Pie: A Programmable Serving System for Emerging LLM Applications

In Gim
Yale University

Zhiyao Ma
Yale University

Seung-seob Lee
Yale University

Lin Zhong
Yale University

Abstract

Emerging large language model (LLM) applications involve diverse reasoning strategies and agentic workflows, straining the capabilities of existing serving systems built on a monolithic token generation loop. This paper introduces Pie, a programmable LLM serving system designed for flexibility and efficiency. Pie decomposes the traditional generation loop into fine-grained service handlers exposed via an API and delegates control of the generation process to user-provided programs, called *inferlets*. This enables applications to implement new KV cache strategies, bespoke generation logic, and seamlessly integrate computation and I/O—entirely within the application, without requiring modifications to the serving system. Pie executes inferlets using WebAssembly, benefiting from its lightweight sandboxing. Our evaluation shows Pie matches state-of-the-art performance on standard tasks (3-12% latency overhead) while significantly improving latency and throughput (1.3×-3.4× higher) on agentic workflows by enabling application-specific optimizations.

CCS Concepts: • **Computing methodologies** → Natural language processing.

Keywords: LLM serving, programmable inference, KV cache

ACM Reference Format:
In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. 2025. Pie: A Programmable Serving System for Emerging LLM Applications. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3731569.3764814

1 Introduction

Large language model (LLM) usages are evolving far beyond simple text completion. Today's LLM serving systems require sophisticated strategies to execute LLM application logic ranging from token-level [9, 27, 55] to workflow-level [21, 77, 84]. These systems must also handle real-time user events, such as audio [4] and vision [67], while seamlessly integrating with external tools [1, 20, 80] and code execution environments [52, 68, 71] to support increasingly common agentic workflows [70, 81].

These emerging LLM applications reveal fundamental limitations in the efficiency and flexibility of today's serving infrastructure (§2), prompting us to identify three requirements for next-generation LLM serving systems:

R1. **Application-specific KV cache control.** LLM workflows such as tree-of-thought reasoning [75], map-reduce summaries [7], multi-step generations [31] require explicit, fine-grained control over KV cache management. This includes customizing allocation [38], eviction policies [80], and reuse strategies [84] according to application-specific logic, rather than relying on the implicit, system-wide heuristics built into serving systems.

R2. **Customizable generation processes.** Emerging decoding methods (assisted decoding [13], MCTS-based generation [25]), stateful sampling strategies [27], and safety grounding [36] require the ability to precisely control and potentially modify the token prediction and sampling loop on a per-request or even per-step basis.

R3. **Integrated computation and I/O.** Agentic workflows and interactions with external systems necessitate tightly coupling token generation with arbitrary computations (*e.g.*, running symbolic checks [68]) and I/O operations (*e.g.*, making API calls [63, 65]) within the generation flow, without incurring prohibitive latency penalties or complex external orchestration.

Existing systems, such as vLLM [38] and TGI [29], struggle to meet these requirements due to their reliance on an inflexible, monolithic token generation loop (Figure 1). This architecture enforces global policies (*e.g.*, for KV cache management, hindering R1), employs a closed generation process that resists customization (hindering R2), and isolates inference from external operations, such as tool use, forcing the client to coordinate such operations or otherwise requiring custom modifications to the serving system (hindering R3). While recent systems like SGLang [84] and Parrot [44] introduce some programmatic abstractions, they are fundamentally constrained by the same monolithic design (§9).

This paper introduces Pie, a programmable LLM serving system designed to meet these requirements. The core idea of Pie is to *delegate control over the end-to-end generation process to user-provided programs*, called *inferlets*. Pie decomposes the process into fine-grained handlers for stages like embedding, forward pass, and sampling (Figure 2), instead

SOSP '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/2025/10
https://doi.org/10.1145/3731569.3764814

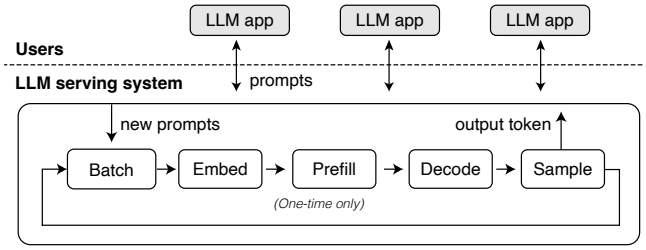

Figure 1. Current LLM serving systems conceptually follow a monolithic prefill-decode loop, batching prompts and applying global policies for KV cache. This design lacks flexibility to support application-specific logic.

of a global generation pipeline. Inferlets orchestrate these handlers via API calls (§4), enabling them to explicitly manage resources like the KV cache, define custom generation logic, and directly integrate arbitrary computation and I/O.

Instead of treating prompts as the basic unit of service, Pie elevates programs to this role, allowing hundreds of concurrent inferlets to adopt distinct optimization strategies. For example, one may exploit custom KV caching, another speculative decoding [43], and yet a third may exploit a complex agentic loop, all on the same underlying serving infrastructure as illustrated by Figure 2. Viewed through this lens, all existing serving systems effectively operate with a single, fixed inferlet, i.e., an autoregressive loop, which explains why they are inflexible for modern AI applications.

We implement Pie using a layered architecture (§5) and build its inferlet runtime around WebAssembly (Wasm). Developers can program inferlets using any programming language that compiles to Wasm (*e.g.*, C++, Rust, Python) with the API bindings provided by Pie.

Our evaluation (§7) demonstrates the effectiveness of Pie. We implement a diverse range of LLM techniques as inferlets, including attention variants [5, 74], constrained and speculative decoding [9, 61], deliberate prompting strategies [7, 75], and agentic workflows. We show that Pie matches state-of-the-art performance for traditional tasks (*e.g.*, 3-12% latency overhead for text completion) while significantly outperforming existing systems on advanced tasks like Graph-of-Thought and agentic workflows (1.1×-2.4× lower latency, 1.3×-3.4× higher throughput) by enabling application-specific optimizations.

In summary, we report the following contributions.

- Identifying three key limitations in current LLM application serving: implicit resource control, inflexible generation processes, and poor workflow integration.
- The Pie architecture, which decomposes the monolithic generation pipeline into fine-grained service handlers.
- A Turing-complete programming model providing applications with full and end-to-end control over the generative workflow, KV cache, and I/O.

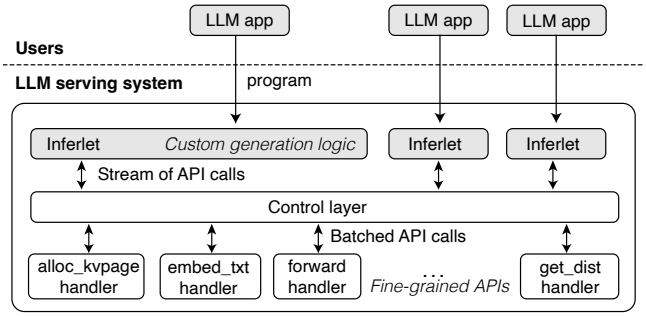

Figure 2. Our proposed system, Pie, dismantles the sequential generation process into independent handlers, and delegates control to user-provided programs called *inferlets*.

- Implementation and an evaluation demonstrating Pie matches state-of-the-art performance on standard tasks while significantly improving latency and throughput on emerging applications.

Pie is open-sourced at https://github.com/pie-project/pie.

2 Background and Motivation

We first outline the architecture underpinning existing LLM serving systems. We then detail how this architecture falls short in supporting increasingly complex LLM applications.

2.1 How LLM serving systems work

Existing LLM serving systems [2, 19, 29, 38, 64, 79] are designed for high-throughput text completion, treating each user request as a single input prompt to be processed by a prefill-and-decode model (Figure 1). This generation process is composed of discrete steps. It begins with a prefill step, where the system processes the entire prompt in parallel to populate an initial Key-Value (KV) cache to avoid redundant computation. The system then enters an iterative decode phase, where each step uses the KV cache and the last token to predict and sample the next token, then updates the cache. To maximize hardware utilization, a central scheduler groups multiple requests (prompts) into a batch, which advances all of them through each generation step in lockstep.

2.2 Limitations of existing serving systems

The current LLM serving architecture centers around three key aspects whose inherent inflexibility creates challenges for emerging applications: (1) the management of the KV cache, (2) the fixed structure of the token prediction and sampling loop, and (3) the integration of the generation process with external computations or workflows.

Implicit KV cache management. Current systems typically manage the KV cache using implicit, system-wide policies, such as LRU eviction [38, 84] or prompt caching [21, 47], where common leading token sequences across requests share cached entries. While efficient for simple prompt

variations, this approach fails to provide the application-specific control required by advanced strategies. Techniques like Recursion-of-Thought [41] and Graph-of-Thought [7] demand explicit, runtime decisions about which specific KV cache blocks to retain, discard, reuse, or duplicate based on the application's dynamic state or reasoning structure—control that global heuristics cannot provide. Furthermore, fine-grained manipulation of the KV cache structure is needed for techniques like attention sink [74] and beam search [18]. Implementing such features within the monolithic architecture often requires invasive modifications deep within the system's memory manager and scheduler. The engineering effort can hinder adoption; for example, support for beam search was considered for removal from vLLM (v0.6.3) due to its complexity [37].

Inflexible decoding process. The tightly coupled "predict-then-sample" operation within the monolithic loop offers limited flexibility for customizing the core generation algorithm. Emerging decoding methods like parallel decoding [23, 61] or speculative decoding [43, 62] deviate from the standard decoding process, often predicting multiple tokens per step or using verification steps. Integrating these is challenging because their variable output granularity interferes with the regular process, which assumes a single token is generated per request in each batching step. Consequently, such techniques are often implemented as system-wide toggles [29, 38, 84], rather than being flexibly selectable or configured on a per-request basis. This lack of per-request customization is inefficient, as different applications or even different phases within one application might benefit from distinct decoding strategies. Similarly, stateful generation strategies (MCTS [25], grammar-constrained decoding [8, 9]) require managing state across steps, which is cumbersome in the standard stateless loop. Lastly, dynamic control over the output distribution (for watermarking [34], safety filters [36]) requires injecting logic around the sampling step, for which current systems lack clean, application-customizable hooks.

Poor workflow integration. The current architecture inherently assumes a "closed-loop" generation process, where the system focuses solely on producing tokens based on the initial prompt and the model's internal state. This is ill-suited for agentic workflows [31, 76] or interactive applications that must integrate token generation with arbitrary computations, such as code execution [52, 65], API calls [57], symbolic solvers [68]. Integrating such external computations requires an inefficient workaround: the generation process must return control to the client, which performs the external action and then submits a new request with the updated context to resume generation. This round-trip incurs network latency [1]. More critically, because serving systems are stateless across requests, the state from the initial generation, i.e., the KV cache, is discarded. The system treats the continuation as a new prompt, forcing a costly re-prefill of the interaction history [20, 80]. While techniques like prefix caching can mitigate this, they are system-level heuristics rather than a native solution for stateful interaction. Additionally, incorporating application-specific pre- or post-processing steps (*e.g.*, custom tokenization adjustments) into the request lifecycle is also challenging without introducing overly complex APIs.

Collectively, these limitations reveal a mismatch: The static, monolithic design optimized for batched text completion cannot efficiently or flexibly accommodate the dynamic, heterogeneous, and interactive nature of modern LLM applications. The root cause lies in the tight coupling of application control logic with the core execution engine within a fixed architectural pattern. Overcoming these challenges requires a paradigm shift towards decoupling the application's control logic from the underlying model execution infrastructure, as embodied by Pie.

3 Overview of Pie

Pie achieves programmability through two fundamental architectural shifts, derived from the insight that application-specific control logic can be efficiently executed *outside* the core inference engine if provided with the right interfaces. First, Pie *dismantles the monolithic decoding loop into fine-grained procedures*. Instead of a fixed prefill-decode loop, it exposes a set of independent, fine-grained services (called *handlers*) responsible for specific token prediction stages like input text embedding, KV cache operations (*e.g.*, allocation, update), and model forward pass. Second, Pie *delegates end-to-end control to user-provided programs*, called *inferlets*. These inferlet programs define bespoke generation workflows by issuing API calls to the handlers. Inferlets can:

1. *Manage resources explicitly*, particularly allocating, manipulating, and reusing KV cache pages based on application logic. This provides the fine-grained control needed for advanced caching strategies (R1 in §1).
2. *Orchestrate handlers* precisely, defining custom sequences of operations beyond standard decode loop. This addresses the need for customizable generation processes (R2).
3. *Integrate arbitrary computation and I/O* seamlessly within their control flow (*e.g.*, calling external APIs, running checks). This tackles the poor workflow integration of existing systems (R3).

To realize this programmable model, Pie employs a layered architecture designed for flexibility and efficiency. At a high level, user inferlets execute within a dedicated *application layer*, interacting with the system via the API. The *control layer* orchestrates these interactions, managing resources, virtualizing access, and intelligently batching API calls destined for the hardware. Finally, the *inference layer* executes these batched operations, translating them into

low-level computations (*e.g.*, GPU kernel invocations) using specialized handlers for different model operations.

The following sections elaborate on the specifics of this design. Section 4 details the Pie *programming model*, outlining the API primitives that inferlets use to orchestrate generation and manage resources. Section 5 elaborates on the *three-layer system architecture*, explaining how it supports the execution of inferlets, manages control flow, and interfaces with the underlying inference hardware.

4 Programming Model and API

The Pie programming model provides an API for developers to create what we call *inferlets*—specialized programs that orchestrate LLM generation. Each inferlet executes within a single-threaded, event-driven runtime. Concurrency within a inferlet is handled through asynchronous, non-blocking API calls, a model well-suited for I/O-bound agentic workflows that spend significant time waiting for API calls or tool execution to complete.

The programming model for inferlets addresses three key goals: (1) *expressiveness*—supporting a wide range of generation logic, particularly addressing requirements R1-R3 outlined in §1; (2) *efficiency*—enabling performance optimizations through fine-grained resource control; and (3) *extensibility*—remaining agnostic to specific LLM architectures, while providing a foundation for future extensions.

Table 1 provides an overview of Pie APIs. Among the 42 APIs, 18 are core to defining the LLM forward pass and resource management in the inference layer, which we elaborate on in §4.1. The remaining 24 APIs are for runtime management, inter-inferlet communication, and I/O, which are essential for building interactive, agentic applications. These APIs do not require GPU processing and are handled entirely by the control layer. We describe them in §4.3.

Scope and tradeoffs. The Pie API is designed for Transformer-based LLMs. It abstracts the workflow into three stages, exposing each as an opaque function accessible through the API. As a result, low-level GPU optimizations, such as kernel fusion and scheduling [12], quantization techniques [45], or tensor-parallel execution strategies [83], remain orthogonal concerns delegated to the inference layer (§5). The Pie API prioritizes fine-grained programmability over simplicity. This incurs programming complexity. We ease the burden through a high-level support library that provides common subroutines and abstractions (§6.3).

4.1 Abstractions

Pie views the LLM forward pass as a three-stage process: (1) **embed**, which prepares input embeddings from raw data (text, images); (2) **forward**, which computes output embeddings from input embeddings, potentially utilizing the KV cache; and (3) **sample**, which derives meaningful outputs (*e.g.*, next-token logits) from output embeddings. APIs related to the model forward pass fall into one of these three categories. Inferlets combine API calls from each of these stages to define a complete forward pass.

Resources. Pie defines two primary resources that can be passed from stage to stage by pointer. (1) Embed represents a sequence of token embeddings (input or output), allocated on a per-token basis to offer flexibility in manipulating individual token representations. (2) KvPage represents a contiguous chunk of the KV cache, following PagedAttention [38]. A KvPage can consist of 8–32 tokens. Pie requires explicit resource allocation and deallocation (*e.g.*, `alloc_embed` and `alloc_kvpage` in Table 1) by inferlets. The returned resource handles are opaque pointers to the underlying memory managed by Pie. Each inferlet has its own virtual resource address space, managed by Pie's control layer. It is possible to share resources between inferlets through import/export APIs (Table 1), where the control layer handles the necessary physical-virtual resource address mappings.

Command queue. Command queue abstracts a logical sequence of API calls issued by an inferlet. Its purpose is to inform the batch scheduler (§5.2) to make optimal batching decisions by (1) making the dependencies between API calls unambiguous, and (2) enabling the setting of priorities between different queues. All API calls that are batch processed by the batch scheduler, such as resource allocation/deallocation, forward pass, and sampling, take a pointer to a command queue object (Queue) as an argument. On the other hand, API calls that do not require batch processing, such as I/O operations, do not need a command queue argument and are handled directly by the control layer. For example, in the following inferlet program, Pie's control layer may batch the two `forward` API calls depending on different Queues so that GPU can process them in parallel in the inference layer.

```
q1,q2 = create_queue(), create_queue()
forward(q1, [], iemb1, [], oemb1)
forward(q2, [], iemb2, [], oemb2)
await synchronize(q1)
await synchronize(q2)
```

4.2 APIs

Embed API. The embed APIs are responsible for converting raw input data (*e.g.*, text, images) into Embed pointers, which can then be utilized by other APIs. Key functions in this category include `embed_txt` and `embed_img`, which populate allocated Embed pointers based on the provided input data. Additionally, Pie offers auxiliary functions such as `tokenize`, `detokenize`, and `num_embeds_needed` to facilitate preprocessing and embedding-related operations in this stage.

Forward API. `forward` and `forward_with_adapter` fall under this category. The `forward` API executes the core Transformer

Table 1. Overview of APIs for programming inferlets (non-exhaustive). Pie provides a total of 42 APIs, with 18 dedicated to LLM execution and the remainder supporting core runtime operations and agentic workflow. To ensure model-agnosticism and extensibility, Pie organizes LLM-related APIs into traits (§4.4). API calls that involve a command queue (`q`) are processed by the inference layer, while those that do not are directly handled by the control layer.

Trait (Supertraits)	Function	Behavior
	`get_arg() -> list[str]`	Gets command-line arguments passed during launch.
	`send(msg)`	Sends a message to the client that launched the inferlet.
	`receive() -> future[str]`	Receives messages from the client.
	`http_get(url) -> future[str]`	Performs an HTTP GET request to the specified URL.
	`available_models() -> list[Model]`	Gets the list of models.
	`available_traits(model) -> list[str]`	Gets the list of given model's traits.
	`create_queue(model) -> Queue`	Creates a new command queue.
	`synchronize(q) -> future`	Blocks until the queue finishes execution.
	`set_queue_priority(q, pri)`	Hints the controller which queue to process first.
Allocate	`export_kvpage(kv, name)`	Exports paged KV cache for use in other programs.
	`import_kvpage(name) -> list[KvPage]`	Imports the paged KV cache.
	`alloc_kvpage(q, size) -> list[KvPage]`	Allocates memory for paged KV cache.
	`dealloc_kvpage(q, kv)`	Deallocates memory for paged KV cache.
	`alloc_emb(q, size) -> list[Embed]`	Allocates buffer space for embeddings.
	`dealloc_emb(q, emb)`	Deallocates buffer space for embeddings.
	`copy_kvpage(q, src, dst)`	Copy KV cache contents at token-level.
Forward (*Allocate*)	`forward(q, ikv, iemb, okv, oemb, mask)`	Transform to KV cache and/or output embeddings.
	`mask_kvpage(q, tgt, mask)`	Masks the KV cache in a token-level manner.
InputText (*Allocate, Forward*)	`embed_txt(q, tok, pos, embs)`	Embeds text input into embeddings
InputImage (*Allocate, Forward*)	`num_embs_needed(m, size) -> int`	Calculates the number of embeddings needed.
	`embed_img(q, blob, embs)`	Embeds image input into embeddings.
Tokenize (*InputText*)	`tokenize(q, text) -> list[int]`	Converts text into a list of token IDs.
	`detokenize(q, token_ids) -> str`	Converts token IDs back into text.
	`get_vocabs(q) -> list[list[byte]]`	Retrieves the vocabulary list.
OutputText (*Allocate*)	`get_next_dist(q, emb) -> future[Dist]`	Gets the next token distribution.

forward pass. `forward_with_adapter` allows users to specify LoRA adapters [28] to support fine-tuned models. The `forward` API takes input Embeds and/or input KvPages, performs the attention and transformation steps, and populates output Embeds and/or output KvPages. `forward` operates based on explicit sequence positions associated with the resources. The API either accepts an explicit attention mask in boolean matrix form or infers it from the provided sequence positions. For example, omitting the KvPage of a token that precedes the input Embeds effectively masks that token during attention. This allows inferlets to directly manipulate the attention context, essential for techniques leveraging custom attention masks [5, 74, 74] or modular caching [21]. We illustrate its usage in the code below. Lines 1-3 generate one output embedding given the 9 tokens (i.e., prefill). The same operation can be split into two `forward` calls, as shown in lines 6-7, using a KvPage to store intermediate states. We note that the `forward` API calls in lines 6-7 may be batched together, even though they use the same command queue, which we cover in more detail under *vertical batching* (§5.2).

```
1 iemb = alloc_emb(9)      # 9 input tokens
2 oemb = alloc_emb(1)      # 1 output token
3 forward([], iemb, [], oemb)
4 # in examples, 1 KvPage = 1 token
5 kv = alloc_kvpage(8)           # 8 tokens
6 forward([], iemb[:-1], kv)  # gen kvpage
7 forward(kv, iemb[-1:], oemb)# use kvpage
```

Sample API. Pie provides APIs to extract the final outcomes from the output embeddings produced by the forward APIs. For instance, `get_next_dist` takes an Embeds as input and outputs the next-token probability distribution. The inferlet then controls sampling or other post-processing using the host language. This flexibility is useful for implementing custom sampling schemes [35, 36] and probability-based LLM introspection such as LLM watermarking [34] that require broad access to the underlying token distribution. Returning the full distribution to the inferlet may incur significant memory overhead for large vocabularies (*e.g.*, 128K vocabularies in Llama 3). To mitigate this, Pie truncates the distribution to the top-K tokens. K is configurable by the user, with a default of 256 vocabularies.

Putting it all together. The primitives described above provide the building blocks for custom generation logic. Below, we implement an autoregressive loop that generates 10 new tokens given the input prompt "Hello," using greedy sampling. Since only a single command queue is used in this example, the Queue argument is omitted for brevity.

```
prom = tokenize("Hello, ")
tok_limit = len(inp) + 10

# Resource allocation
prom_emb = alloc_emb(len(prom))
gen_emb = alloc_emb(1)
kv = alloc_kvpage(tok_limit)

# Prefill
pos = list(range(len(prom)))
embed_txt(prom, pos, prom_emb)
forward([], prom_emb,
     kv[:len(prom)], gen_emb)

# Decode
for i in range(len(prom), tok_limit):
  dist = await get_next_dist(gen_emb)
  gen = dist.max_index()
  print(detokenize(gen))
  embed_txt(gen, [i], gen_emb)
  forward(kv[:i], gen_emb,
       kv[i:i+1], gen_emb)

# Resource cleanup
dealloc_emb(prom_emb)
dealloc_emb(gen_emb)
dealloc_kvpage(kv)
```

The inferlet programming model exposes fine-grained control similar to OpenGL in GPU graphics programming. Developers who need low-level flexibility can directly orchestrate resources and generation logic, while most users can rely on higher-level abstractions or reusable libraries built atop this foundation. In practice, many applications will only use these libraries or simply run pre-compiled inferlets that suit their needs. For instance, Pie's support library (§6.3) allows users to implement the above autoregressive loop in just three lines of code:

```
ctx = Context(model)
ctx.fill("Hello, ")
ctx.generate_until(max_tokens=10)
```

4.3 Supporting agentic workflows

Pie provides essential runtime APIs, such as reading command-line arguments (`get_arg`), querying available models (`available_models`), and enabling communication between user (i.e., the client that initiated the inferlet) and inferlets (`send`, `receive`). Additionally, inferlets can independently perform I/O operations, such as interacting with external servers using networking APIs (`http_get`, `http_post`) or collaborating with other inferlets through message-passing APIs (`broadcast`, `subscribe`). These capabilities enable the creation of dynamic, interactive workflows that seamlessly integrate LLM inference with external data, tools, computations, and other agents.

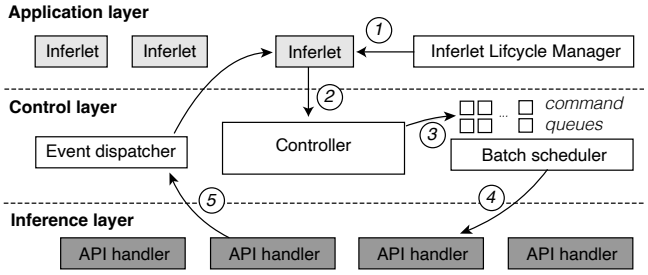

Figure 3. Inferlet service workflow (§5). The application layer executes inferlets that make API calls to the control layer whose batch scheduler adaptively batches these calls and forwards them to the inference layer. Results are sent back to the control layer and then to the inferlets.

4.4 API extensibility

LLMs vary in their capabilities (*e.g.*, text-only, multimodal) and may gain new functionalities over time. To provide a stable yet extensible API, Pie defines each set of related operations as a `Trait` (similar to traits in Rust). For example, the `InputText` trait includes `tokenize` and `embed_txt`, while the `Forward` trait includes `forward`. Specific model implementations then realize one or more traits, and traits themselves can have dependencies (i.e., supertraits). Table 1 summarizes the currently defined traits. Adding support for new modalities (*e.g.*, audio input) or capabilities simply requires defining a new trait and having models implement it. Existing inferlets written against older traits continue to work unchanged. Inferlets can query the traits supported by a model at runtime using `available_traits` and adapt their logic accordingly.

On the other hand, traits can be used to extend the API set for advanced optimizations that are not possible with the current API sets. For example, schemes like PyramidKV [11] and AdaKV [17] rely on internal model statistics like token-level attention scores. One can define a variant of the Forward trait, say, `IntrospectiveForward`, that returns such statistics to allow inferlets to implement custom caching strategies.

5 System Architecture

Pie employs a three-layer architecture—application, control, and inference—as illustrated in Figure 3, to efficiently serve multiple user inferlets. This design separates concerns: The application layer manages the lifecycle of individual inferlets; the control layer oversees system-wide resource coordination, including batch scheduling of API calls; and the inference layer handles low-level, GPU-specific execution of model inference tasks.

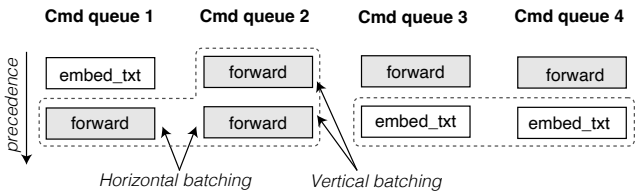

Figure 4. Batch scheduling example (§5.2). A batch of two `embed_txt` API calls from command queues 3 and 4, or a batch of three `forward` API calls from queue 1 and 2, is eligible to be dispatched to the inference layer. Horizontal batching groups calls across different command queues, while vertical batching groups consecutive calls of the same type within the same queue if they do not conflict.

5.1 Application layer

The application layer executes inferlets in a runtime that provides isolation between them. Pie uses WebAssembly (Wasm) runtime to run and sandbox inferlets. It serves as an interface for inferlets to interact with Pie via the API in §4. The Inferlet Lifecycle Manager (ILM) in the application layer manages inferlet lifecycles, including creation, destruction, and communication. It provides RPCs for querying, submitting, and launching inferlets with Wasm binaries and arguments (① in Figure 3). Users can communicate with inferlets through the ILM after launch. The `send` and `receive` APIs in inferlets handle these user events.

5.2 Control layer

The control layer bridges application and inference layers by batching Pie API calls and routing them to inference layer operations while providing resource abstraction. A central controller handles API calls from many concurrent inferlets (② in Figure 3), performing three key functions: (1) handling non-GPU API calls (*e.g.*, core runtime API calls and I/Os) directly, (2) managing allocation and virtual address mappings of Embed and KvPage resources (§4.1), and (3) using a batch scheduler to group GPU API calls (*e.g.*, `forward`, `embed`) for performance efficiency (③). The event dispatcher receives and broadcasts results from the inference layer back to the inferlets (⑤).

Resource management. The control layer manages a global pool of Embed and KvPage resources, physically located in the inference layer, with a size configurable at startup based on GPU memory. It provides inferlets with a virtualized view of these resources via allocation (*e.g.*, `allocate_kvpage`) APIs to provide resource isolation across inferlets. To handle resource contention, the control layer uses a First Come First Serve (FCFS) policy, terminating the most recently created inferlets until sufficient resources are freed.

Batch scheduler. To enhance GPU utilization and overall throughput, the batch scheduler groups compatible GPU-bound API calls before dispatching them to the inference layer. As shown in Figure 4, the scheduler manages pending API calls per command queue (§4.1). By leveraging the unambiguous dependencies and priorities provided by command queues, it forms batches using two techniques:

1. *Vertical batching*. Consecutive commands of the same type within a queue can be batched together, provided they do not conflict, such as writing to the same KvPage.
2. *Horizontal batching*. Commands of the same type across different queues can be batched together.

Within a batch, calls from higher-priority queues are placed earlier. If a batch would exceed the inference backend's maximum supported size, the scheduler truncates from the tail. When multiple API types have eligible batches, the scheduler selects the batch whose oldest pending call has waited the longest. A central challenge is deciding *when* to dispatch: flushing too early underutilizes kernels; waiting too long inflates latency. Pie uses a work-conserving policy (§6.1) to maximize GPU utilization .

5.3 Inference layer

The inference layer acts as the hardware execution backend for Pie. It receives batched API calls from the control layer via RPC and manages the physical Embed and KvPage resources, which are allocated based on the configuration set by the control layer at startup. Importantly, resource management is entirely delegated to the control layer, while the inference layer retains the actual memory.

The inference layer contains multiple API handlers, each specialized for executing a specific type of batched API call (*e.g.*, `forward`) that requires GPU execution. Upon receiving a batch (④ in Figure 3), a handler resolves the API call by invoking the necessary GPU-side tensor operations and sends the results back to the control layer's Event Dispatcher (⑤). The inference layer resides on the host server connected to the hardware accelerators, and communicate with the control layer using inter-process communication (IPC).

6 Implementation

Pie is implemented with 13,650 source lines of code (counted with `cloc`), including 11,640 lines for the core system and the support library in Rust, and the remaining lines for the GPU-side API handler implementation in Python. Pie uses wasmtime [10] as the Wasm runtime to execute inferlets and employs WASI to provide system interfaces like network I/O. The API handlers currently support the Llama family of models. We highlight two important implementation aspects that enhance system performance, and additionally describe a support library that simplifies development by providing higher-level abstractions for common generation patterns.

6.1 Adaptive batch scheduling

We implement the batch scheduler (§5.2) using a simple work-conserving strategy. The GPU can be in one of two states:

busy or *idle*. When the GPU is busy, i.e., processing some API calls, the scheduler queues incoming API calls. As soon as the GPU becomes idle, the inference layer immediately notifies the control layer (via IPC) to trigger batch formation.

6.2 API handlers

We implement the API handlers of the inference layer using PyTorch [56] and the FlashInfer GPU kernel library [78]. These handlers communicate with the control layer via ZeroMQ, a messaging library that supports IPC [82]. We also provide a native C++/CUDA implementation of the inference layer that does not depend on PyTorch as part of our open-source release. In our measurements it achieves 10-30% lower end-to-end latency and more efficient GPU memory utilization than the Python/PyTorch counterpart, owing to custom memory management that pre-allocates and reuses GPU buffers to avoid fragmentation. Because this implementation currently supports only a subset of API traits (e.g., Forward, InputText), we omit a full evaluation here. Interested readers can find further details on the project page.

6.3 Support library

We provide a support library, written in Rust, to simplify inferlet development. It provides procedural macros and a lightweight asynchronous runtime to reduce the boilerplate code required to define the Wasm entrypoint and enable the usage of async/await syntax directly in inferlets. We offer high-level abstractions, such as `Context` for automatically managing KvPage, and implement common subroutines such as sampling methods (*e.g.*, top-k, temperature), stopping criteria (*e.g.*, end-of-sequence or maximum tokens), and fork-join parallelism similar to SGLang's API, reducing the need to reimplement these features in each inferlet.

7 Evaluation

In our evaluation, we address the following questions:

Q1. How does Pie's programming model (§4) facilitate the deployment of emerging LLM applications while meeting requirements R1 to R3 outlined in §1? (§7.1-§7.2)
Q2. Can Pie effectively lower the end-to-end latency and improve the throughput of LLM applications compared to existing LLM serving systems? (§7.3)
Q3. What overhead is introduced by Pie's programmable LLM serving system design? (§7.4)

Setup. For our evaluation, we utilize a GCP VM G2 instance (`g2-standard-32`) equipped with an NVIDIA L4 GPU featuring 24 GB of memory to host Pie and baselines. We use Llama 3 models [15] (1B, 3B, 8B) with BF16 weights; KV cache and activations also use BF16. We measure end-to-end latency from a remote Python client in a campus network connecting to the serving system on the GCP instance.

Table 2. LLM applications and inference techniques implemented as inferlets, along with their lines of code (LoC) and compiled Wasm binary sizes. The right column lists support from vLLM (V), SGLang (S), and LMQL (L). The Wasm binary sizes can be reduced by stripping debug symbols. Some applications (*e.g.*, CodeACT) have larger binaries due to embedded libraries such as a JavaScript runtime.

Technique	R1-3 (§1)	LoC	Wasm	Supported
Support library (§6.3)		728	-	
Text completion		38	129 KB	V, S, L
ToT [75]	R1, R3	198	148 KB	S
RoT [41]	R1, R3	106	152 KB	
GoT [7]	R1, R3	87	171 KB	
SKoT [53]	R1, R3	82	173 KB	S
Prefix caching [38]	R1	45	131 KB	V, S
Modular caching [21]	R1	72	139 KB	
EBNF decoding [13]	R2	225	2 MB	V, S, L
Beam search [18]	R2	98	142 KB	V, L
Watermarking [34]	R2	43	130 KB	
Output validation [35]	R2	52	131 KB	
Speculative decoding [62]	R2	255	152 KB	V
Jacobi decoding [61]	R2	88	96 KB	
Attention sink [74]	R1	60	133 KB	
Windowed attn [5].	R1	60	133 KB	
Hierarchical attn. [66]	R1	42	130 KB	
Agent-ReACT [76]	All	60	309 KB	
Agent-CodeACT [71]	All	62	6.7 MB	
Agent-SWARM [85]	All	95	135 KB	

Baselines. We compare Pie with state-of-the-art LLM serving systems: vLLM [38] (v0.6.0), SGLang [84] (v0.4.4), and specialized frameworks such as LMQL [8] (v0.7.3) for structured generation and StreamingLLM [74] for attention sink where relevant. To foster a fair comparison focused on architectural differences rather than kernel optimizations, Pie, vLLM, and SGLang all use the FlashInfer GPU backend [78].

Applications. To demonstrate the expressiveness and performance of Pie (Q1, Q2), we implement a wide range of LLM applications using the Pie API and Rust, detailed in Table 2. These cover standard techniques, advanced reasoning strategies, and agentic workflows. For baseline comparisons, we replicate the same high-level application logic using Python scripts interacting with the respective system's API server.

7.1 Serving agentic workflows

Agentic workflows, which integrate LLM inference with external tools or multi-step reasoning, highlight the importance of seamless computation and I/O integration. We implement three representative agents–ReACT (web API interactions), CodeACT (code execution), and Swarm (inter-agent

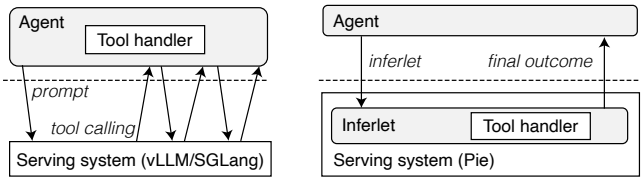

Figure 5. Implementation comparison of agentic workflows in vLLM/SGLang (left) vs. Pie (right).

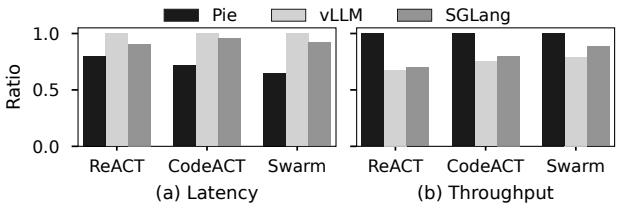

Figure 6. Latency (lower the better) and **throughput** (higher the better) of LLM agents (X axis: ReACT, CodeACT, and Swarm) hosted by different serving systems (Pie, vLLM, and SGLang). Numbers are normalized to the longest latency or the greatest throughput in each case.

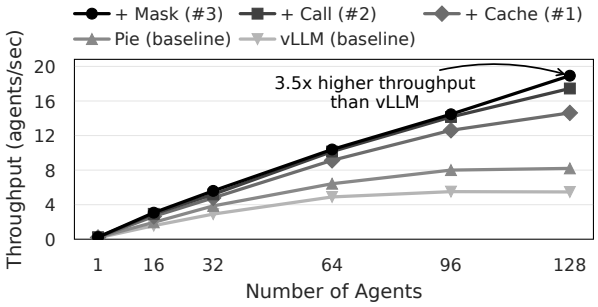

Figure 7. Performance gains via applying workload-specific optimizations to the simple agentic workflow (§7.2). Stacked optimizations further improve the performance.

communication)–as inferlets, showcasing Pie's ability to co-locate I/O. For comparison, we replicate the same workflows in vLLM and SGLang using Python scripts, applying best-effort optimizations and parallelizing client requests where beneficial for throughput.

The main differences in implementation between Pie and the baseline systems are illustrated in Figure 5. In baselines, the application logic only resides within the client, with the serving system handling only LLM inference. This separation incurs latency due to round trips for each external interaction, as well as potential re-prefills when the context changes. In contrast, Pie's inferlets encapsulate both inference and external interactions within a single runtime, eliminating these round trips and allowing direct manipulation of the KV cache to retain context across interactions.

We measure the latency and throughput in Figure 6 using a 1B model and representative workloads involving 8 (ReACT), 8 (CodeACT), and 32 (Swarm) external I/Os, respectively, per agent. The observed latencies are 4.27 s, 3.18 s, and 6.14 s, with corresponding throughputs of 29.94, 40.18, and 5.21 agents/s, respectively. Pie significantly outperforms baseline systems, *e.g.*, reducing latency by up to 15% and increasing throughput by up to 30% on ReACT. The performance gains are closely tied to the ratio of I/O to the total number of tokens in the workflow. For instance, we observe no performance difference between baselines when the number of external interactions less than two, and the gap linearly widens as the number of interactions increases.

Two factors contribute to the performance improvements. For smaller models (1B, 3B), the primary factor is the elimination of round-trip latency between the client and server for each external interaction, as the round-trip time (tens of milliseconds) is comparable to token generation time (a few milliseconds). For larger models (8B and above), the dominant factor is the ability to retain the KV cache across external interactions, avoiding costly re-prefills required by the baselines.

7.2 Customizing generation strategies

New LLM generation strategies enhance reasoning, output quality, and efficiency but often require modifications to underlying serving systems to be efficient or even feasible. Pie's programmability addresses this challenge, enabling users to implement and deploy novel strategies as inferlets without modifying the core system.

Applying novel optimizations. Pie creates new opportunities for application-level inference optimizations. Consider a typical agentic workflow that initiates multiple LLM function calls to external APIs based on the user query and the provided set of available APIs. Let us assume that this application uniquely exhibits the following characteristics:

1. Certain APIs are used more frequently than others.
2. The majority of APIs are designed to be fire-and-forget.
3. Most APIs within a set are invoked only once.

This application-specific knowledge cannot be explicitly leveraged by existing serving systems, which typically rely on implicit optimizations like automatic prefix caching in vLLM. However, a Pie user can optimize this workflow by employing the following techniques, each exploiting an aforementioned characteristic:

#1. Retain the KV cache of frequently used API documentation using `export_kvpage`.
#2. Concurrently call APIs immediately upon detecting the callable signature in the generated tokens.
#3. Drop the KV cache of API specifications used only once from the context using `mask_kvpage`.

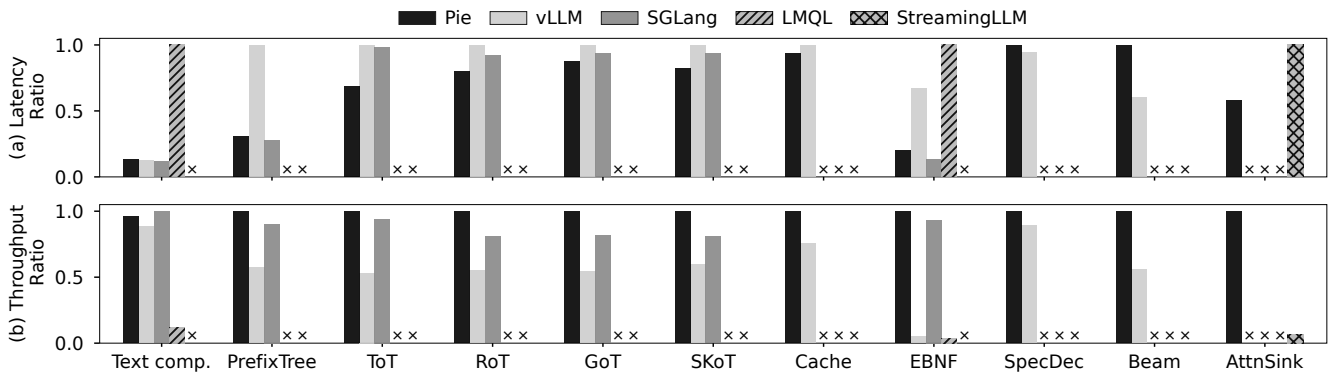

Figure 8. Latency (lower the better) and **throughput** (higher the better) of example LLM inference techniques hosted by different serving systems. Numbers are normalized the same way as in Figure 6. Pie matches state-of-the-art serving systems in performance. × indicates unsupported techniques or impossible comparisons. For example, SGLang lacks support for the specific speculative decoding implementation [62] used by Pie and vLLM.

As Figure 7 shows, each successive optimization builds upon the last, culminating in a 3.5× throughput increase over the baseline Python-implemented workflow on vLLM. This highlights how Pie's fine-grained control over KV cache (R1), generation process (R2), and I/O (R3) enables developers to tailor inference strategies to application semantics, yielding substantial performance benefits beyond what generic system features like prefix caching can offer alone.

Deliberate prompting strategies. We implement four deliberate prompting strategies as inferlets: Tree-of-Thought (ToT), Recursion-of-Thought (RoT), Graph-of-Thought (GoT), and Skeleton-of-Thought (SkoT), each comprising approximately 80-100 lines of code (see Table 2). In our implementation, we use a simplified version of the tasks described in the original papers (*e.g.*, arithmetic tasks for ToT and RoT, and document summarization for GoT). For RoT, we allow recursive branching up to a depth of 5, resulting in a maximum of 2^5=32 branches. Pie achieves better performance across all four strategies, with latency reductions of up to 28% and throughput improvements of up to 34% compared to baseline systems. The performance advantage arises from Pie's program-controlled KV cache reuse, which offers more precise control and flexibility than the implicit KV cache management in existing systems. This proves beneficial for complex strategies like Recursion-of-Thought (RoT), where SGLang's RadixAttention cannot be effectively applied due to the dynamic nature of the graph structure. Furthermore, Pie's integrated I/O capabilities provide additional performance benefits for strategies like Tree-of-Thought (ToT), where the system can efficiently interleave compute during value evaluation phases, such as the symbolic evaluation to trim the search space.

Attention-level techniques. Leveraging the `mask_kvpages` and mask argument in the `forward` API, Pie can implement various attention-level techniques, including attention sink, windowed attention, and hierarchical attention. To the best of our knowledge, these techniques have not been previously implemented in vLLM or SGLang. We thus compare Pie against specialized implementations, *e.g.*, StreamingLLM for attention sink [74]. Compared to the original StreamingLLM implementation, Pie demonstrates significant performance improvements, achieving 1.5× lower latency and over 30× higher throughput. While this comparison is influenced by differences in the underlying GPU kernel libraries, it highlights Pie's ability to effectively express and implement attention-level techniques as inferlets.

7.3 Replicating existing serving features

A key aspect of Pie is its ability to programmatically replicate optimizations often implemented as monolithic, system-wide features in other LLM serving systems. This allows such optimizations to be composable and customized per application. We implement several such features as inferlets.

For prefix caching, we replicate vLLM's mechanism using `export_kvpage` and `import_kvpage` to enable application-controlled cache sharing. For speculative decoding, we implement vLLM's n-gram prompt-lookup method [62]. We also implement beam search. Additionally, we create prefix trees (RadixAttention equivalent) to support SGLang-style branching generation by managing shared KV prefixes and divergent streams. Finally, we implement structured generation by integrating a Rust-based constrained decoding library (`llguidance`) via Wasm to constrain token sampling at each step. We highlight the EBNF implementation as an example of Pie's ability to integrate third-party libraries seamlessly.

As shown in Figure 8, Pie generally achieves comparable performance to vLLM and SGLang on these tasks. For beam search using three beams, Pie exhibits slightly higher latency, but achieves better throughput. For EBNF decoding using JSON grammar, Pie matches SGLang's performance

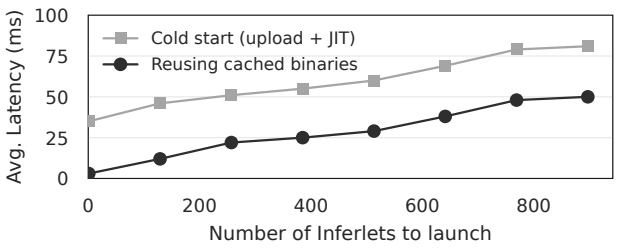

Figure 9. Average latency to launch an inferlet. The latency is insignificant compared to per-token generation latency. Caching JIT-ed binaries helps reduce latency.

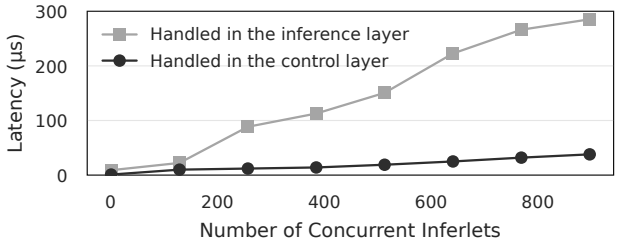

Figure 10. Latency of API calls when handled by the control or inference layer. Higher latency in the inference layer is attributed to the IPC boundary crossing and Python-side API call deserialization overhead.

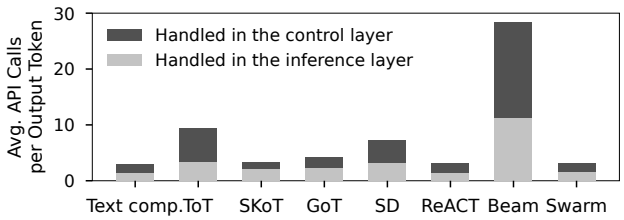

Figure 11. Average number of API calls per task normalized by the number of generated tokens. Beam search (beam=5) involve significantly more API calls because only the tokens from the winning beam are counted as output.

and significantly outperforms vLLM and LMQL. These results demonstrate that Pie's programming model is both expressive and efficient, enabling the implementation of state-of-the-art optimizations at the application level.

7.4 System overhead analysis

We employ several microbenchmarks to analyze the performance impact and overhead introduced by Pie's programming model and system architecture.

Time to launch new inferlets. Pie is designed to serve multiple concurrent inferlets, each running in its own Wasm runtime instance. The number of concurrent inferlets can grow to several hundred. This might raise concerns about the overhead associated with launching a large number of inferlets. To investigate this, we measure the end-to-end time required to launch an inferlet for the text completion task (in Table 2). Specifically, we modify the `text_completion` inferlet to send an acknowledgement message to the user before initiating token generation. We then measure the time elapsed between the request to launch an inferlet and the reception of acknowledgement, from the Python client. We measure the latency under two scenarios: cold start and warm start. In a cold start, the client uploads the Wasm binary before launching the inferlet, and Pie JIT-compiles the received binary. In a warm start, the client utilizes the cached binary in Pie. As shown in Figure 9, the launch overhead ranges from 10 ms to 50 ms for a warm start and 35 ms to 81 ms for a cold start when up to 896 inferlets request launching at the same time. This overhead is insignificant given that the typical per-token generation latency ranges from 10 ms to 60 ms. Pie achieves the low startup latency through the use of wasmtime's pooled allocation feature [72], which preallocates virtual memory for the Wasm runtime for up to 1,000 concurrent instances.

Overhead per API call in inferlets. We examine the overhead of API calls since inferlets can trigger hundreds to thousands of calls per task, potentially creating performance bottlenecks. API calls fall into two categories: (1) handled by the control layer (*e.g.*, `get_arg`, `send`) and (2) handled by the inference layer (*e.g.*, `forward`, `embed_txt`). Figure 11 shows the frequency of API calls for various tasks, normalized by the number of output tokens generated. For basic text completion, each output token corresponds to approximately 1.6 calls handled by the inference layer and 1.5 by the control layer. A token generated by more complex operations like beam search (width=3) corresponds to significantly more—about 17 and 13 calls handled by the inference and control layer, respectively. We measure the overhead of each API call in microseconds, as shown in Figure 10. This overhead is the time from issuing an API call to its completion, excluding handling time (*e.g.*, GPU execution or controller processing, which is typically 6ms-50ms per token on 8B models). We disable batch scheduling for this measurement. The measured latency represents an upper bound since most API calls are executed asynchronously, allowing some latency to be hidden as they do not wait for the previous call to complete. API calls handled in the control layer are inexpensive, costing less than 30 microseconds per call, even with 896 concurrent inferlets. In contrast, those handled in the inference layer are costlier, ranging from 10 to 300 microseconds per call, depending on the number of concurrent inferlets. The main overhead arises from Python's single-threaded deserialization of API calls, which becomes noticeable with more concurrent inferlets. IPC boundary crossing adds minor, constant latency. A native multi-threaded implementation of inference layer (*e.g.*, in C++/Rust) could mitigate this.

Table 3. Opportunity cost of Pie's programming model. Compared to vLLM, the overhead is small compared to Pie's per-token latency (17 ms to 65 ms).

Component	Latency
Text completion TPOT (vLLM)	64.06 ms
Lack of pipelined sampling on GPU	+1.320 ms
Lack of pipelined input embedding on GPU	+0.070 ms
Overhead of control layer batch scheduling	+0.050 ms
Overhead of returning output distribution	+0.070 ms
Boundary crossing (control-inference layer)	+0.006 ms
Boundary crossing (application-control layer)	+0.001 ms
Wasm processing overhead	+0.001 ms
Text completion TPOT (Pie)	65.59 ms

Table 4. Generation time per output token (TPOT) for text completion tasks across different model sizes. Larger model sizes help amortize the overhead of Pie.

Param. size	vLLM	Pie	Overhead (%)
8B	64.06 ms	65.59 ms	1.53 ms (2.39%)
3B	30.30 ms	32.01 ms	1.71 ms (5.64%)
1B	16.83 ms	18.75 ms	1.92 ms (11.41%)

Table 5. Throughput across batching strategies. The adaptive policy, using work-conserving scheduling, achieves the highest throughput compared to no batching (Eager) and the fixed-rate policies (K-only, T-only).

	Eager	K-only	T-only	Adaptive
Requests/s	5.61	30.09	78.11	84.85

Opportunity cost of the programming model. The fine-grained APIs in Pie create an opportunity cost when compared to monolithic systems like vLLM. A monolithic decoding loop, such as vLLM's, offers three primary advantages: (1) pipelining the embedding of input tokens and the sampling of output tokens with the LLM forward pass, (2) eliminating the need for separate management of output token distributions, and (3) simplifying batch scheduling. To measure this cost, we conducted an ablation study on Pie's programming model. We created less granular, custom APIs (*e.g.*, `forward_with_sampling`) that emulate the fused operations of a monolithic system and measured the resulting performance gains. The results from 32 concurrent inferlets, summarized in Table 3, indicate that the opportunity costs associated with advantages (2) and (3) are relatively minor. In contrast, the lack of pipelining—advantage (1)—imposes a more significant latency overhead, ranging from 0.23 to 1.39 ms per token. Despite these costs, the overhead remains small compared to Pie's typical per-token latency of 5 to 60 ms. This highlights a trade-off: for highly latency-sensitive applications, introducing specialized APIs could boost performance, albeit at the cost of reduced composability.

Impact of model size. The relative overhead of Pie diminishes as model size increases, as the system's overhead is amortized over the longer computation time. As shown in Table 4, which compares the end-to-end time per output token (TPOT) with vLLM, the performance gap between the two systems narrows for larger models. For instance, with an 8B parameter model, the 1.53 ms overhead from Pie constitutes just 2.4% of vLLM's TPOT.

Batching strategies. To demonstrate the advantage of adaptive batch scheduling (§6.1), we compared its throughput against three baseline strategies: no batching (Eager), fixed-size batching based on queue length (K-only), and timeout-based batching based on wait time (T-only). Under a fully saturated scheduler with 128 concurrent inferlets, the adaptive strategy proved superior. As shown in Table 5, it delivered up to 17× higher throughput than the Eager baseline and outperformed the K-only and T-only strategies by 8–40%.

8 Discussion and Limitations

Security implications. Delegating control to user inferlets and enabling network I/O expand the attack surface relative to closed-loop serving. Risks include side-channel leakage, resource-exhaustion (DoS), and misuse of access to token distributions (*e.g.*, aiding jailbreaks or model extraction). A comprehensive security treatment is beyond this paper's scope, but hardening is essential for commercial adoption of Pie. Promising directions include capability-based I/O, per-inferlet rate limits and quotas, and limiting or obfuscating exposure of logits (*e.g.*, top-K caps).

Runtime implementation choice. We chose WebAssembly (Wasm) as the inferlet runtime to balance isolation, startup latency, and portability for multi-tenant workloads. Alternatives fared worse for our goals: native OS processes [22] provide weaker containment; containers (*e.g.*, Docker) add cold-start and resource overhead for frequent inferlet instantiation; embedded interpreters (*e.g.*, Lua) often incur FFI overhead and provide weaker isolation. In practice, Wasm offered the best trade-off for Pie.

Scalability of the control layer. Today, Pie 's control layer is a centralized batch scheduler feeding a single inference backend. Real deployments must span multiple GPU nodes, raising challenges in load balancing, KvPage locality (placement/migration), and end-to-end SLO enforcement. Scaling Pie thus requires distributed coordination, globally-aware scheduling, and cache-aware placement policies, as well as fault tolerance across nodes.

Retrofitting programmability into existing systems. Supporting fine-grained programmability entails three architectural changes: (1) integrating a runtime to execute user

code safely, (2) designing low-level APIs for control and resource management, and (3) redesigning batch scheduling around API-level GPU operations. Retrofitting these into current serving stacks would amount to Pie's architecture. That said, more incremental, coarser-grained programmability layers that better fit existing designs remain an interesting avenue for future work.

Handling resource contention. When demand for KvPage exceeds capacity, Pie currently applies an FCFS policy that terminates the most recently started inferlet to free resources. Richer policies could improve fairness and efficiency: per-inferlet quotas, SLO/priority-aware admission and preemption, and even controlled overprovisioning by swapping KvPage between CPU and GPU memory. Exploring these mechanisms is promising future work.

9 Related Work

Pie builds on a lineage of programmable systems [6, 16, 26, 46, 49, 50, 58] across operating systems, networking, and distributed frameworks, which expose fine-grained interfaces to separate application-level control from system internals—a core principle Pie applies to the domain of LLM serving.

9.1 Application-aware LLM serving

Pie is in line with the recent trend towards holistic serving system optimizations for LLM workflows [1, 2, 30, 48, 60]. Notably, recent systems like SGLang [84] and Parrot [44] have introduced programmatic control constructs. SGLang uses primitives like fork/join/gen to optimize shared generation paths via its RadixAttention, while Parrot employs "semantic variables" to improve KV cache reuse based on application context. While offering valuable abstractions for structured generation and prompt management (partially addressing R1 and R2), they are still fundamentally constrained by a monolithic generation process. Consequently, they lack fine-grained, direct control over low-level resources (*e.g.*, individual KV cache pages), the sequence of inference steps (*e.g.*, embedding, attention, sampling), or seamless integration of arbitrary I/O within the generation process —limiting their ability to meet R3. In contrast, Pie provides lower-level, explicit control via its API, enabling customization beyond what these higher-level abstractions permit. Orthogonally, several works improve the interaction efficiency between LLMs and external tools to address R3—for example, by parallelizing LLM function calls [33, 54] or managing the KV cache while awaiting call completion [20, 80]. Pie supports such policies as user-defined programs, eliminating the need for internal changes to the serving system.

9.2 Programming LLM behavior

Pie shares conceptual similarities with recent works on "LLM programming", which has emerged along two main directions: (1) constrained decoding using formal grammars, and (2) prompt engineering and orchestration frameworks. In the first direction, there exists LMQL [8], Guidance [24], Outlines [14], XGrammar [13], and AICI [51]. These systems allow developers to control generation semantics—typically via EBNF grammars or declarative constraints—within the confines of the standard decoding loop. Notably, AICI uses WebAssembly to allow more flexibility. In contrast, Pie extends this idea by using a Turing-complete language not only to specify what the model should generate, but also *how* generation is performed. That is, Pie *controls the inference process itself*—including tokenization, attention, KV cache management, sampling, and I/O—step by step. This provides a level of programmability that goes beyond output semantics. In the second direction, frameworks such as LangChain [39], LangFlow [40], DSPy [32], and AutoGen [73] simplify LLM agent workflows, and prompt optimization, with a focus on developer usability and output quality. Pie is orthogonal and complementary to these systems. It can serve as a backend by running inferlets that either manually embed their logic or are generated from their abstractions. This allows such frameworks to inherit Pie's performance and I/O integration.

9.3 Efficient Transformer inference

A growing body of work has improved Transformer inference through quantization [45, 69], parallelism [3, 83], GPU kernel optimization [12, 78], and memory management [42, 59, 77]. Pie complements these efforts by enabling a programmable LLM generation process that builds on top of such efficiency-oriented techniques.

10 Conclusion

Pie represents a paradigm shift in LLM serving towards programmability. Moving beyond inflexible monolithic designs, Pie decomposes the token generation process into modular API handlers. This allows user-provided programs (inferlets) to orchestrate the entire workflow, enabling fine-grained control over KV cache management, custom decoding procedures, and tight integration of external computation and I/O. We show that this programmability allows Pie to implement a wide array of modern LLM techniques, from custom attention patterns to agentic workflows. Crucially, our evaluation confirms that Pie achieves competitive performance on standard tasks while delivering substantial throughput and latency improvements for complex workloads by facilitating targeted optimizations. Our results demonstrate that programmability is key to the performance and flexibility required by the next generation of LLM applications.

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] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. In *Proc. ICML*.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *Proc. USENIX OSDI*.

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed- Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proc. IEEE/ACM SC Conf.*

[4] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. 2025. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923* (2025).

[5] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *CoRR* abs/2004.05150 (2020).

[6] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan J Eggers. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM SOSP*.

[7] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. In *Proc. AAAI*.

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

[9] Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. In *Proc. ICML*.

[10] Bytecode Alliance. 2025. bytecodealliance/wasmtime: A lightweight WebAssembly runtime that is fast, secure, and standards-compliant. https://github.com/bytecodealliance/wasmtime. Accessed: 2025-08-26.

[11] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069* (2024).

[12] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Proc. NeurIPS*.

[13] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. *CoRR* abs/2411.15100 (2024).

[14] Dottxt-AI. 2025. Outlines: Structured Text Generation. https://github.com/dottxt-ai/outlines. Accessed: 2025-04-16.

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024).

[16] Dawson R Engler, M Frans Kaashoek, and James W O'toole Jr. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. ACM SOSP*.

[17] Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. 2024. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *arXiv preprint arXiv:2407.11550* (2024).

[18] Markus Freitag and Yaser Al-Onaizan. 2017. Beam Search Strategies for Neural Machine Translation. *ACL* (2017), 56.

[19] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *Proc. USENIX OSDI*.

[20] Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. Fast State Restoration in LLM Serving with HCache. In *Proc. EuroSys*.

[21] 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 *Proc. MLSys*.

[22] In Gim and Lin Zhong. 2025. Serve Programs, Not Prompts. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*. 179–186.

[23] Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. 2024. Better & Faster Large Language Models via Multi-token Prediction. In *Proc. ICML*.

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

[25] Neha Gupta, Harikrishna Narasimhan, Wittawat Jitkrittum, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. 2024. Language Model Cascades: Token-Level Uncertainty And Beyond. In *Proc. ICLR*.

[26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. USENIX NSDI*.

[27] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *Proc. ICLR*.

[28] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

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

[30] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. 2025. RAGO: Systematic Performance Optimization for Retrieval-Augmented Generation Serving. abs/2503.14649 (2025).

[31] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-Search-Predict: Composing retrieval and language models for knowledge-intensive NLP. *CoRR* abs/2212.14024 (2022).

[32] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *Proc. ICLR*.

[33] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2023. An LLM Compiler for Parallel Function Calling. abs/2312.04511 (2023).

[34] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A Watermark for Large Language Models. In *Proc. ICML*.

[35] Michael Kuchnik, Virginia Smith, and George Amvrosiadis. 2023. Validating Large Language Models with ReLM. In *Proc. MLSys*.

[36] Aounon Kumar, Chirag Agarwal, Suraj Srinivas, Soheil Feizi, and Hima Lakkaraju. 2023. Certifying LLM Safety against Adversarial Prompting. abs/2309.02705 (2023).

[37] Woosuk Kwon. 2024. [RFC] Drop beam search support. https://github.com/vllm-project/vllm/issues/6226. GitHub issue #6226, vLLM Project.

[38] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. ACM SOSP*.

[39] LangChain-AI. 2025. LangChain: Build context-aware reasoning applications. https://github.com/langchain-ai/langchain. Accessed: 2025-04-16.

[40] Langflow. 2025. Langflow: Low-code AI builder for agentic and RAG applications. https://www.langflow.org/. Accessed: 2025-04-16.

[41] Soochan Lee and Gunhee Kim. 2023. Recursion of Thought: A Divide-and-Conquer Approach to Multi-Context Reasoning with Language Models. In *Proc. ACL*.

[42] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *Proc. USENIX OSDI*.

[43] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *Proc. ICML*.

[44] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *Proc. USENIX OSDI*.

[45] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Proc. MLSys*.

[46] Linux Foundation. 2024. eBPF - Extended Berkeley Packet Filter. https://ebpf.io/. Accessed: 2025-04-17.

[47] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proc. ACM SIGCOMM*.

[48] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, and Ion Stoica. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. abs/2502.13965 (2025).

[49] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* (2008).

[50] Robert Morris, Eddie Kohler, John Jannotti, and M Frans Kaashoek. 1999. The Click modular router. In *Proc. ACM SOSP*.

[51] Michal Moskal, Madan Musuvathi, and Emre Kıcıman. 2024. AI Controller Interface. https://github.com/microsoft/aici/.

[52] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proc. ICML*.

[53] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. 2024. Skeleton-of-Thought: Prompting LLMs for Efficient Parallel Generation. In *Proc. ICLR*.

[54] OpenAI. 2023. Parallel Function Calling. https://platform.openai.com/docs/guides/function-calling Accessed: 2024-10-26.

[55] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. In *Proc. NeurIPS*.

[56] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *Proc. NIPS Wrkshp. Autodiff*.

[57] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large Language Model Connected with Massive APIs. In *Proc. NeurIPS*.

[58] Simon Peter, Jialin Li, Irene Zhang, Dan R K Ports, Doug Woos, Arvind Krishnamurthy, Thomas E Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*.

[59] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proc. ACM ASPLOS*.

[60] Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Alexander Levis, and Matei Zaharia. 2024. ALTO: An Efficient Network Orchestrator for Compound AI Systems. In *Proc. ACM EuroMLSys at EuroSys*.

[61] Andrea Santilli, Silvio Severino, Emilian Postolache, Valentino Maiorca, Michele Mancusi, Riccardo Marin, and Emanuele Rodolà. 2023. Accelerating Transformer Inference for Translation via Parallel Decoding. In *Proc. ACL*.

[62] Apoorv Saxena. 2023. Prompt Lookup Decoding. https://github.com/apoorvumang/prompt-lookup-decoding/

[63] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Proc. NeurIPS*.

[64] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proc. ICML*.

[65] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. In *Proc. IEEE ICCV*.

[66] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zhelin Zhu, and Bin Luo. 2022. AST-trans: Code summarization with efficient tree-structured attention. In *Proc. ACM/IEEE ICSE*.

[67] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[68] Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. (2024).

[69] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. BitNet: Scaling 1-bit Transformers for Large Language Models. *CoRR* abs/2310.11453 (2023).

[70] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* (2024).

[71] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *Proc. ICML*.

[72] Wasmtime Project. 2025. *PoolingAllocationConfig — Wasmtime API Documentation*. Wasmtime Documentation. https://docs.wasmtime.dev/api/wasmtime/struct.PoolingAllocationConfig.html Accessed: 2025-08-26.

[73] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. abs/2308.08155 (2023).

[74] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *Proc. ICLR*.

[75] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Proc. NeurIPS*.

[76] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proc. ICLR*.

[77] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Proc. ACL*.

[78] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. *CoRR* abs/2501.01005 (2025).

[79] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proc. USENIX OSDI*.

[80] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proc. EuroSys*.

[81] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

[82] ZeroMQ Community. 2023. libzmq: ZeroMQ core engine in C++. https://github.com/zeromq/libzmq.

[83] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proc. USENIX OSDI*.

[84] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, Clark W Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proc. NeurIPS*.

[85] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: Language agents as optimizable graphs. In *Proc. ICML*.