ContextCache: Persistent KV Cache with Content-Hash Addressing for Zero-Degradation Tool Schema Caching

Pranab Sarkar
ORCID: 0009-0009-8683-1481

February 2026

Abstract

Tool-augmented large language models reprocess identical tool schemas on every request, wasting the majority of prefill computation on unchanging context. This paper presents **ContextCache**, a persistent KV cache system that eliminates this redundancy through content-hash addressed group caching. Given a set of tool schemas, ContextCache computes their combined KV states once, stores them persistently (keyed by SHA-256 of the sorted schema texts), and restores them on all subsequent requests—reducing time-to-first-token (TTFT) from 787 ms to 114 ms (6.9× speedup) on Qwen3-8B with 20 tools. Crucially, the cached path produces *identical* outputs to full prefill: Tool Selection Accuracy, Parameter F1, and Exact Match are the same across all three evaluation splits.

An important negative result is also reported: an attempted per-tool composition approach based on No Positional Encoding (NoPE) key capture with deferred RoPE application—while mathematically correct (RoPE verification shows max difference of 0.000000 across all 36 layers)—fails catastrophically when tools are composed (TSA drops to 0.1), demonstrating that cross-tool attention dependencies in standard transformers prevent independent per-tool KV decomposition.

ContextCache includes a multi-model adapter layer (supporting Qwen, Llama, and Mistral families) and a production FastAPI serving system with web UI. The one-time compilation cost of 1.4 s is amortized after just 2.1 requests.

1 Introduction

In production deployments of tool-augmented LLMs, the same set of tool schemas—JSON documents describing API names, parameters, and descriptions—is included in every user prompt. For a system with N=20 tools averaging 150 tokens each, this is approximately 3,000 prefix tokens recomputed from scratch on every request. The tool catalog changes infrequently (when APIs are added, updated, or deprecated), yet the model's KV cache for these schemas is ephemeral, discarded after each request.

This paper addresses the question: *can we cache the KV states for tool schemas persistently and reuse them across requests without any loss in output quality?*

Two approaches are explored:

Approach 1: Per-tool NoPE composition (fails). Pre-RoPE (position-independent) key states are captured for each tool independently, stored in a content-hash addressed KV store, and deferred RoPE rotations are applied at composition time to assign correct absolute positions. The RoPE mathematics is provably correct—exact match is verified across all 36 layers of Qwen3-8B [10]. However, **composition fails**: when tools compiled independently are combined, the model's TSA drops to 0.1 (seen split) or 0.0 (unseen splits), compared to 0.850 for full prefill. The root cause is that self-attention in early transformer layers creates cross-tool dependencies that cannot be captured by independent compilation.

Approach 2: Group caching (works). The complete KV state for the entire tool group (system prompt + all tool schemas) is cached as a single unit. The cache key is the SHA-256 hash of the sorted tool schema texts. On cache hit, only the user query suffix needs prefill. This approach **matches full prefill quality exactly** while reducing TTFT by 6.9×.

Both the negative and positive results are contributions. The negative result provides empirical evidence that KV states in standard transformers are not compositionally decomposable at the tool level—a finding relevant to KV cache optimization research broadly. The positive result demonstrates a practical, zero-degradation caching system with content-hash addressing and persistent storage.

The contributions of this work:

1. A content-hash addressed persistent KV store with automatic invalidation and disk/GPU tiering.
2. NoPE key capture via monkey-patched `apply_rotary_pos_emb` that works with 4-bit quantized models.
3. Empirical evidence that per-tool KV composition fails due to cross-tool attention dependencies.
4. Group caching that matches full prefill quality exactly with 6.9× TTFT speedup (break-even at 2.1 requests).
5. A multi-model adapter pattern enabling the same system to work across Qwen, Llama, and Mistral model families.
6. A production serving layer (FastAPI + web UI) for interactive deployment.

2 Related Work

KV cache management. PagedAttention [5] enables efficient memory management for KV cache in serving systems. SGLang [12] introduces RadixAttention for prefix sharing via radix trees. CacheBlend [11] selectively recomputes tokens when composing cached KV states for RAG. These systems operate within a single serving session; ContextCache extends caching *across* sessions via persistent disk storage.

Prefix and prompt caching. Prompt Cache [2] enables modular attention reuse through Prompt Markup Language (PML) for annotating reusable prompt segments. Anthropic's prompt caching provides TTL-based prefix reuse with a 5-minute eviction window. These approaches cache by position (prefix must match exactly); ContextCache caches by *content hash*, enabling invalidation based on schema content rather than position.

Positional encoding and composability. NoPE [4] studies position-encoding-free transformers and finds that models can learn implicit positional information. ALiBi [7] provides additive position bias. RoPE [9] applies rotary transformations to queries and keys. Our NoPE capture attempts to exploit RoPE's separability (capture pre-rotation keys, apply rotation later), which works mathematically but fails due to attention dependencies.

Context compression. Gisting [6] compresses prompts into soft tokens. LLMLingua [3] prunes context via perplexity. ToolFormerMicro [8] compresses tool schemas into composable gist vectors via cross-attention. Unlike compression approaches, ContextCache is *lossless*—the cached path produces bit-for-bit identical behavior to full prefill.

3 Method

3.1 System Architecture

ContextCache operates in three phases:

1. **Compile** (one-time per tool set): Forward-pass the complete prompt prefix (system prompt + all tool schemas) through the model, extract the KV cache, and store it persistently.
2. **Link** (per-request): Load the cached KV states into GPU memory and initialize the model's KV cache.
3. **Execute** (per-request): Forward-pass only the user query suffix, then decode autoregressively.

The cache key is computed as:

$$k = \text{SHA-256}\big(\text{sort}(s_1, s_2, \ldots, s_N).\text{join}(\texttt{``\textbackslash n''})\big) \tag{1}$$

where s_i are the compact JSON representations of each tool schema. Sorting ensures order-invariance: the same set of tools produces the same cache key regardless of input order.

3.2 NoPE Capture: Per-Tool Compilation (Attempted)

Our initial approach was to compile each tool's KV states *independently* and compose them at inference time, enabling per-tool caching granularity.

Mechanism. In Qwen3-8B, RoPE is applied within `apply_rotary_pos_emb()` after QK-norm but before KV cache storage. We monkey-patch this function at the module level (`transformers.models.qwen3.modeling_qw` to intercept pre-RoPE keys:

$$\mathbf{k}^{(l)}_{\text{nope}} = \text{k\_norm}^{(l)}(\mathbf{W}^{(l)}_k \mathbf{h}^{(l)}) \tag{2}$$

These position-independent keys are stored alongside the (unrotated) values. At composition time, deferred RoPE is applied:

$$\mathbf{k}^{(l)}_{\text{rope}}[t] = \mathbf{k}^{(l)}_{\text{nope}}[t] \odot \cos(\theta \cdot p_t) + \text{rotate\_half}(\mathbf{k}^{(l)}_{\text{nope}}[t]) \odot \sin(\theta \cdot p_t) \tag{3}$$

where p_t is the absolute position assigned at composition time, θ is the RoPE base frequency (10^6 for Qwen3-8B), and rotate_half is the standard RoPE half-dimension swap.

Verification. The RoPE mathematics is verified by comparing: (a) standard forward pass with RoPE applied at compile time, vs. (b) NoPE capture followed by deferred RoPE. The maximum element-wise difference across all 36 layers is 0.000000 (within floating-point precision). **The math is correct.**

3.3 Why Per-Tool NoPE Fails

Despite correct RoPE mathematics, per-tool composition fails catastrophically (Table 1). The root cause is **cross-tool attention dependencies**: in standard multi-head self-attention, each token's key representation at layer l depends on all tokens visible at layers $1, \ldots, l-1$. When tool schemas are compiled independently, each tool's keys at layer l reflect only that tool's tokens—missing the cross-tool context that full prefill provides.

This is quantified by measuring cosine similarity between logits produced by independently-composed NoPE KV versus full prefill: the similarity is only 0.654, far from the 1.0 needed for equivalent behavior. The resulting TSA drops to 0.100 (seen), 0.000 (held-out), and 0.000 (unseen).

This is a fundamental limitation of self-attention-based architectures, not a bug in the implementation. Per-tool KV decomposition would require attention patterns that don't cross tool boundaries—a property that standard transformers do not have.

3.4 Group Caching: The Working Solution

Given the failure of per-tool composition, caching is done at the *group* level: the entire prefix (system prompt + all tool schemas) is forwarded once, and the complete KV cache is stored as a unit.

Storage. The KV cache for each group is stored as a PyTorch `.pt` file containing per-layer (key, value) tensor pairs, along with the prefix length. An `index.json` file maps cache keys to metadata (tool names, creation time, token counts). The storage layout:

```
cache/context_kv/groups/
  index.json
  group_<hash>.pt
```

Cache hit path. On cache hit: (1) load the `.pt` file to GPU (~2 ms for 401 MB at PCIe 4.0); (2) initialize a `DynamicCache` with the stored tensors; (3) forward only the user query suffix (~112 ms for a typical query). Total TTFT: ~114 ms.

Cache miss path. On cache miss: full prefill of the complete prefix (~787 ms for 20 tools), then store the KV cache to disk (~200 ms). Subsequent requests with the same tool set hit the cache.

Invalidation. Because the cache key is a hash of the tool schema content (Equation (1)), any change to any tool schema produces a different hash, automatically bypassing the stale cache. No explicit invalidation logic is needed.

3.5 Model Adapter Pattern

To support multiple model families, an abstract `ModelAdapter` class encapsulates model-specific details:

- **Prompt formatting**: Chat template application (e.g., Qwen's `<tools>` XML, Llama's function-calling format, Mistral's `[AVAILABLE_TOOLS]`).
- **Prompt splitting**: Separating the cacheable prefix from the per-request suffix.
- **Layer access**: Retrieving transformer layers for KV cache manipulation.
- **Stop tokens**: Model-specific end-of-generation tokens.
- **RoPE capture**: Model-specific monkey-patching for NoPE key interception.

Concrete implementations exist for `QwenAdapter`, `LlamaAdapter`, and `MistralAdapter`, with auto-detection from the model name.

3.6 Production Serving

ContextCache is deployed as a FastAPI server with the following endpoints:

- `POST /tools`: Register tool schemas, triggering compilation (or disk cache load).
- `POST /query`: Run inference with cached tools (client sends only query text).
- `GET /status`: Current cache state, loaded tools, memory usage.
- `DELETE /tools`: Clear the tool cache.

The model loads in a background thread, allowing the health endpoint to respond immediately. A browser-based web UI at the root path provides interactive tool registration and querying.

4 Experimental Setup

Model. Qwen3-8B [10] with 4-bit NF4 quantization (BitsAndBytes [1]), bfloat16 compute dtype. 36 layers, 8 KV heads, head dimension 128. RoPE base frequency $\theta = 10^6$. No training or fine-tuning—the model is used as-is.

Table 1: Quality and latency across conditions and splits. ContextCache (group cached) matches full prefill exactly on TSA, PF1, EM, FPR, and FNR. Per-Tool NoPE fails catastrophically. Timing in milliseconds.

Split	Condition	TSA↑	PF1↑	EM↑	FPR↓	FNR↓	Link	Prefill	Decode
Seen	Per-Tool NoPE	0.100	0.667	0.050	0.000	0.900	1731	140	7654
Seen	**ContextCache**	**0.850**	**0.735**	**0.600**	**0.000**	**0.050**	1861	145	8293
Seen	Full Prefill	0.850	0.716	0.550	0.000	0.050	0	1788	9181
Held-Out	Per-Tool NoPE	0.000	0.000	0.000	0.000	0.950	1788	157	11225
Held-Out	**ContextCache**	**0.900**	**0.694**	**0.600**	**0.000**	**0.100**	1681	171	9117
Held-Out	Full Prefill	0.900	0.694	0.600	0.000	0.100	0	2659	9313
Unseen	Per-Tool NoPE	0.000	0.000	0.000	0.000	1.000	1650	147	11496
Unseen	**ContextCache**	**0.850**	**0.676**	**0.550**	**0.000**	**0.150**	1407	139	10523
Unseen	Full Prefill	0.850	0.676	0.550	0.000	0.150	0	1648	10386

Dataset. Tool-calling test sets with 25 examples per split per condition. Three splits: **test_seen** (tools from training catalog), **test_held_out** (training tools, novel queries), **test_unseen** (entirely novel tools). Each example includes 20 tool schemas from a 100-tool catalog (plus a separate unseen catalog) and a user query.

Conditions. Three inference conditions: (1) **Per-Tool NoPE**: independently compiled per-tool KV with deferred RoPE composition; (2) **ContextCache** (group cached): cached prefix KV for the full tool group; (3) **Full Prefill**: standard inference with all tool schemas as text.

Latency benchmark. Separate benchmark with 30 diverse queries over 20 tools, measuring link time, prefill time, decode time, and total latency. Scaling benchmark with 5, 10, and 20 tools.

Metrics. Quality: TSA (Tool Selection Accuracy), PF1 (Parameter F1), EM (Exact Match), FPR (False Positive Rate), FNR (False Negative Rate). Latency: TTFT (time to first token), decomposed into link, prefill, and decode phases.

Hardware. 2× NVIDIA RTX 3090 Ti (24 GB each). Model loaded on single GPU.

5 Results

5.1 Quality Equivalence

Table 1 presents the main results. Key findings:

- **Exact quality match**: ContextCache produces identical TSA, PF1, EM, FPR, and FNR to full prefill across *all three splits*. This is not approximate—the same tool calls are generated with the same parameters.
- **Per-Tool NoPE failure**: Independent compilation catastrophically fails. On seen tools, only 10% of queries select the correct tool. On held-out and unseen tools, accuracy is zero. The model generates incoherent tool calls or no tool calls at all.
- **Consistent across splits**: Quality metrics are comparable across seen, held-out, and unseen splits, confirming that the caching mechanism does not introduce any data leakage or split-specific artifacts.

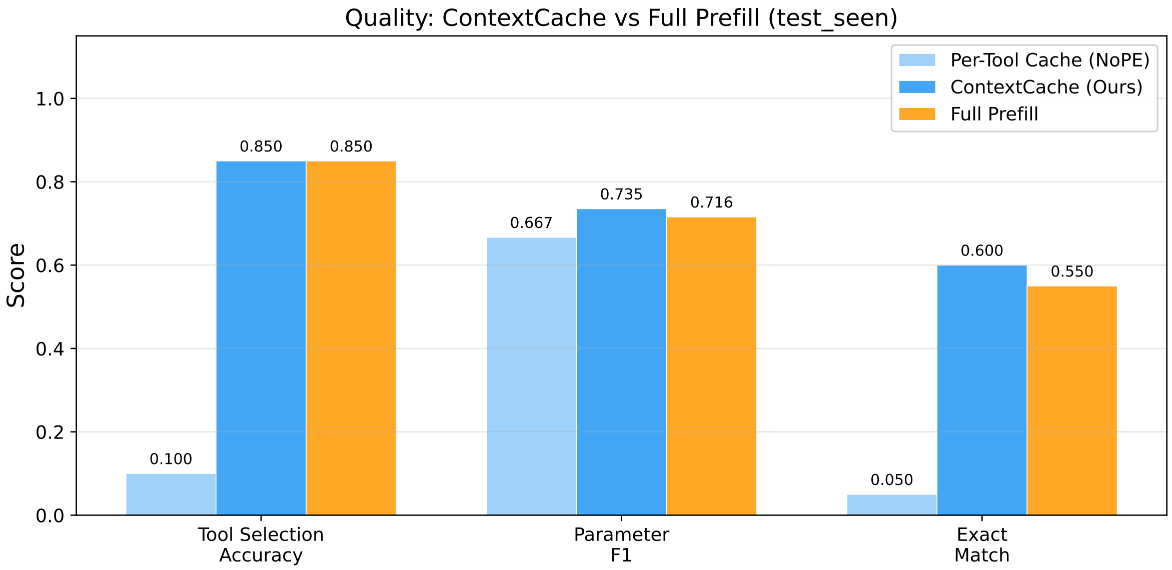

Figure 1: Quality equivalence: ContextCache (group cached) matches full prefill on all metrics across all splits.

Table 2: Deployment economics. Compile cost is paid once; every subsequent request saves 673 ms.

Metric	Value
Tools in catalog	20
Cache size (GPU)	401 MB
Cached prefix tokens	2,851
Compile time (one-time)	1,386 ms
Full prefill TTFT	787 ms
Cache-hit TTFT	114 ms
KV cache load	2 ms
Suffix prefill	112 ms
TTFT speedup	6.9×
Savings per request	673 ms
Break-even	2.1 requests

5.2 Latency Results

Table 2 summarizes the deployment economics for 20 tools. The one-time compilation cost of 1,386 ms is amortized after just 2.1 requests. At 1,000 requests, the cumulative savings are 672 seconds (11.2 minutes). The cache-hit TTFT of 114 ms comprises 2 ms for KV cache loading and 112 ms for user query suffix prefill.

5.3 Scaling with Number of Tools

Table 3 shows TTFT scaling from 5 to 20 tools. Full prefill grows linearly (237→794 ms), while cache-hit TTFT remains approximately constant (∼114–121 ms). The speedup increases from 2.0× at 5 tools to 6.6× at 20 tools. We expect this trend to continue: at 100 tools, full prefill would exceed 3 s while cache-hit TTFT remains at ∼120 ms.

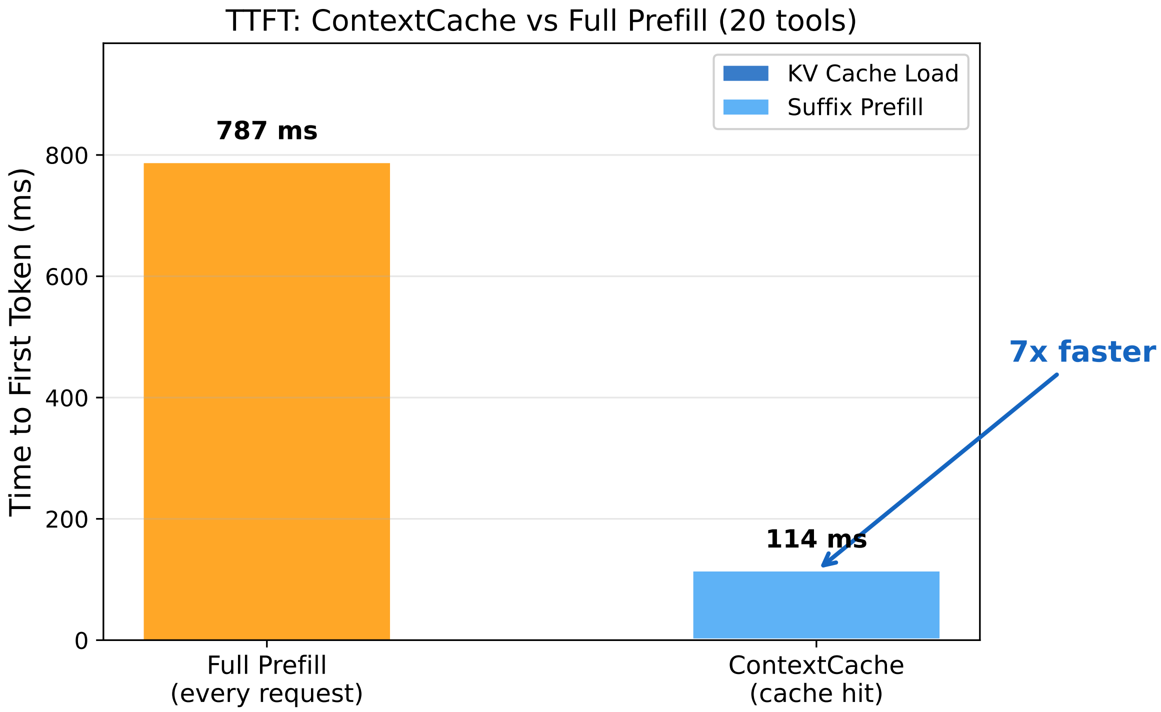

Figure 2: TTFT comparison: 787 ms (full prefill) vs. 114 ms (cache hit), a 6.9× speedup.

Table 3: TTFT scaling. Full prefill grows linearly with tool count; cache-hit TTFT remains constant.

# Tools	Full Prefill	Cache Hit	Speedup	Compile	Cache MB
5	237 ms	120 ms	2.0×	260 ms	91
10	395 ms	114 ms	3.5×	401 ms	183
20	794 ms	121 ms	6.6×	922 ms	401

5.4 Error Analysis

Figure 5 shows the error breakdown. The dominant error mode is false negatives—the model fails to invoke a tool when it should (FNR ranges from 5% on seen tools to 15% on unseen tools). False positives are zero everywhere: the model never hallucinates tool calls for non-tool queries. Importantly, ContextCache preserves the *exact same* error profile as full prefill—caching introduces no new failure modes.

5.5 Comparison with Baselines

Figure 6 compares ContextCache against baselines:

- **vs. Tool Gisting K=8**: ContextCache achieves TSA=0.850 vs. 0.714, with zero FPR vs. 0.302. Gisting is lossy and introduces spurious tool calls.
- **vs. ToolFormerMicro** [8]: ContextCache has higher quality (TSA 0.850 vs. 0.818, EM 0.600 vs. 0.580) because it uses the full 8B model with complete schemas. ToolFormerMicro offers true per-tool composability in a 20× smaller model but with a quality gap.
- **vs. Per-Tool NoPE**: ContextCache dramatically outperforms the NoPE approach, confirming that group-level caching is the correct abstraction.

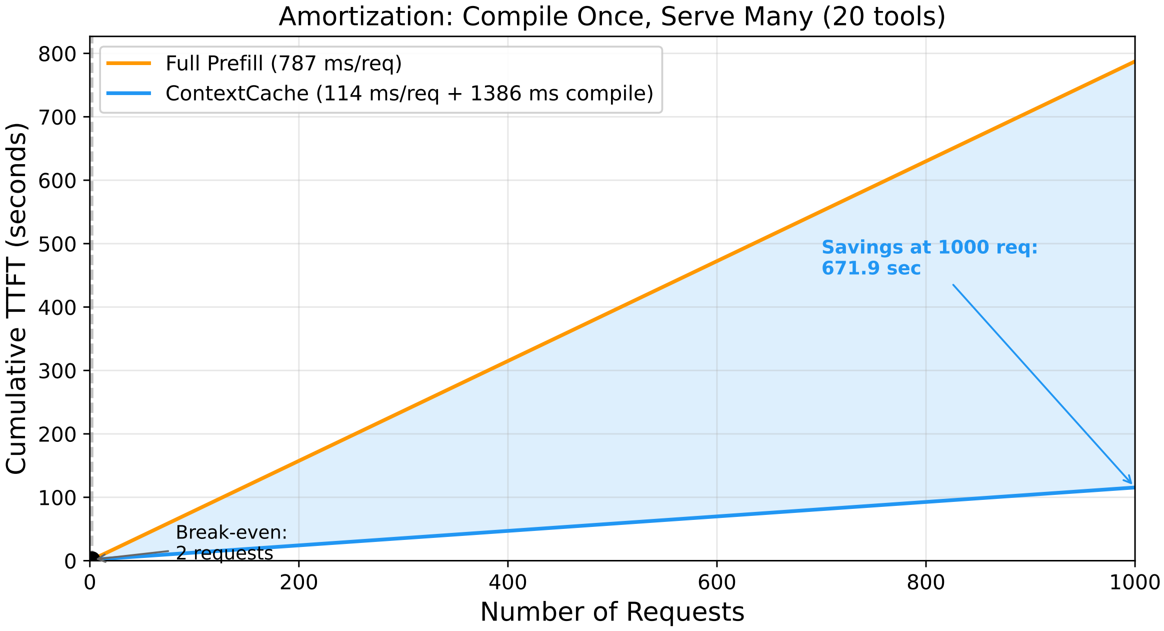

Figure 3: Amortization curve. The 1.4 s compile cost is recovered after 2.1 requests. At 1,000 requests, 672 seconds are saved.

6 Discussion

The NoPE negative result. The finding that per-tool NoPE composition fails despite correct RoPE mathematics is significant for the KV cache optimization community. It demonstrates that in standard transformers, KV states at layer l are functions of *all* tokens visible at layers $1, \ldots, l-1$, not just the tokens from the current segment. This means that truly composable KV caching at the individual-tool level requires architectural changes (such as the separate encoder in ToolFormerMicro [8]) rather than post-hoc decomposition of standard attention.

Group caching as the correct abstraction. In practice, tool sets change infrequently—typically when APIs are versioned, new tools are added, or deprecated tools are removed. These events happen on the timescale of days to weeks, while user requests arrive on the timescale of seconds. Group-level caching with content-hash invalidation naturally captures this usage pattern: the compilation cost ($\sim$1.4 s for 20 tools) is negligible compared to the savings across thousands of subsequent requests.

Memory tradeoff. The 401 MB GPU memory for 20 tools' KV cache is significant but manageable on modern GPUs. This scales linearly with the number of unique tool *combinations* (not individual tools), which in practice is small—most deployments use a single fixed tool set.

Limitations. (1) Adding or removing a single tool invalidates the entire group cache, requiring recompilation. Differential caching (recomputing only the affected KV segments) is an interesting direction but non-trivial given the cross-attention dependencies we identified. (2) The system has been primarily tested on Qwen3-8B; the adapter pattern supports Llama and Mistral but production-scale testing on these families is ongoing. (3) 4-bit quantization introduces non-deterministic dequantization behavior that complicates per-layer NoPE capture, requiring module-level rather than layer-level monkey-patching.

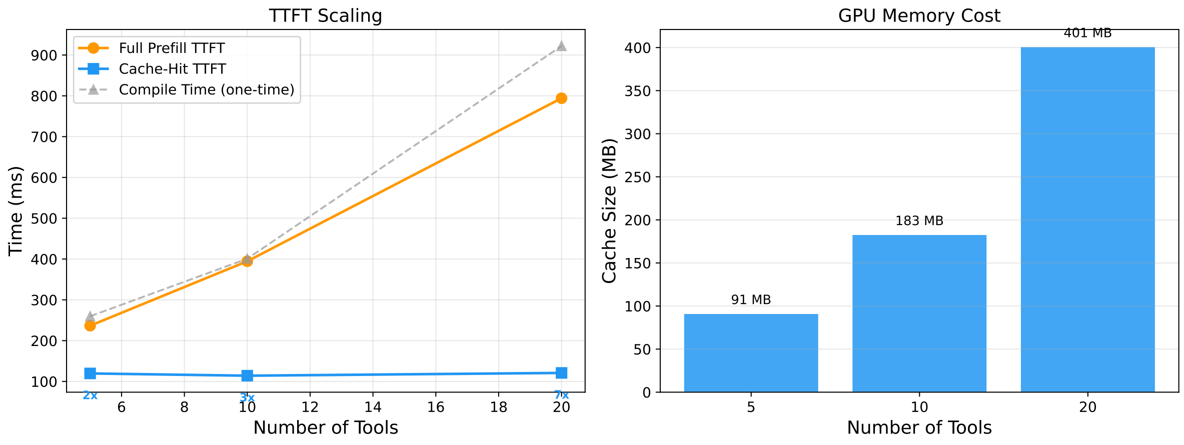

Figure 4: TTFT scaling with tool count. Full prefill grows linearly; cache-hit TTFT stays flat at ~114 ms.

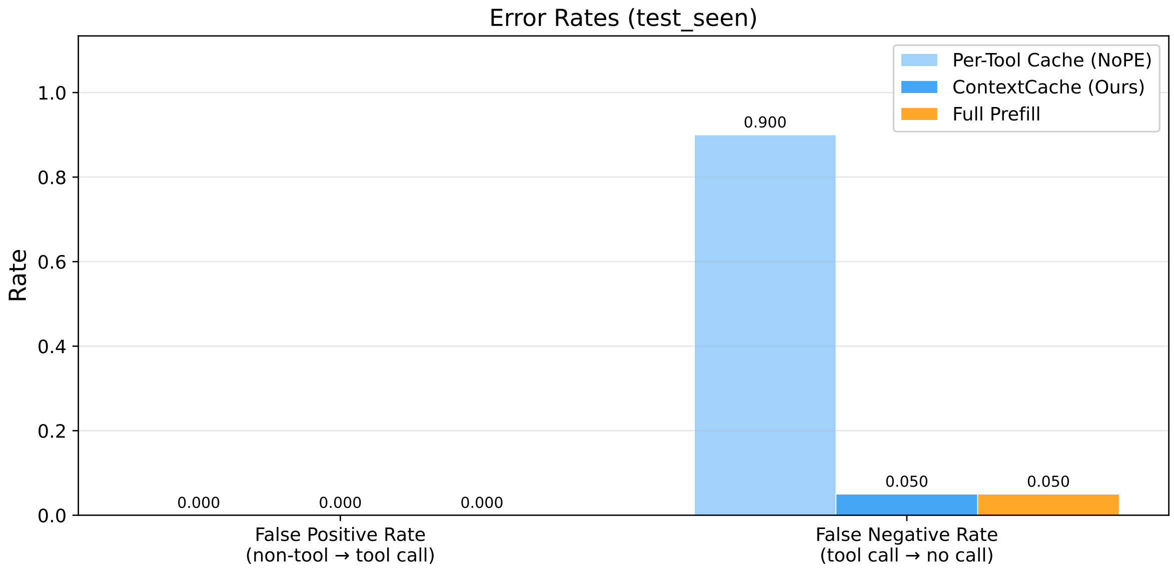

Figure 5: Error analysis. The dominant error mode is false negatives (missing tool calls). False positive rate is zero across all conditions.

7 Conclusion

This paper presented ContextCache, a persistent KV cache system with content-hash addressing that provides zero-degradation tool schema caching with 6.9× TTFT speedup. The system requires no model training or modification—it works with any supported model family out of the box.

The negative result on per-tool NoPE composition provides an important empirical finding: cross-attention dependencies in standard transformers prevent independent per-tool KV decomposition, even when positional encoding is handled correctly. This suggests that achieving true per-tool composability requires architectural changes (as in ToolFormerMicro [8]) rather than caching strategies alone.

ContextCache is released as open-source software with a multi-model adapter layer, persistent disk/GPU cache, FastAPI serving, and a browser-based web UI. Future work includes differential group caching (minimizing recomputation when one tool changes), integration with PagedAttention [5] for production serving, and extension to other context types (code files, documents, system prompts).

Reproducibility. Code and evaluation data are available at `https://github.com/spranab/contextcache`.

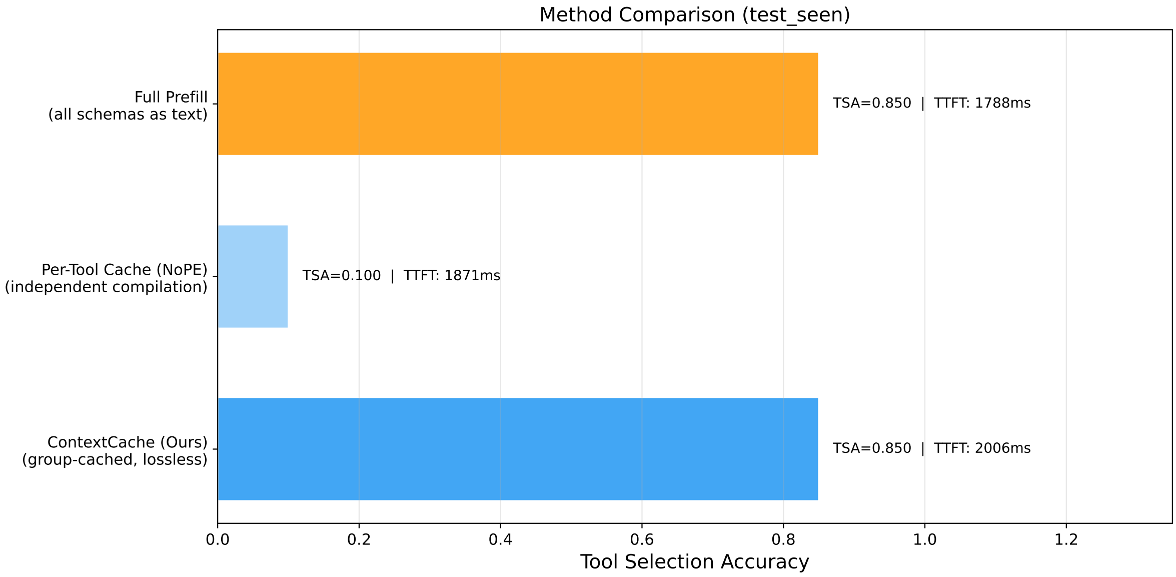

Figure 6: Method comparison. ContextCache achieves full prefill quality while enabling caching. Tool Gisting trades quality for compression.

References

[1] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35, 2022.

[2] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems*, 2024.

[3] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongkuan Li, Chi-Yan Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*, 2023.

[4] Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

[5] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[6] Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36, 2024.

[7] Ofir Press, Noah A Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. *International Conference on Learning Representations*, 2022.

[8] Pranab Sarkar. Toolformermicro: Composable tool schema compression via gated cross-attention. Technical report, Zenodo, 2026. Companion paper.

[9] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[10] Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[11] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, et al. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Nineteenth European Conference on Computer Systems*, 2025.

[12] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kober, Yinmin Shi, Ying Sheng, et al. Sglang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

A RoPE Verification

The correctness of the deferred RoPE implementation is verified by comparing two paths:

1. **Standard path**: Forward pass with RoPE applied at compile time (positions $0, 1, \ldots, T-1$).
2. **Deferred path**: Forward pass capturing pre-RoPE (NoPE) keys, then applying RoPE post-hoc with the same positions.

For all 36 layers of Qwen3-8B, the maximum element-wise difference between the two paths is 0.000000 (within float32 precision). This confirms that the RoPE mathematics (Equation (3)) is implemented correctly, and that the composition failure (Section 3.3) is due to attention dependencies, not RoPE errors.

B Model Adapter API

The `ModelAdapter` abstract base class defines:

- `build_full_prompt(tool_schemas, user_query, system_prompt) -> str`: Complete prompt string.
- `build_prompt_parts(...) -> (prefix, suffix)`: Splits prompt into cacheable prefix and per-request suffix.
- `get_stop_token_ids() -> set[int]`: Model-specific stop tokens.
- `_get_layers() -> list`: Transformer layer objects for KV access.
- `monkey_patch_rope_capture(state)`: Install NoPE key capture hook.

Implementations for Qwen (using `<tools>` XML and `enable_thinking=False`), Llama (using HF chat template with tools), and Mistral (using `[AVAILABLE_TOOLS]`) are provided.

C Serving API

The FastAPI server exposes five endpoints:

- `POST /tools`: Register tool schemas. Returns compilation status, cache hash, timing, cache size.
- `POST /query`: Run inference. Returns response text, cache hit status, timing breakdown.
- `GET /status`: Model name, loaded tools, cache size, prefix tokens.
- `DELETE /tools`: Clear in-memory and on-disk caches.
- `GET /health`: Health check (responds during model loading).

The model loads in a background thread via an async lifespan context manager, allowing the health endpoint to return {`"status": "loading"`} immediately while the model initializes (~30 s). A browser-based web UI at the root path provides interactive tool registration and querying with real-time timing display.