# Serve Programs, Not Prompts

In Gim
in.gim@yale.edu
Yale University
New Haven, CT, USA

Lin Zhong
lin.zhong@yale.edu
Yale University
New Haven, CT, USA

## Abstract

Current large language model (LLM) serving systems, primarily designed for text completion, are neither efficient nor adaptable for increasingly complex LLM applications due to their inflexible design. We propose a new LLM serving system architecture that serves *programs* instead of prompts to address this problem. These programs, called *LLM Inference Programs (LIPs)*, allow users to customize token prediction and KV cache management at runtime and to offload parts of their application logic, such as tool execution, to the server. We describe an example of this architecture through a system named Symphony, which functions as *an operating system* for LIPs. Symphony exposes LLM model computations via system calls and virtualizes KV cache with a dedicated file system, while ensuring GPU efficiency with a two-level process scheduling scheme. Symphony has the potential to open the door to a more efficient and extensible ecosystem for LLM applications.

## CCS Concepts

• **Computing methodologies** → **Natural language processing**.

## Keywords

Large language models, LLM serving systems, KV cache

**ACM Reference Format:**
In Gim and Lin Zhong. 2025. Serve Programs, Not Prompts. In *Workshop on Hot Topics in Operating Systems (HOTOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730398

## 1 Introduction

Current large language model (LLM) serving systems are primarily designed for high-throughput text completion. They accept *prompts* as input and stream generated text as output. This prompt-serving paradigm has become the de facto standard, commonly available both as cloud APIs and as open-source software [22, 30, 41]. Unfortunately, this paradigm faces growing efficiency and adaptability challenges as LLM applications evolve into complex, compound AI systems [50]. For instance, the stateless nature of this paradigm makes multi-round, programmatic interactions with LLMs [31, 48, 49] inefficient due to redundant recomputations. In addition, emerging techniques that deviate from standard token generation process [21, 27, 32] or require tool or data-augmented generation workflows [23, 44, 45] can be difficult to implement within these systems, forcing developers to modify the core system [1, 15] or create inefficient client-side workarounds (See §2).

This paper motivates a new LLM serving architecture that shifts the fundamental *unit of service* from prompts to *programs*. In this model, users flexibly define and offload their own LLM token generation routines to the LLM serving system, utilizing fine-grained APIs provided by the system. That is, instead of a prompt, a user sends a program to the serving system to control the generation process. We term these user-defined routines *LLM Inference Programs (LIPs)*. We identify three core properties for APIs to program LIPs: (1) decoupling generation from model computation, (2) application-managed model states (e.g., KV cache), and (3) co-locating external interactions. Even fundamental processes like the standard autoregressive generation loop should be explicitly definable using these APIs when needed (See §3).

To exemplify this idea, we present a new LLM serving system named *Symphony* (§4). Symphony directly leverages existing operating system abstractions to design the APIs needed to program LIPs, and serve them efficiently. For example, Symphony uses a file system to virtualize the KV cache and a system call for model computation. Users can employ host OS APIs (e.g., POSIX) for custom inference strategies, such as parallel generation or integrating external tools.

The proposed program-serving paradigm enables developers to incorporate application-specific optimizations directly at the LLM generation level and implement new LLM techniques without altering the core LLM serving system. Our initial experiments indicate that Symphony can deliver substantial performance improvements, such as achieving up to 7 times greater throughput compared to existing systems like vLLM [30], by empowering users to implement custom KV cache replacement policies through LIPs (See §5).


*HOTOS '25, Banff, AB, Canada*

ACM ISBN 979-8-4007-1475-7/2025/05
https://doi.org/10.1145/3713082.3730398

We discuss the main challenges and opportunities of this new approach in §6, regarding the granularity of the APIs, security implications, performance overhead, and the need for new benchmarks.

## 2 Motivation

Prompt-centric serving systems struggle with efficiency, control, and adaptability when handling complex LLM workflows beyond basic text completion. To address these challenges, developers frequently resort to ad-hoc system modifications or inefficient client-side solutions. Consider the example of developing an LLM-based code editor which provides live code autocompletions.

- **Efficiency:** As the user types, each keystroke ideally triggers an update. A naive prompt-based system recomputes the entire prompt repeatedly. Even with server-side prompt caching [17, 30], the policy is server-defined and not application aware. For example, the serving system might cache prompts that will not be used any longer.
- **Interaction:** Suppose the editor implements Retrieval-Augmented Generation (RAG) [24] to fetch relevant API documentation via function calling [37]. This means the client application acts as an intermediary: Prompt → Serving System → Function Call Spec → Client → Execute Function (e.g., fetching API documentation) → Client → Function Result → Serving System. Each arrow involves boundary crossing overhead.
- **Control:** Enforcing code conventions or ensuring generated code fits a specific structure (constrained decoding [7]) requires manipulating the token generation process. Prompt-based APIs offer limited control (e.g., temperature, basic JSON mode enforcement), insufficient for complex grammars or custom sampling strategies.

We elaborate on these underlying problems below.

### 2.1 Resource Inefficiency

Existing systems suffer from resource inefficiency due to the lack of application-driven management over the KV cache. The KV cache is a crucial component in the efficient serving of "GPT-style" LLMs [39, 40]. It enables incremental LLM model computations by avoiding the need to recompute all input tokens. This is possible because the internal representations of a token (K and V states in self-attention) depend solely on preceding tokens in causal Transformers, allowing them to be reused for subsequent LLM model computations when the preceding token sequences remain unchanged. Optimizing the management and reuse of the KV cache is one of the most critical areas for enhancing LLM serving performance [17, 30, 35, 52].

However, current systems are designed around prompts and lack awareness of application-specific reuse patterns, which limits optimization potential. Typically, KV cache management is governed by a system-wide policy (e.g., LRU eviction) that applies to all requests. For instance, in scenarios involving multi-round prompting, maintaining the KV cache from prior interactions can significantly decrease latency [15]. However, users lack the ability to manage the KV cache retention, even when they possess knowledge of reuse patterns. Current methods, like automatic prefix caching in vLLM, fail to provide this level of flexibility.

Several LLM providers, including Anthropic, offer APIs for prompt caching [2], enabling users to determine what should be cached prior to generation. Systems like SGLang and PromptCache [17] further allow users to define the structure of input prompts [52], enhancing the efficient reuse of KV caches in parallel generation strategies such as Tree-of-Thought [48]. However, while these systems are beneficial in various contexts, they still handle KV cache management implicitly and lack the capability to accommodate application-specific reuse patterns beyond their intended design, such as using graph [5] or recursive [31] generation strategies.

**Our solution.** We propose making KV cache management an explicit, application-defined operation—shifting optimization responsibility from the serving system to user programs.

### 2.2 Communication Overheads

Current serving systems are only responsible for token generation, forcing any additional non-token generation logic, such as function calling, to be implemented on the client side. This introduces communication overheads. LLM function calling [37, 38, 49] is essential for interfacing an LLM with external APIs and data sources, which is crucial for many LLM applications [44]. The typical workflow for LLM function calling involves: (1) the user providing a description of functions and their parameters as part of the prompt, (2) the LLM generating a function call in response, and (3) the user parsing and executing the function call, followed by a request to return the execution result to the LLM.

In this process, the user effectively acts as a code interpreter, necessitating network round trips between the user and the LLM serving system. These round trips can significantly increase the end-to-end latency of an AI application, especially as the number of function calls increases [18, 25]. However, in many cases where function calls do not rely on the user's environment for execution (e.g., accessing third-party APIs like weather or stock price APIs, or executing simple code snippets such as NumPy calculations), these round trips can be avoided by enabling the LLM serving system to execute the function directly. Moreover, multi-agent LLM applications [46] implement inter-agent communication through LLM function calling. However, this agent-to-agent communication incurs high costs because

users must handle the communication logic. We can reduce this overhead by enabling the serving system to manage the communication autonomously.

**Our solution.** We propose that the serving system should incorporate a code execution environment to handle LLM function calls and code executions internally, rather than relying on the user to manage all execution and communication.

## 2.3 Uncontrollable Generation

Current LLM serving systems integrate the autoregressive token generation loop into their core, continuously sampling the next token until they encounter an end-of-sequence (`EOS`) token. As a result, users have limited control over token generation, typically restricted to adjusting a few parameters in the next-token sampler and the temperature. This limitation complicates the implementation of emerging techniques for more robust and sophisticated LLM use, such as constrained generation, which ensures the LLM output adheres to a specific format or grammar [7, 16], and policy-based generation [20, 26, 29] for improved output quality. These stateful sampling strategies often necessitate intrusive modifications to the LLM serving system and expose specialized APIs. While some serving systems support popular methods like constrained decoding through JSON, Regex, and Context-Free Grammar [6, 12, 13], these methods do not extend to arbitrary sampling strategies.

We note that exposing the sampling process as an end-user facing API is impractical due to the large size of the next-token distribution. For example, GPT-4's vocabulary size exceeds 100K tokens, resulting in a distribution size of approximately 200 KB using FP16.

**Our solution.** We propose offloading arbitrary programs to the serving system, enabling applications to directly access and manipulate the token distribution during generation.

## 2.4 API Fragmentation

While ad-hoc solutions can address the limitations of current LLM serving system, they often struggle with adaptability as LLM workloads grow more varied and complex. For instance, an application focused on solving complex mathematical problems might leverage parallel reasoning techniques without prioritizing latency [43]. Conversely, a robotics application with numerous LLM function calls might prioritize reduced latency over absolute accuracy [9].

These diverse workloads present challenges in API design, as each specialized solution demands unique API requirements. This has led to a fragmentation of LLM APIs among major providers. For example, Google Cloud [11], OpenAI [36], and Anthropic [2] each offer distinct API designs and semantics for prompt caching, LLM function calling, and constrained decoding.

**Our solution.** We emphasize the need for composable, fine-grained APIs that can accommodate the programming of varied LLM workloads.

## 2.5 Related Work

Recent research aims to make LLM serving systems more application-aware, but it often addresses challenges in isolation without a unified approach. Systems like Prompt-Cache [17], SGLang [52], and Parrot [33] offer APIs for programmatically defining input prompt structures. These systems leverage this structure for efficient KV cache reuse. However, the KV cache management is implicit and cannot express arbitrary reuse patterns beyond the system's predefined abstractions.

For controlled generation, tools such as XGrammar [12], Outlines [13], and Guidance [19] allow users to enforce output constraints using custom rules or domain-specific languages (DSLs). These systems embed a fixed set of decoding strategies directly into the serving stack, limiting extensibility. Symphony generalizes this model by exposing the low-level token sampling loop as a programmable interface, enabling arbitrary control strategies that go beyond what built-in grammars or templates can support.

For LLM function calling, InferCept [1] optimizes the KV cache during such function calls. Some other works, such as LLMCompiler [25] or AsyncLM [18], make this interaction efficient by allowing multiple function calls to be run concurrently. Symphony makes such workflows native, treating function execution, caching, and token generation as composable building blocks within a single user-defined LIPs.

# 3 Program as the Unit of Service

In light of the challenges outlined above, we propose that LLM serving system should transition from processing mere prompts to handling *programs*, which we term LLM Inference Programs (LIPs). We specifically champion three core attributes for LIPs, aimed at empowering users to program their own optimization and generation strategies by utilizing fine-grained APIs for model computation and KV cache management.

- *Separation of Generation and Model Computation*: The logic for generating tokens, such as an autoregressive loop, should be decoupled from the model computation, like Transformer model operations on GPUs. The generation process should be defined within LIPs, while the LLM serving system only needs to focus on efficiently managing the model computations requests.

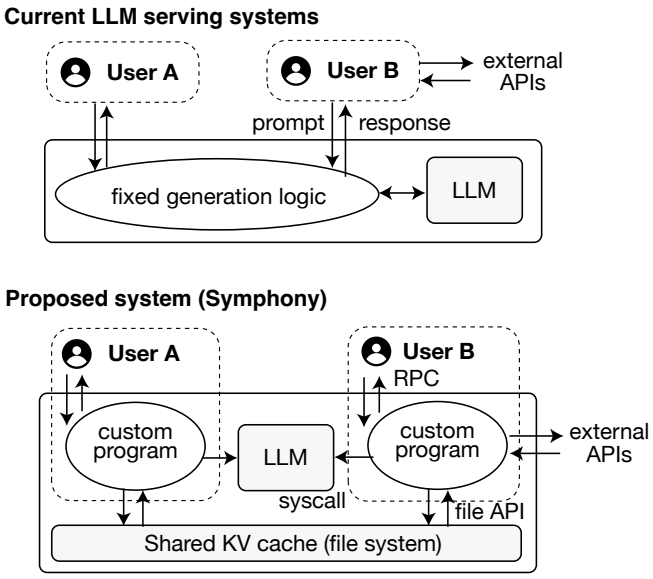


**Figure 1: Comparison with existing serving systems (top) and Symphony (bottom). Symphony serves as an operating system for user-defined inference programs.**

- *Application-Controlled Model States*: LLM states, including the KV cache, should be able to persist beyond a single LIP and be explicitly managed by the program. It is the user's responsibility, not the LLM serving system's, to efficiently utilize the KV cache for their tasks. The LLM serving system should offer an API that allows users to manage the KV cache, such as creating, updating, or deleting it.
- *Integrated External Interactions*: Each LIP should independently manage its external interactions, such as function calls and I/O, without depending on the LLM serving system or external client-side logic for coordination.

We provide an example code for LIP in Figure 2. We elaborate on the API design in the following section.

## 4 LLM Serving System as OS

We introduce a LLM serving system called *Symphony*, designed to serve LIPs. Symphony treats the execution of a LIP like a process in an operating system (OS), leveraging existing OS APIs, abstractions, and implementations. By doing so, we aim to extend the core functions of the LLM serving system beyond next token prediction, to include resource virtualization (e.g., KV cache), concurrent program execution, and I/O management—key responsibilities of an OS.

### 4.1 Model Computation via System Calls

Symphony provides a specialized system call for LLM model computation, named `pred`, which stands for next token prediction. Its signature is as follows:

```
// load precomputed kv cache
prefix_kv = kv_open("sys_msg.kv");
suffixes = {
    tokenize(/** query 1 */), ...
    tokenize(/** query n */)
};
for (suffix : suffixes) {
    // fork prefix kv and thread
    kv = kv_fork(prefix_kv);
    pthread_create({
        pos = prefix_kv.len(), step = 0;
        t = suffix;
        // generate until eos token
        while (t != EOS) {
            d = pred(kv, t, pos + step);
            t = sample(d);
            step++;
            print(detokenize(t));
        }
        kv_remove(kv);
    });
}
join_all_threads();
kv_close(prefix_kv);
```

**Figure 2: Example program demonstrating parallel token generation with shared prefix KV cache.**

```
pred(kv: kv_file, tokens, positions) -> list[dist]
```

The `pred` function accepts two parameters: `kv`, a pointer to the KV cache file (further details in §4.3), and `tokens`, a list of tuples where each tuple contains a token ID and its absolute position within the context. Upon completion of the system call, the KV cache file is updated with new tensors corresponding to the provided `tokens`, and the function returns a list of next token distributions for each input token.

The `dist` contains a list of floating-point numbers. With full access to the token distribution, LIPs implement various decoding algorithms, such as constrained decoding and speculative decoding. For example, to achieve constrained decoding, LIPs integrate a state machine into the generation loop to restrict the distribution variables to those that align with the state machine. For speculative decoding, LIPs pass multiple input tokens (draft tokens) to the `pred` system call and verify them by inspecting the distributions of the tokens.

### 4.2 KV Cache Management via File System

Symphony treats the KV cache as files, enabling it to persist beyond a single process's lifecycle, share across multiple processes, and allow LIPs to dynamically manipulate it. Symphony introduces a specialized file system called *KVFS*. KVFS, similar to traditional file systems, lets LIPs manage files (KV cache) using file APIs similar to POSIX, supplemented by specialized APIs for operations like `extract` and `merge`. KVFS

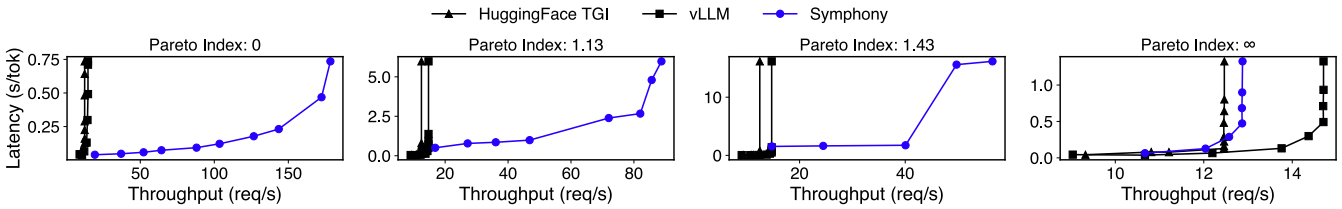


**Figure 3: Estimated performance benefits of prompt caching implemented via LIPs in Symphony. The figure shows normalized mean end-to-end latency per generated token and throughput using the Llama [14] 13B model on NVIDIA A100 GPU. Symphony enables application-specific LLM optimizations, such as caching frequently reused KV cache, without requiring modifications to the serving system design.**

virtualizes GPU memory areas that store token-level KV tensors in pages, using PagedAttention [30].

In a simple text completion scenario, LIPs start by creating an empty file, retrieve its handle with `kv_open`, and use this handle with the prompt in `pred` (See §4.1), which then fills the file with the KV cache. They use the same file in the subsequent autoregressive generation loop. For parallel generation with shared prefixes, LIPs clone the prefix file for each thread (See §4.3), allowing different tokens to fill in without duplicating the actual tensors behind the KV cache.

LIPs can directly manipulate files, enabling them to create new files from existing ones by extracting specific token indices with `extract` or merging existing files into one. This capability benefits inference speedup techniques like runtime context prunning [42, 47], by removing invalid or unimportant tokens from files. When writing files, LIPs can apply a file lock to ensure exclusive access. KVFS enforces access control, allowing only authorized parties to access files. For example, a file containing "system prompts" might be readable by all LIPs but writable only by the admin.

## 4.3 Generations as Threads

To support advanced reasoning strategies that leverage parallel generation, LIPs spawn multiple threads using POSIX thread APIs. For example, a single LIP implements the entire Tree-of-Thought [48] reasoning, with each thread generating one branch of hypotheses, forking recursively if needed, and joining if a generated hypothesis proves unlikely.

LIPs manage I/O independently, eliminating the communication overhead of server-user roundtrips. For instance, LIPs initiate LLM function calls by themselves or incorporate arbitrary computation with the generation process. When I/O with external APIs blocks thread execution, Symphony interrupts to put the thread in a waiting state. Symphony exploits this for resource efficiency: when threads wait for I/O, Symphony offloads their KV caches from the GPU to the CPU and restores them upon I/O completion. Additionally, LIPs communicate directly with each other using inter-process communication, which is useful for implementing cooperative multi-agent systems.

## 4.4 Two-level Scheduling

Symphony implements a two-level scheduling scheme with two distinct schedulers: the thread scheduler and the batch inference scheduler. The thread scheduler handles CPU scheduling tasks typical of operating systems and executes the thread. When a thread triggers the `pred` system call for LLM inference, Symphony transfers the thread to the "inference pool" and marks it as blocked.

Conversely, the inference scheduler aggregates multiple `pred` system calls into a single batch and schedules this batch on the GPU(s). Efficient handling of the `pred` system call by the GPU(s) necessitates batch processing to optimize GPU utilization. Consequently, Symphony must strategically batch these calls. The primary challenge lies in timing the batch execution to achieve peak GPU efficiency. Executing the batch prematurely can result in underutilized GPU resources, diminishing throughput, whereas delaying it excessively can increase wait times for the threads. Symphony dynamically adjusts batch size according to the average frequency of system calls, leveraging models like Poisson process.

# 5 Preliminary Results

We conduct simulated experiments to demonstrate how Symphony enables scalable LLM workloads by allowing users to define custom optimization strategies through LIPs. We compare Symphony with two popular prompt-serving systems, vLLM [30] and TGI [22], in a retrieval-augmented generation (RAG) application scenario. The application inputs a topic, fetches the relevant document, and generates an answer. There are 100 documents, each containing 3,000 tokens.

In this setup, a LIP implements prompt caching [17] by retaining the KV cache for the top 20 most popular topics and discarding it for others. We evaluate throughput and latency under varying request loads and Pareto indices, which model the skewness of the popularity distribution. The results, shown in Figure 3, indicate that Symphony outperforms vLLM and TGI when the Pareto index is small (i.e., when a few topics are queried frequently). This improvement arises from Symphony's ability to leverage increased cache

hits through user-controlled KV cache management. Furthermore, developers can refine LIP logic—e.g., caching only after consecutive requests for the same topic—to optimize performance for specific workload patterns.

We note that these experiments do not yet fully capture the overheads of Symphony. Specifically, the current implementation has a simplified design where all LIPs are the same and start and finish at the same time; therefore, it does not reflect the scheduling overhead and opportunity costs of using different sampling strategies. We will evaluate these aspects in future work.

## 6 Discussion

We believe that bringing programmability to LLM serving systems is a key step towards creating more efficient and capable AI applications. Symphony is a step towards achieving this goal. The transition proposed by Symphony —from serving prompts to executing LLM Inference Programs (LIPs) opens up interesting questions about the design and operation of future LLM serving infrastructure. We discuss key challenges and research directions below.

**Finding the right interface granularity.** Symphony currently provides one system call, `pred`, for model computation. This treats a single LLM model forward pass as an atomic operation. This simplifies system design and batching but limits the application's ability to control the model computation process. Enabling finer-grained access (e.g., to attention mechanisms [3, 51] or layer outputs [10]) could unlock powerful application-specific optimizations. However, such interfaces increase complexity, risk violating model abstraction boundaries, and complicate efficient batch scheduling. Determining the optimal granularity—balancing application empowerment against system manageability and performance predictability—remains an open question.

**Security implications.** Executing user-provided LIPs fundamentally shifts the trust boundary within the serving system. Unlike prompt-based systems where user input is data, Symphony takes code, i.e., LIPs, as input. This introduces security vulnerabilities, such as resource exhaustion, model confidentiality, LLM jailbreaking, and parameter extraction via distribution analysis. Practical deployments necessitate robust sandboxing (e.g., WASM, seccomp filters, lightweight VMs), resource accounting, and fine-grained access control to protect the system and other tenants.

**Performance overhead.** Traditional serving systems achieve high performance through centralized control over batching, scheduling, and GPU pipelining. Symphony decentralizes control, empowering LIPs to manage their generation logic. This potentially sacrifices global optimization opportunities. Symphony's batch scheduler operates with less complete information, potentially leading to suboptimal GPU utilization, especially with heterogeneous LIPs. Furthermore, decoupling the sampling logic into LIPs hinders server-side optimizations such as pipelining sampling with model execution. Designing intelligent scheduling algorithms and potentially new co-design approaches between LIPs and the system to mitigate these performance "opportunity costs" is a key research challenge.

**Beyond the OS analogy.** Framing Symphony in terms of OS concepts helps convey its broader responsibilities, i.e., resource management, concurrency and I/O, using a familiar mental model. However, this analogy serves more as a scaffold than as an implementation prescription. Alternative abstractions may be equally, if not more, suitable for realizing the core ideas. For example, language runtimes that support coroutines, actors, or async tasks could offer a natural fit for enabling concurrent, stateful generation workflows. Lightweight execution environments such as WASM or secure containers could provide a sandboxed context for running user-defined logic. Moreover, we can draw inspiration from other domains where systems evolved into programmable platforms by accepting user-supplied logic, such as OS kernels [4, 34], networking [8], and databases [28]. These systems show that we can expose carefully scoped programmability at key abstraction boundaries without rearchitecting everything from scratch. This raises a question: can existing LLM serving systems be salvaged? In our view, the tight coupling of sampling, caching, and scheduling in today's prompt-centric architectures makes this difficult. We argue that application-level control, spanning model state management, fine-grained control over generation, and tool use, requires a clean break from the existing architecture. In this sense, a redesign like Symphony is not just a matter of elegance but one of necessity.

**Evaluation space.** Standard benchmarks for LLM serving focus on per-prompt throughput and latency and are inadequate for evaluating systems like Symphony. The benefits of programmability manifest in the context of complex, multi-step application workflows involving state management (KV cache reuse), external interactions (function calls), and custom control flow (parallel reasoning). Meaningful evaluation requires new benchmarks that capture end-to-end performance, resource consumption (GPU, CPU, memory), and responsiveness for realistic application scenarios, moving beyond isolated token generation metrics.

## Acknowledgments

This work is supported in part by National Science Foundation (NSF) Athena AI Institute (Award #2112562) and Yale University. The authors thank the reviewers for their constructive comments.

## References


[1] R. Abhyankar, Z. He, V. Srivatsa, H. Zhang, and Y. Zhang. InferCept: Efficient intercept support for augmented large language model inference. In *Proc. ICML*, 2024.

[2] Anthropic. Prompt caching with claude. https://www.anthropic.com/news/prompt-caching. Accessed: 2025-01-16.

[3] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM SOSP*, 1995.

[5] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, and T. Hoefler. Graph of thoughts: Solving elaborate problems with large language models. 2024.

[6] L. Beurer-Kellner, M. Fischer, and M. Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

[7] L. Beurer-Kellner, M. Fischer, and M. T. Vechev. Guiding llms the right way: Fast, non-invasive constrained generation. In *Proc. ICML*, 2024.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[9] G. Chen, X. Yu, and L. Zhong. Typefly: Flying drones with large language model. *CoRR*, abs/2312.14950, 2023.

[10] Y. Chen, X. Pan, Y. Li, B. Ding, and J. Zhou. EE-LLM: large-scale training and inference of early-exit large language models with 3d parallelism. In *Proc. ICML*, 2024.

[11] G. Cloud. Context caching overview. https://cloud.google.com/vertex-ai/generative-ai/docs/context-cache/context-cache-overview. Accessed: 2025-01-16.

[12] Y. Dong, C. F. Ruan, Y. Cai, R. Lai, Z. Xu, Y. Zhao, and T. Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *CoRR*, abs/2411.15100, 2024.

[13] Dottxt-AI. Outlines: Structured text generation. https://github.com/dottxt-ai/outlines, 2025. Accessed: 2025-04-16.

[14] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al. The Llama 3 herd of models. *CoRR*, abs/2407.21783, 2024.

[15] S. Gao, Y. Chen, and J. Shu. Fast state restoration in LLM serving with HCache. In *Proc. EuroSys*, 2025.

[16] S. Geng, M. Josifoski, M. Peyrard, and R. West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proc. EMNLP*, 2023.

[17] I. Gim, G. Chen, S.-S. Lee, N. Sarda, A. Khandelwal, and L. Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proc. MLSys*, 2024.

[18] I. Gim, S.-S. Lee, and L. Zhong. Asynchronous LLM function calling. abs/2412.07017, 2024.

[19] Guidance-AI. Guidance: A guidance language for controlling large language models. https://github.com/guidance-ai/guidance, 2025. Accessed: 2025-04-16.

[20] N. Gupta, H. Narasimhan, W. Jitkrittum, A. S. Rawat, A. K. Menon, and S. Kumar. Language model cascades: Token-level uncertainty and beyond. In *Proc. ICLR*, 2024.

[21] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *Proc. ICLR*, 2020.

[22] Hugging-Face. Text generation inference. https://github.com/huggingface/text-generation-inference, 2023. Large Language Model Text Generation Inference Server in Rust and Python.

[23] W. Jiang, S. Subramanian, C. Graves, G. Alonso, A. Yazdanbakhsh, and V. Dadu. RAGO: Systematic performance optimization for retrieval-augmented generation serving. abs/2503.14649, 2025.

[24] O. Khattab, K. Santhanam, X. L. Li, D. Hall, P. Liang, C. Potts, and M. Zaharia. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *CoRR*, abs/2212.14024, 2022.

[25] S. Kim, S. Moon, R. Tabrizi, N. Lee, M. W. Mahoney, K. Keutzer, and A. Gholami. An LLM compiler for parallel function calling. abs/2312.04511, 2023.

[26] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein. A watermark for large language models. In *Proc. ICML*, 2023.

[27] M. Kuchnik, V. Smith, and G. Amvrosiadis. Validating large language models with ReLM. In *Proc. MLSys*, 2023.

[28] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter:bare-metal extensions for multi-tenant low-latency storage. In *Proc. USENIX OSDI*, 2018.

[29] A. Kumar, C. Agarwal, S. Srinivas, S. Feizi, and H. Lakkaraju. Certifying LLM safety against adversarial prompting. abs/2309.02705, 2023.

[30] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proc. ACM SOSP*, 2023.

[31] S. Lee and G. Kim. Recursion of thought: A divide-and-conquer approach to multi-context reasoning with language models. In *Proc. ACL*, 2023.

[32] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *Proc. ICML*, 2023.

[33] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *Proc. USENIX OSDI*, 2024.

[34] Linux Foundation. ebpf - extended berkeley packet filter. https://ebpf.io/, 2024. Accessed: 2025-04-17.

[35] Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan, M. Maire, H. Hoffmann, A. Holtzman, and J. Jiang. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proc. ACM SIGCOMM*, 2024.

[36] OpenAI. Prompt caching guide. https://platform.openai.com/docs/guides/prompt-caching. Accessed: 2025-01-16.

[37] OpenAI. Parallel function calling, 2023. Accessed: 2024-10-26.

[38] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis. In *Proc. NeurIPS*, 2024.

[39] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference. In *Proc. MLSys*, 2023.

[40] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[41] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a single GPU. In *Proc. ICML*, 2023.

[42] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo. Ast-trans: Code summarization with efficient tree-structured attention. In *Proc. ACM/IEEE ICSE*, 2022.

[43] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong. Solving olympiad geometry without human demonstrations. 2024.

[44] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024.

[45] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better LLM agents. In *Proc. ICML*, 2024.

[46] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. abs/2308.08155, 2023.

[47] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient streaming language models with attention sinks. In *Proc. ICLR*, 2024.

[48] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Proc. NeurIPS*, 2023.

[49] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *Proc. ICLR*, 2023.

[50] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi. The shift from models to compound AI systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

[51] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Proc. NeurIPS*, 2023.

[52] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. W. Barrett, and Y. Sheng. SGLang: Efficient execution of structured language model programs. In *Proc. NeurIPS*, 2024.