA, it increases from 0.258s to 0.472s. In contrast, PoT-style + SpecDec reduces latency compared with PoT-style, and Ours + SpecDec further reduces latency compared with Ours. This supports our design choice of reusing the small model as a speculative drafter for code generation and cache construction, rather than treating SpecDec as a universal speedup for every output length.

A.8 Parallel Performance Details

As shown in panels (a) and (b) of Figure 2, Ours achieves higher throughput and lower wall-clock time per request across all concurrency levels. At concurrency 16, Ours reaches its peak throughput of 18.56 req/s, compared with 6.51 req/s for PoT-style. At the same concurrency level, it reduces wall-clock time per request to 0.0539s, while PoT-style requires 0.1535s. When concurrency increases to 32, its throughput slightly drops to 17.38 req/s and wall-clock time per request increases to 0.0575s, suggesting that excessive concurrency begins to introduce queueing or resource contention.

The parallel advantage comes from replacing repeated target-LLM program generation with the cache-hit path. After a reusable program cache is available, Ours only needs to extract variables and execute the cached program for later requests in the same group. Panel (c) illustrates this online cache warm-up process with a representative run at concurrency 8. The number of cache entries grows quickly during approximately the first 1K processed requests and reaches 25 entries around 1.1K requests, after which it stabilizes. The cumulative hit rate increases accordingly, and the final hit rate of the full run reaches approximately 98.23%.

These results highlight an important trade-off in parallel serving. Moderate concurrency improves throughput and provides enough requests for cache generation, while excessive concurrency may send more requests to the miss path before useful caches are available, increasing target-LLM pressure. Thus, the parallel efficiency of Ours depends not only on the execution cost of the cache-hit path, but also on the match between cache generation speed and request arrival rate.

A.9 Length Robustness Details

As the average context length increases from Avg-1K to Avg-8K, the latency of PoT-style increases from 2.577s to 5.737s, while the latency of Ours increases from 0.574s to 2.443s. Ours consistently outperforms PoT-style across all length settings, with speedups of 4.49×, 4.11×, 3.77×, and 2.35×, respectively. The reduced speedup at Avg-8K suggests that very long contexts increase the overall cost of variable extraction, cache-hit execution, and miss/fallback paths, but the remaining 2.35× speedup still shows that program caching is effective for long-context formula computation.

Ours also preserves stable quality under long contexts. From Avg-1K to Avg-4K, its accuracy, cache hit rate, and cache-hit accuracy remain around or above 94%. At Avg-8K, accuracy decreases to 93.01%, cache hit rate decreases to 90.85%, and cache-hit accuracy remains 93.34%. This indicates that extremely long contexts make cache reuse more difficult, but they do not cause a collapse in either answer quality or cache-hit correctness. The combination of high end-to-end accuracy and high cache hit rate further indicates that cache-hit requests are also reliable, so cached programs are both frequently used and correct when applied. These results suggest that when tasks have stable formula-level computation structures, reusable program caches can still provide substantial latency reduction under long-context inputs.