Pie: A Programmable Serving System for Emerging LLM Applications

In Gim, Zhiyao Ma, Seung-seob Lee, Lin Zhong
Yale University

SOSP' 25

Shen Fu, Zewen Jin @ ADSL Reading Group, Nov. 18, 2025

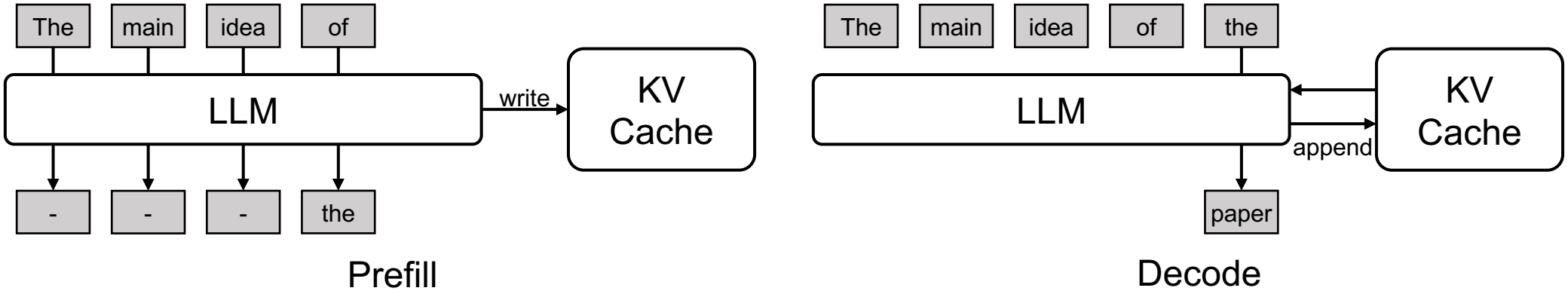
Outline

- **Background**
  - Existing LLM Serving Systems
  - Case Studies
  - Motivation
- **Design**
- **Evaluation**
- **Discussion**

Existing LLM Serving Systems

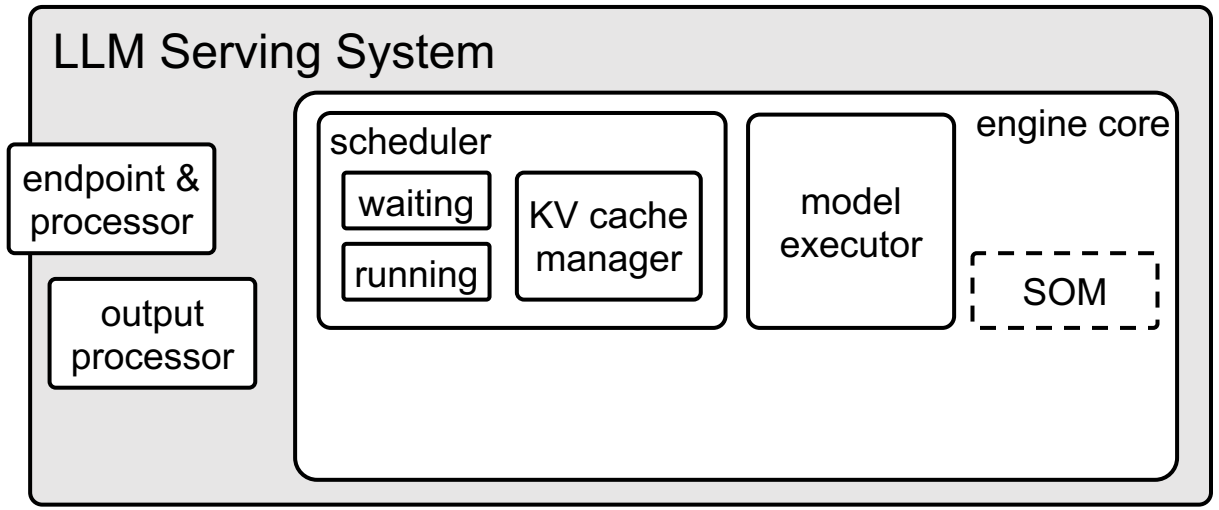
❑LLM Inference

- ❖Generate tokens **autoregressively**
- ❖**KV Cache** stores the key/value vectors of previous tokens
  - ➢Modern serving system pre-allocates device memory as KV cache pool and organize KV cache into virtual blocks



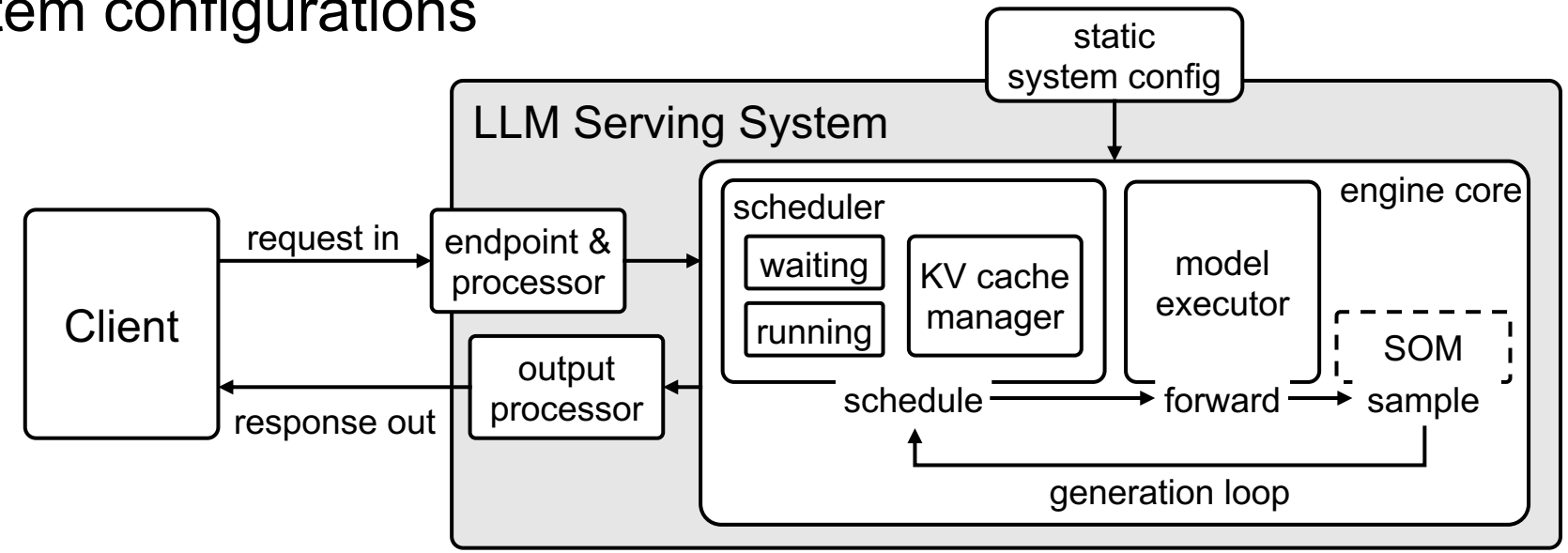
Existing LLM Serving Systems

- **LLM Serving Systems: vLLM, SGLang, …**
  - **Architecture (vLLM V0)**
    - **Scheduler**: decide which requests go into next iteration
    - **Model Executor**: run forward passes
    - **Structured Output Manager**: handles constrained decoding



Existing LLM Serving Systems

- **LLM Serving Systems: vLLM, SGLang, TensorRT, …**
  - **Architecture**
  - **Generation:** monolithic, schedule – forward – sample loop
  - **KV Cache Management:** system-wide, static policy
  - System behavior is determined by both request parameters and initial system configurations



Case Studies on LLM applications

- **Modern LLM applications and decoding techniques requires flexible generation policy**
  - **Post-processing**
    - Structured Output / Constrained Decoding
  - **Special Decoding Techniques**
    - Speculative Decoding
  - **Customized Resource Management**
    - StreamingLLM / Attention Sink
  - **Agents**
    - Agentic Workflow

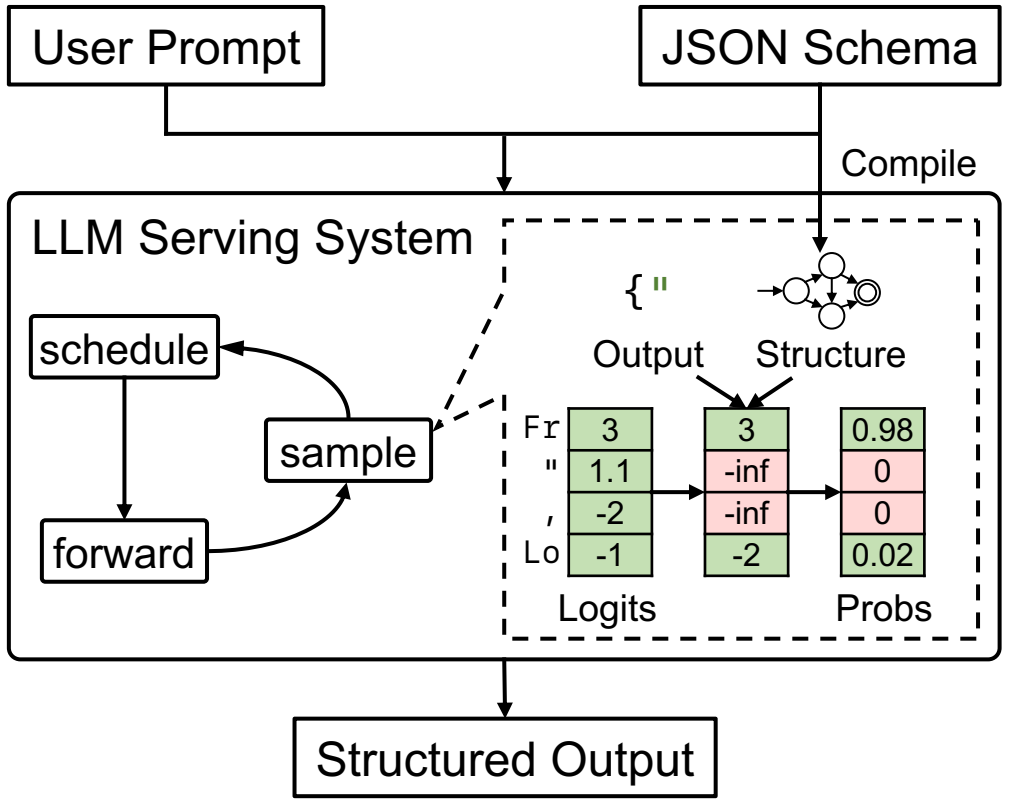
Case Study: Structured Output

❑ **Constrained Decoding:** Enforce the output format of LLM

```
class CapitalInfo:
  name: str = Field(pattern=r"^\w+$")
  latitude: str = Field(pattern=...)
  longitude: str = Field(pattern=...)
```

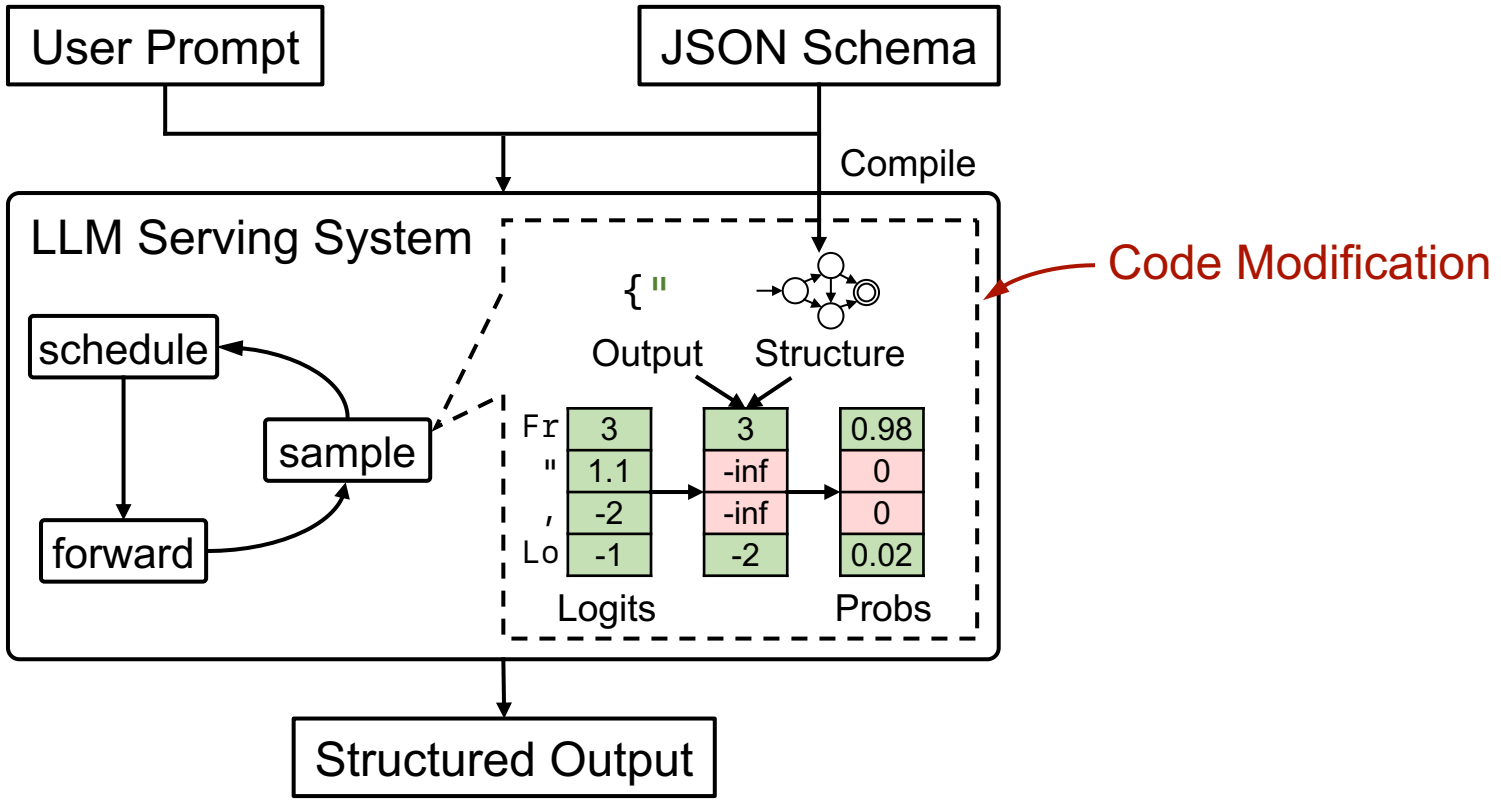
User: Give me the information of the capital of France in the JSON format.

LLM:
```
{
  "name": "France",
  "latitude": "48.86N",
  "longitude": "2.35E",
}
```



Case Study: Post-Processing

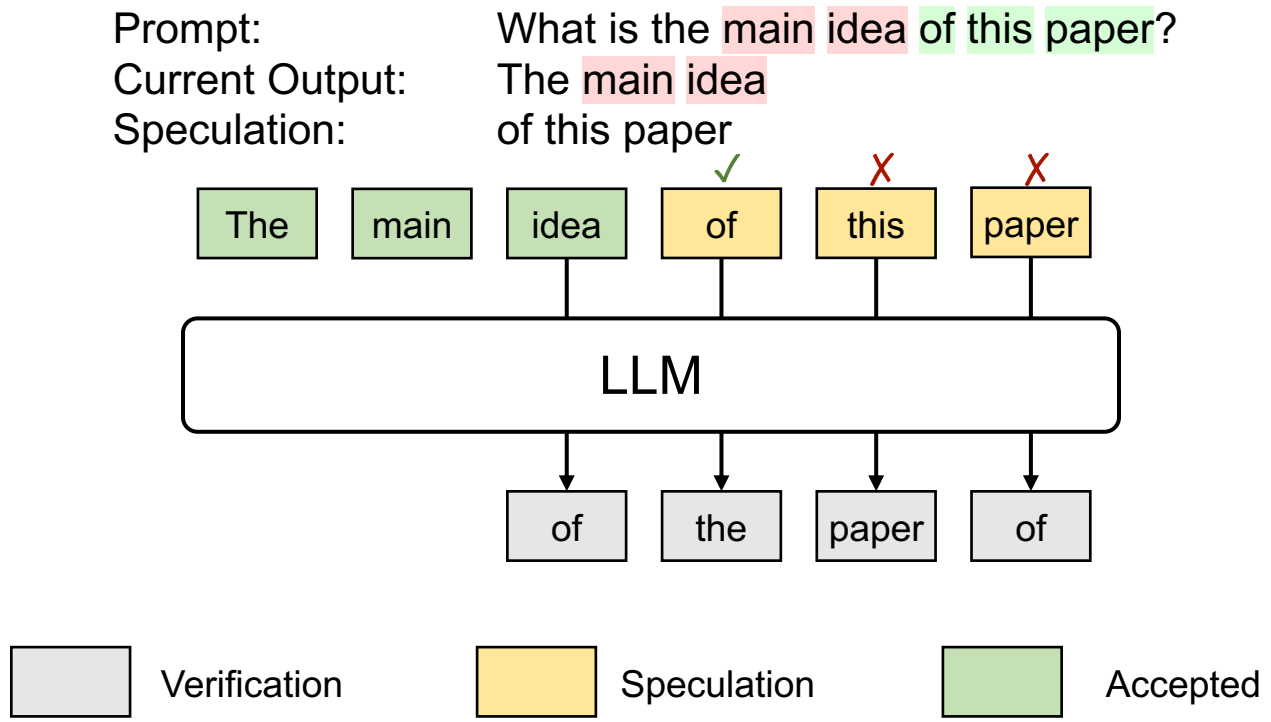
- **Constrained Decoding:** Enforce the output format of LLM
- **In Existing Systems:**
  - ✗ **inject logic** around sampling step



Case Study: Special Decoding Technique

❑*N*-Gram Prompt-Lookup Speculative Decoding

- ❖Copy from input and generated output as draft token (No draft model)
- ❖Usage: code edition[1], summarization, document QA, …

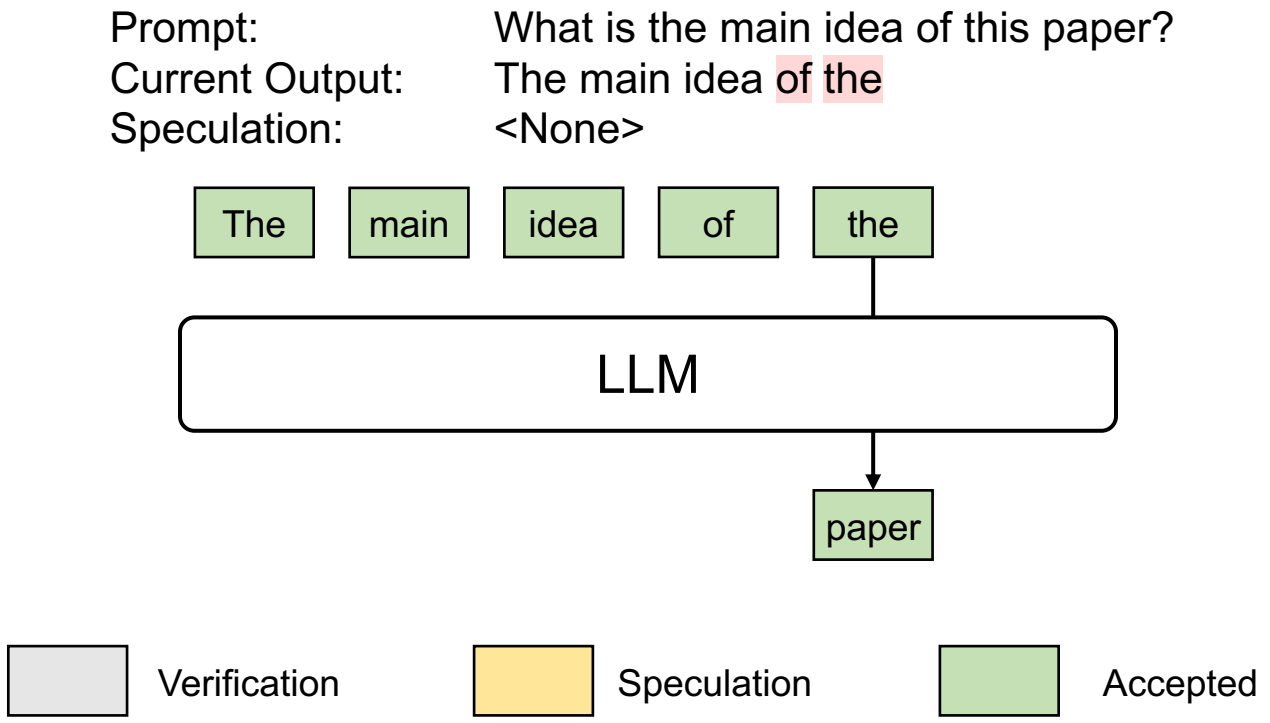


1: https://zed.dev/blog/edit-prediction#minimizing-latency-speculative-decoding.

Case Study: Special Decoding Technique

❑*N*-Gram Prompt-Lookup Speculative Decoding

- ❖Copy from input and generated output as draft token (No draft model)
- ❖Usage: code edition, summarization, document QA, …

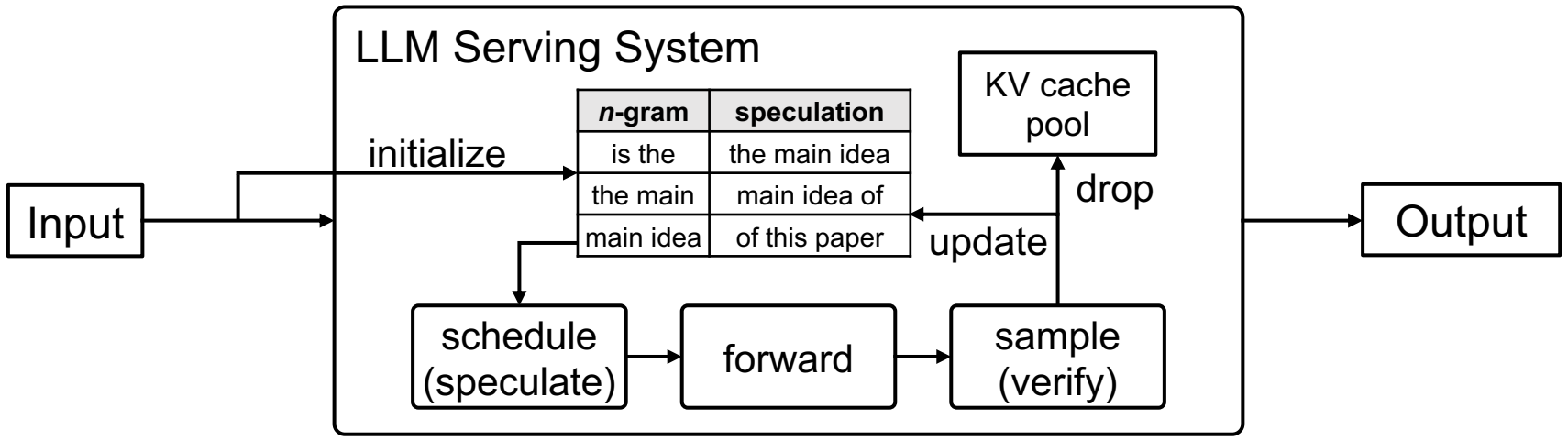


Case Study: Special Decoding Technique

❑*N*-Gram Prompt-Lookup Speculative Decoding

- ❖Copy from input and generated output as draft token (No draft model)
- ❖Usage: code edition, summarization, document QA, …

Prompt: What is the main idea of this paper?
Current Output: The main idea
Speculation: of this paper



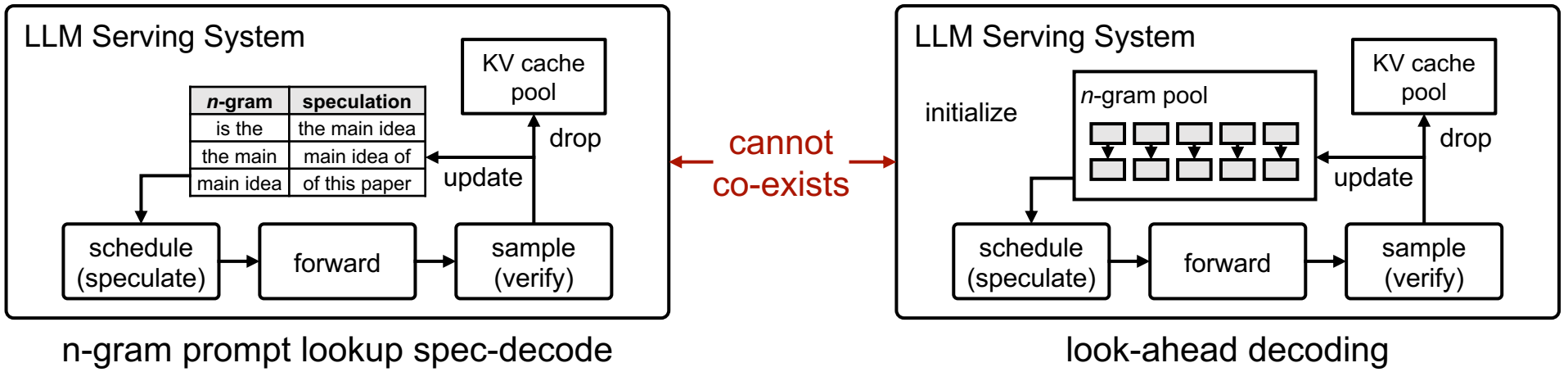
Case Study: Special Decoding Technique

❑Decoding Techniques

- ❖n-gram prompt-lookup speculative decoding, look-ahead decoding, …
- ❖Breaks the schedule-forward-sample loop

❑In existing systems:

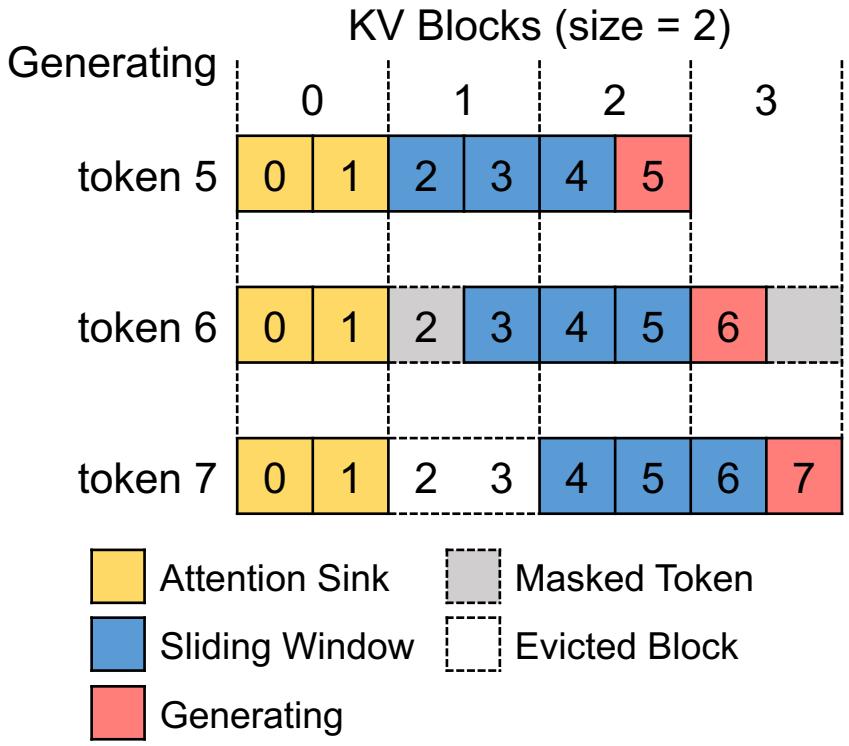
- ❖✗ **Inflexible:** implemented as **system-wide toggles**

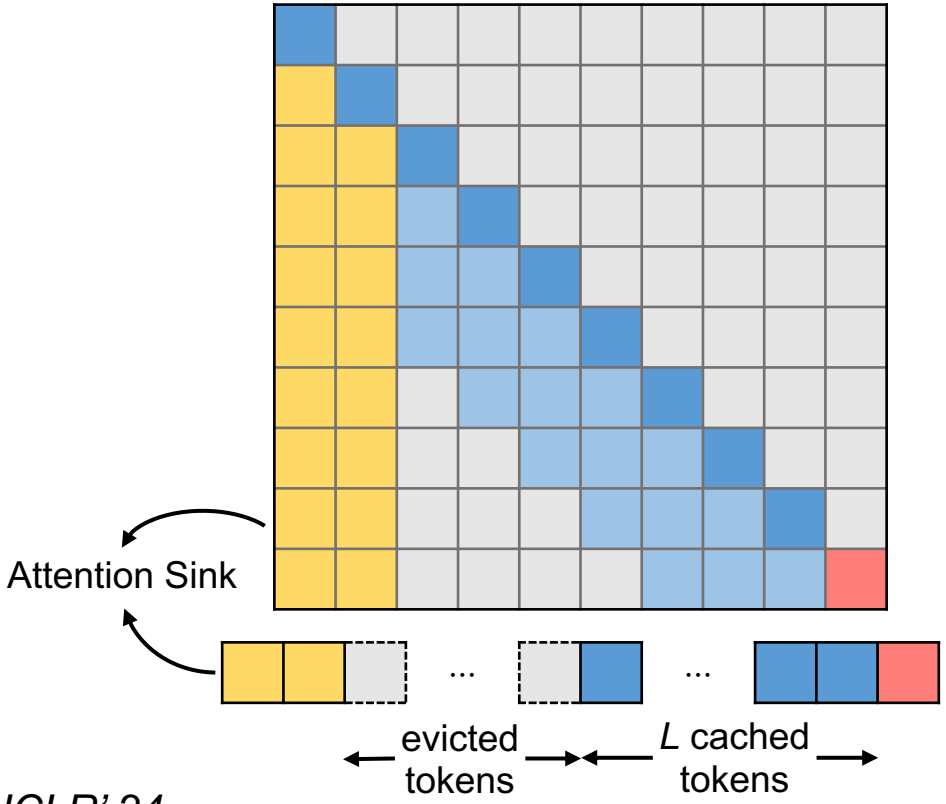


Case Study: Resource Management

❑StreamingLLM[1]: Sliding window w/ Attention Sinks

❖KV cache blocks **masked** (and **evicted**) during the generation process



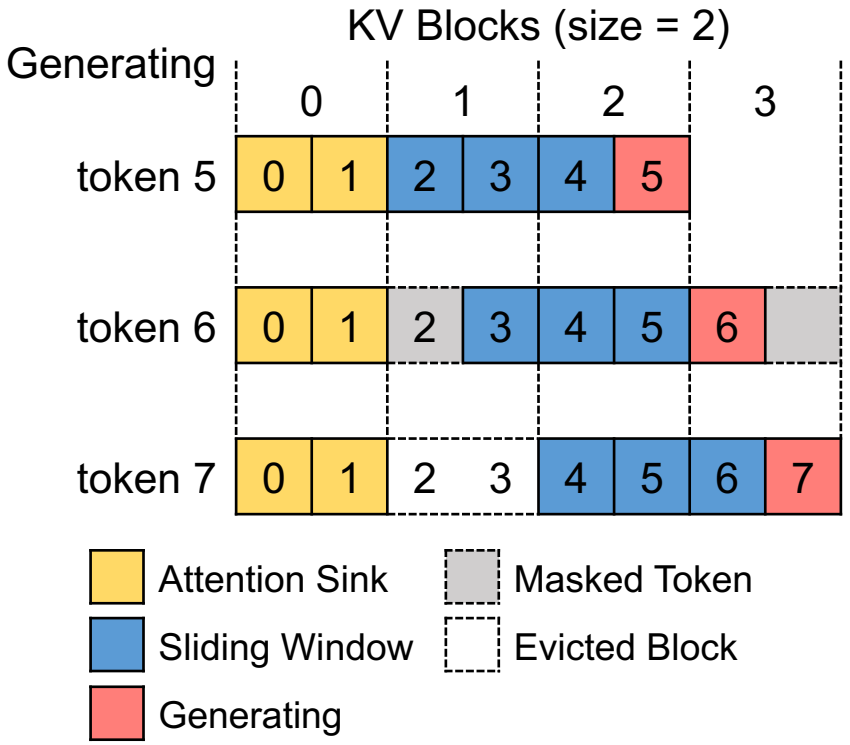


[1] Efficient Streaming Language Models with Attention Sinks. *ICLR' 24*

Case Study: Resource Management

❑StreamingLLM[1]: Sliding window w/ Attention Sinks

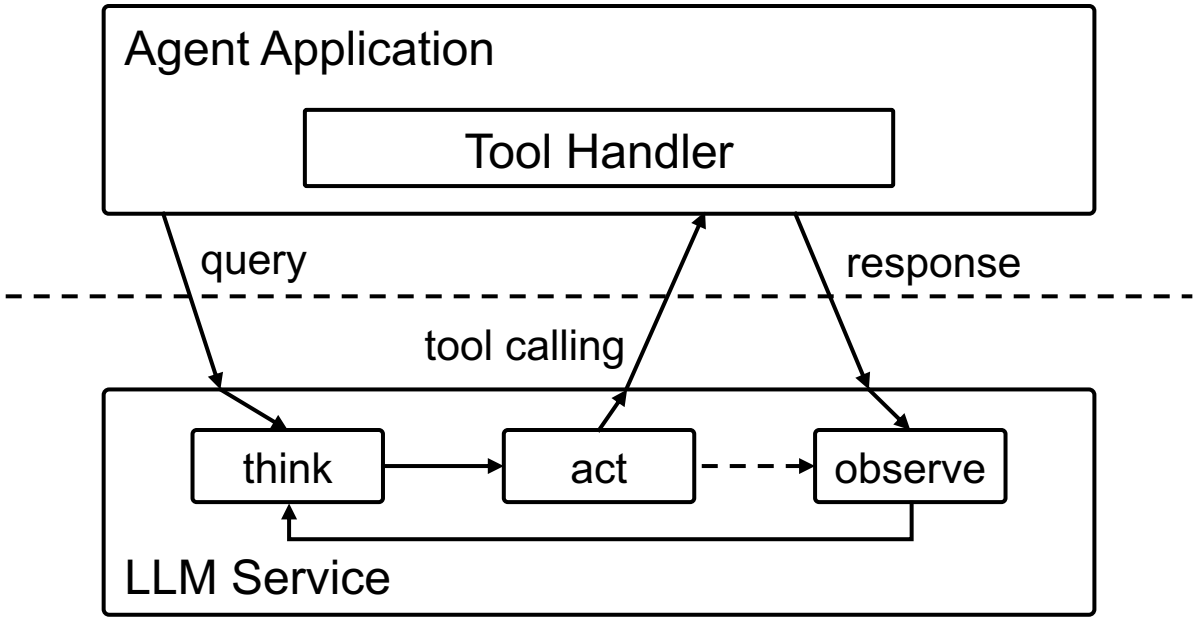
- ❖KV cache blocks **masked** (and **evicted**) during the generation process
- ❖✗ **Invasive** modification **deep within** the memory manager



[1] Efficient Streaming Language Models with Attention Sinks. *ICLR' 24*

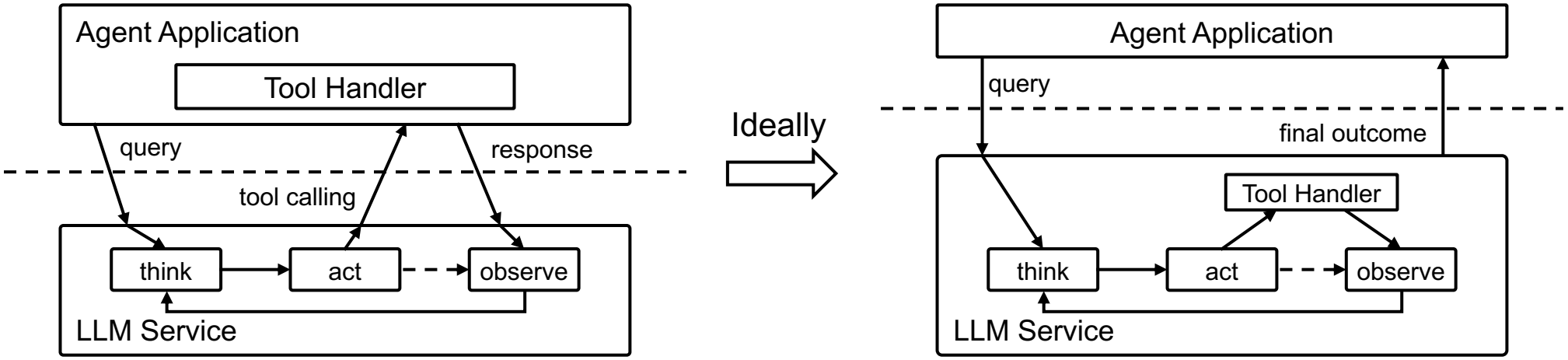
Case Study: Agents

- **Agentic Workflow:** Thought-Action-Observation cycle
  - **Thought:** plan the next step
  - **Action:** take action by calling tool
  - **Observation:** reflect on the response from the tool (*external environment*)



Case Study: Agents

- **Agentic Workflow:** Thought-Action-Observation cycle
  - ✗ **Round-trip control** between client and server
    - Incurs **network latency**
    - Implicit session persistence → possible **costly re-prefill**



Motivation

- **Current systems:**
  - **Inflexible Generation Loop**
    - Logic injection for controlling output distribution
    - Static, system-wide toggles for special decoding technique
  - **Implicit Resource Management**
    - Failed to provide application-specific control
  - **Poor Workflow Integration**
    - Round-trip control switching
- **The monolithic generation loop couples the application control logic and core execution engine.**

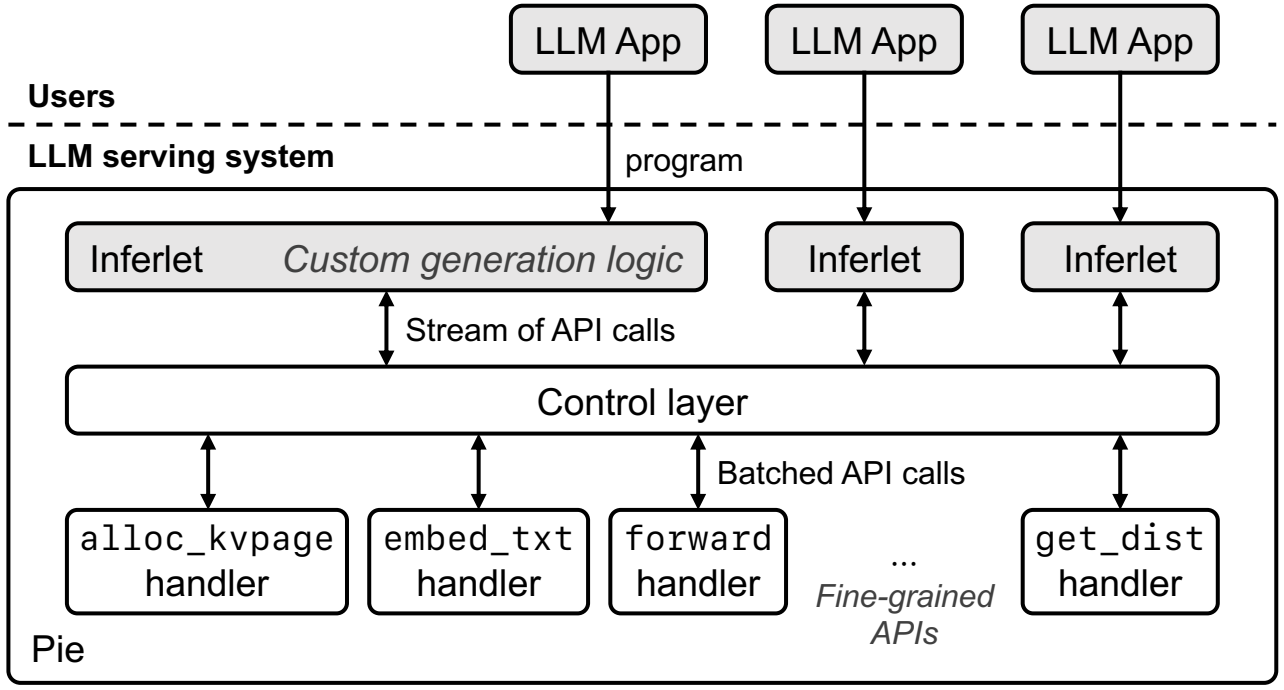
Outline

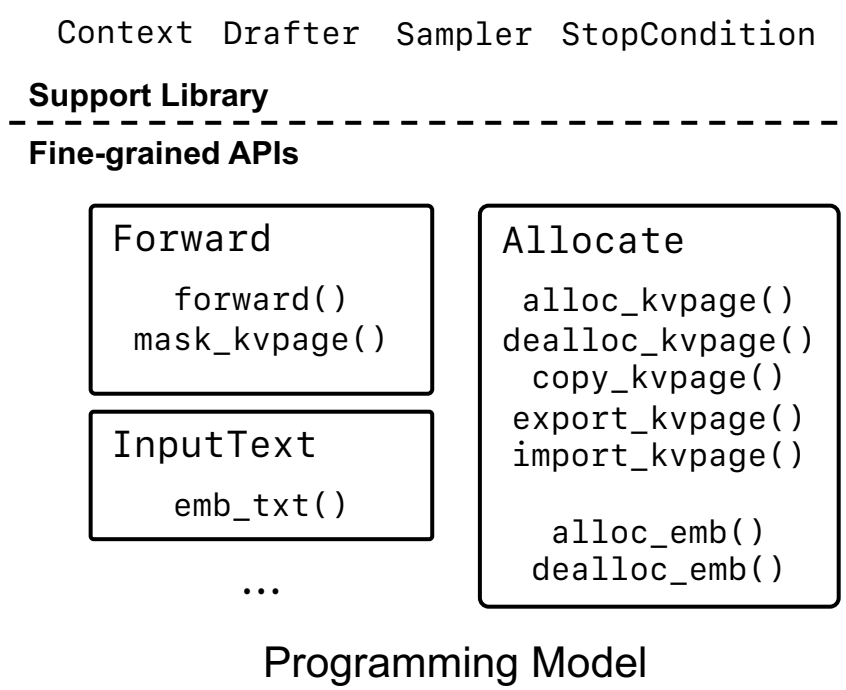
- Background
- **Design**
  - Overview
  - Programming Model
  - System Architecture & Implementation
- Evaluation
- Discussion

Pie – Overview

❑Pie: Serve programs, not prompts

- ❖breaks monolithic generation loop into fine-grained API calls
- ❖Inferlet: user-provided program

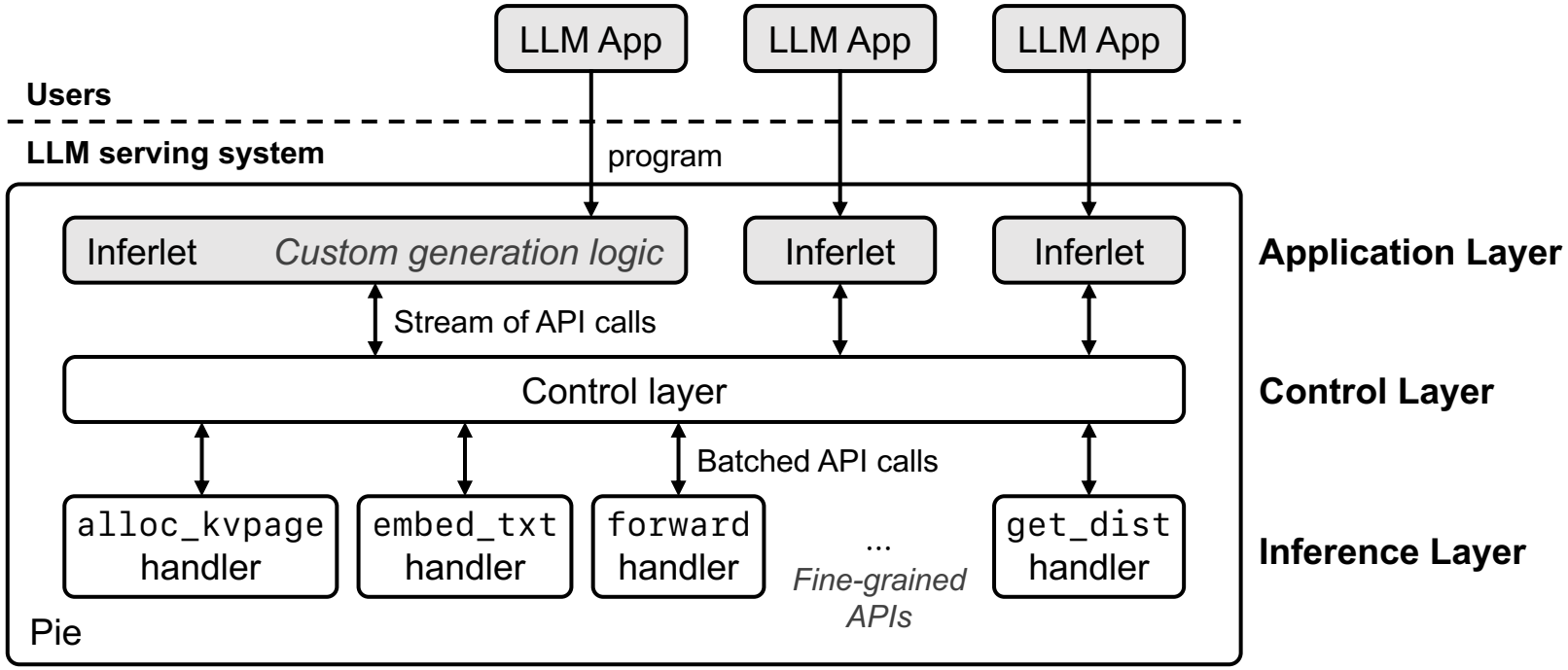




Programming Model

Pie – Overview

- **Pie: Serve programs, not prompts**
  - Schedule: Application Layer + Control Layer
  - Forward: Inference Layer
  - Sample: Application Layer



Pie – Overview

- **Pie**
  - Schedule
    - Application Layer:
      - Define control logic
    - Control Layer
      - Handle non-GPU API call
      - Batch GPU API call
  - Forward: Inference Layer
    - Handle GPU API call
  - Sample: Application Layer
    - Sample probs with customized logic

```
model = get_auto_model()
q = model.create_queue()

ids = tokenize(q, "Hello,")
ilen = len(ids)
limit = ilen + 10

# resource allocation
iemb = alloc_emb(q, ilen)
oemb = alloc_emb(q, 1)
kv = alloc_kvpage(q, limit)

# prefill
pos = len(range(ilen))
embed_txt(q, ids, pos, iemb)
await forward(q, [], iemb, kv[:ilen], oemb)

# decode
for i in range(ilen, limit):
  dist = await get_next_dist(q, oemb)
  ids.extend(dist.max_index())
  embed_txt(q, ids[-1], [i], oemb)
  await forward(q, kv[:i], oemb, kv[i:i+1], oemb)

# output
output = detokenize(q, ids)
print(output)

# resource cleanup
dealloc_emb(q, iemb)
dealloc_emb(q, oemb)
dealloc_kvpage(q, kv)
```

Pie – Overview

- **Pie**
  - **Schedule**
    - **Application Layer:**
      - **Define control logic**
    - Control Layer
      - Handle non-GPU API call
      - Batch GPU API call
  - Forward: Inference Layer
    - Handle GPU API call
  - Sample: Application Layer
    - Sample probs with customized logic

Application Control Logic

```
model = get_auto_model()
q = model.create_queue()

ids = tokenize(q, "Hello,")
ilen = len(ids)
limit = ilen + 10

# resource allocation
iemb = alloc_emb(q, ilen)
oemb = alloc_emb(q, 1)
kv = alloc_kvpage(q, limit)

# prefill
pos = len(range(ilen))
embed_txt(q, ids, pos, iemb)
await forward(q, [], iemb, kv[:ilen], oemb)

# decode
for i in range(ilen, limit):
  dist = await get_next_dist(q, oemb)
  ids.extend(dist.max_index())
  embed_txt(q, ids[-1], [i], oemb)
  await forward(q, kv[:i], oemb, kv[i:i+1], oemb)

# output
output = detokenize(q, ids)
print(output)

# resource cleanup
dealloc_emb(q, iemb)
dealloc_emb(q, oemb)
dealloc_kvpage(q, kv)
```

Pie – Overview

- **Pie**
  - Schedule
    - Application Layer:
      - Define control logic
    - Control Layer
      - Handle non-GPU API call
      - Batch GPU API call
  - Forward: Inference Layer
    - Handle GPU API call
  - **Sample: Application Layer**
    - **Sample probs with customized logic**

Sampling

```
model = get_auto_model()
q = model.create_queue()

ids = tokenize(q, "Hello,")
ilen = len(ids)
limit = ilen + 10

# resource allocation
iemb = alloc_emb(q, ilen)
oemb = alloc_emb(q, 1)
kv = alloc_kvpage(q, limit)

# prefill
pos = len(range(ilen))
embed_txt(q, ids, pos, iemb)
await forward(q, [], iemb, kv[:ilen], oemb)

# decode
for i in range(ilen, limit):
  dist = await get_next_dist(q, oemb)
  ids.extend(dist.max_index())
  embed_txt(q, ids[-1], [i], oemb)
  await forward(q, kv[:i], oemb, kv[i:i+1], oemb)

# output
output = detokenize(q, ids)
print(output)

# resource cleanup
dealloc_emb(q, iemb)
dealloc_emb(q, oemb)
dealloc_kvpage(q, kv)
```

Pie – Overview

❑Pie

❖Customizable generation loop

- ➢ Text generation: schedule – forward – sample
- ➢ SpecDecode: speculate – schedule – forward – verify – drop KV

❖Explicit, fine-grained resource control

- ➢ Attention sink: mask & drop KV cache
- ➢ Graph-of-Thought, Tree-of-Thought: KV cache reusing

❖Workflow Integration

- ➢ Agents: tool handler within inferlet

Pie – Programming Model

❑Pie provides 42 APIs

- ❖18 APIs for LLM execution and resource management
- ❖24 APIs for core runtime operations and agentic workflow

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.

Pie – Programming Model

❑Abstractions: Forward pass

❖A forward pass of LLM: embed – forward – sample

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.

Pie – Programming Model

❑Abstractions: Explicit Resource Managements

- ❖Embeds (per-token) + KvPage (8 – 32 token)
- ❖Each inferlet has its own virtual address space (enable KV sharing)

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.

Pie – Programming Model

❑Abstractions: Command Queue

- ❖A logical sequence of API calls
- ❖Inform control layer with
  - ➢Data dependency
  - ➢Priority

```
model = get_auto_model()
q1, q2 = model.create_queue(), model.create_queue()

forward(q1, [], iemb1, [], oemb1)
forward(q2, [], iemb2, [], oemb1)

await synchronize(q1)
await synchronize(q1)
```

batched API calls

Pie – Programming Model

❑Support Library

❖Provides pre-defined control logic

```
model = get_auto_model()

ctx = model.create_context()

ctx.fill("Hello,")

output = ctx.generate_until(max_token=10)
print(output)

del ctx
```

```
model = get_auto_model()
q = model.create_queue()

ids = tokenize(q, "Hello,")
ilen = len(ids)
limit = ilen + 10

# resource allocation
iemb = alloc_emb(q, ilen)
oemb = alloc_emb(q, 1)
kv = alloc_kvpage(q, limit)

# prefill
pos = len(range(ilen))
embed_txt(q, ids, pos, iemb)
await forward(q, [], iemb, kv[:ilen], oemb)

# decode
for i in range(ilen, limit):
  dist = await get_next_dist(q, oemb)
  ids.extend(dist.max_index())
  embed_txt(q, ids[-1], [i], oemb)
  await forward(q, kv[:i], oemb, kv[i:i+1], oemb)

# output
output = detokenize(q, ids)
print(output)

# resource cleanup
dealloc_emb(q, iemb)
dealloc_emb(q, oemb)
dealloc_kvpage(q, kv)
```

Pie – Programming Model

- **Integration**
  - User-interaction: `send()`, `recv()`
  - Networking: `http_get()`, `http_post()`
  - Inter-inferlet communication: `broadcast()`, `subscribe()`

Pie – Programming Model

❑Integration

❑Extensibility

- ❖Set of related operations/APIs are defined as Traits
  - ➢ Specific model realizes some of (or all of) the traits.
  - ➢ Traits has dependencies
    - `Forward` depends on `Allocate`
  - ➢ Inferlets can query supported traits via `available_traits()`
- ❖Extension (new modal, advanced optimizations) via defining and implementing new traits
  - ➢ Non-invasive: Older traits remains unchanged

Pie – Programming Model

- ❑ **Integration**
- ❑ **Extensibility**
- ❑ **Expressiveness**
  - ❖ Support a wide range of apps and inference techniques
    - ➢ in less than 300 LoC

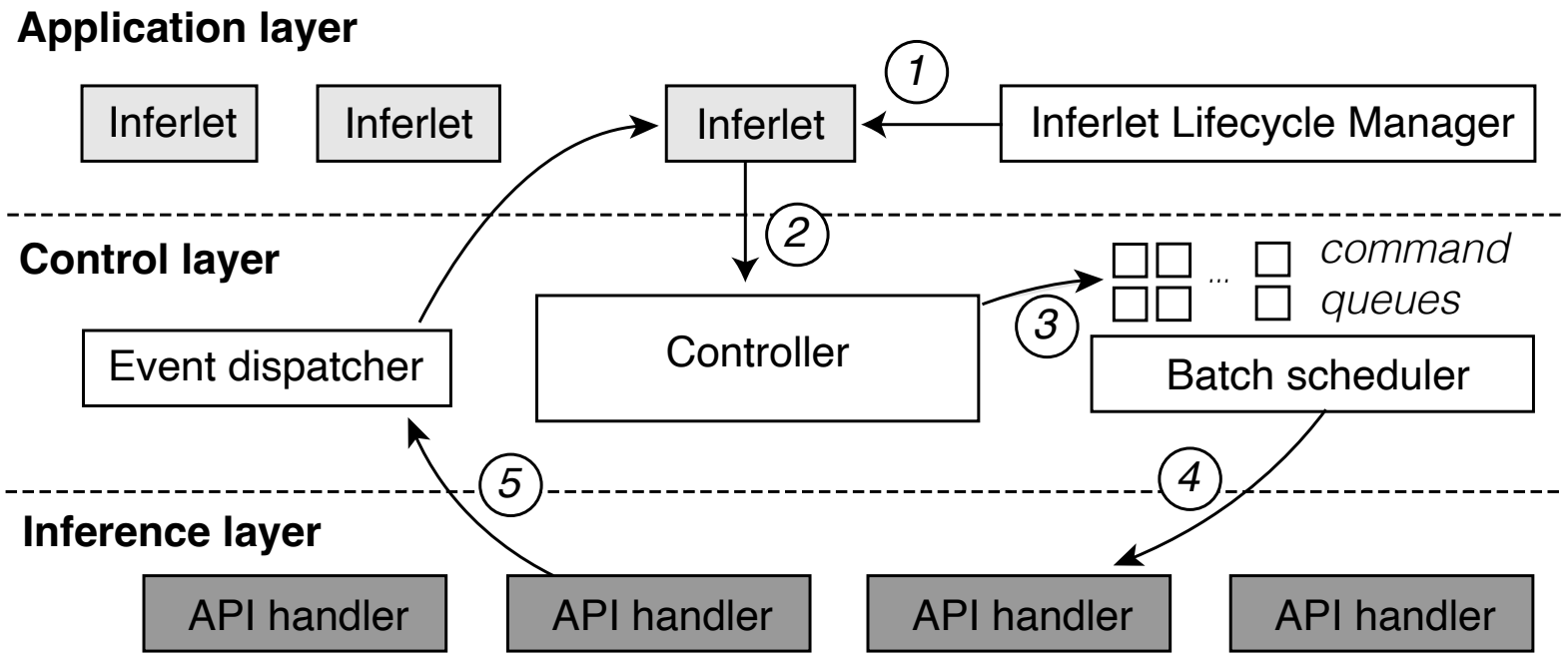
Technique	R1-3 (§1)	LoC	Wasm	Supported
Support library (§6.3)		728	-	
Text completion		38	129 KB	V, S, L
ToT [75]	R1, R3	198	148 KB	S
RoT [41]	R1, R3	106	152 KB	
GoT [7]	R1, R3	87	171 KB	
SKoT [53]	R1, R3	82	173 KB	S
Prefix caching [38]	R1	45	131 KB	V, S
Modular caching [21]	R1	72	139 KB	
EBNF decoding [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	

V: vLLM
S: SGLang
L: LMQL

Pie – System Architecture

❑Pie: Three-layer Architecture

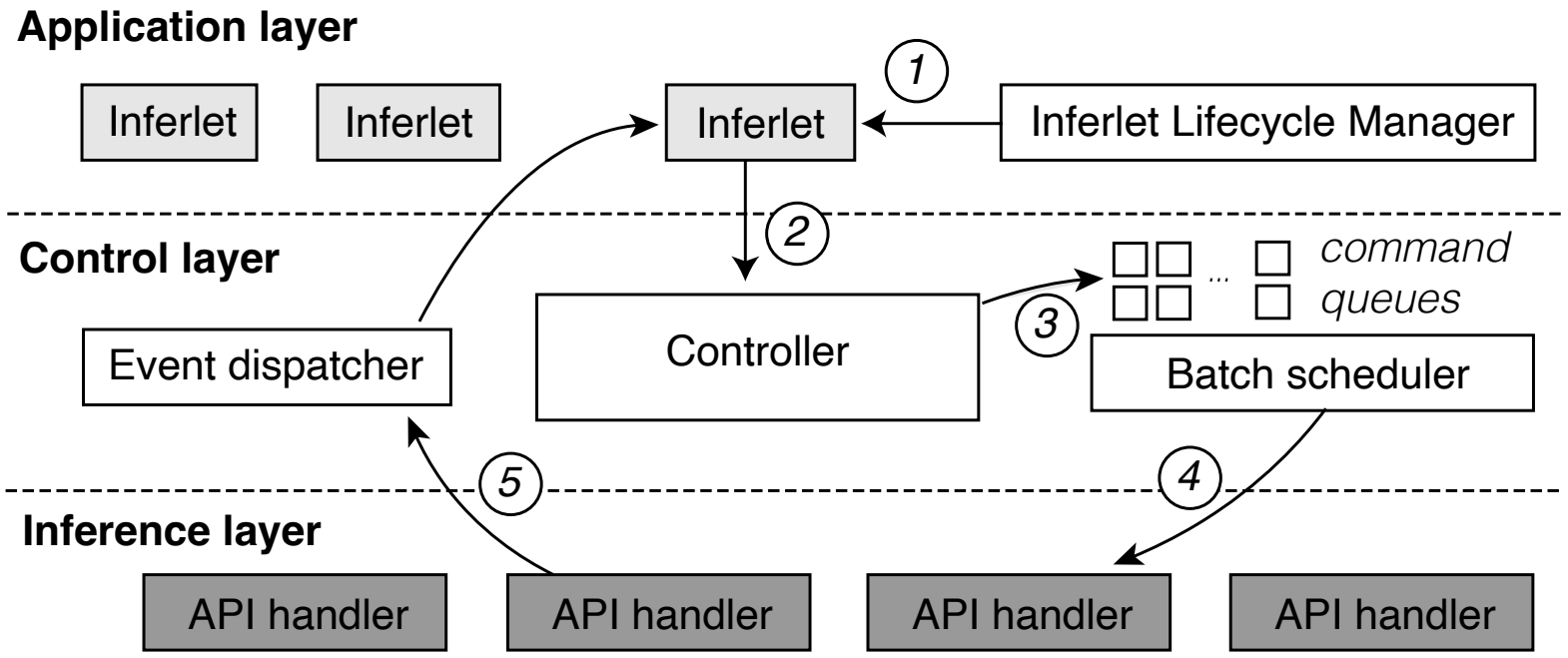
- ❖Application Layer
- ❖Control Layer
- ❖Inference Layer



Pie – System Architecture

❑Application Layer

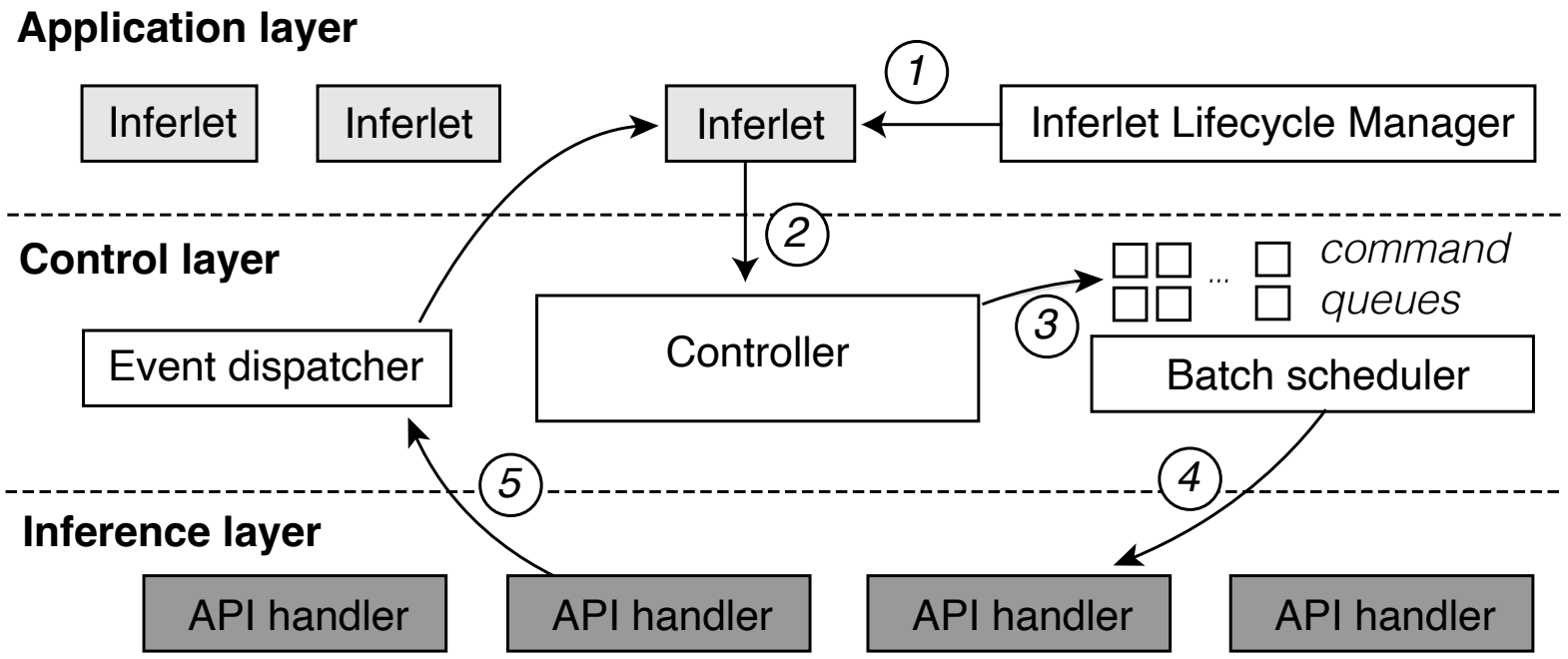
- ❖WebAssembly Runtime for Isolation
- ❖Inferlet Lifecycle Manager:
  - ➢Manages creation, destruction and communication of inferlets
  - ➢Provides `send()` and `recv()` API for user interaction



Pie – System Architecture

❑Control Layer

- ❖Handle non-GPU API: `get_model()`, `create_queue()`, …
- ❖Resource management
- ❖Batch GPU API calls

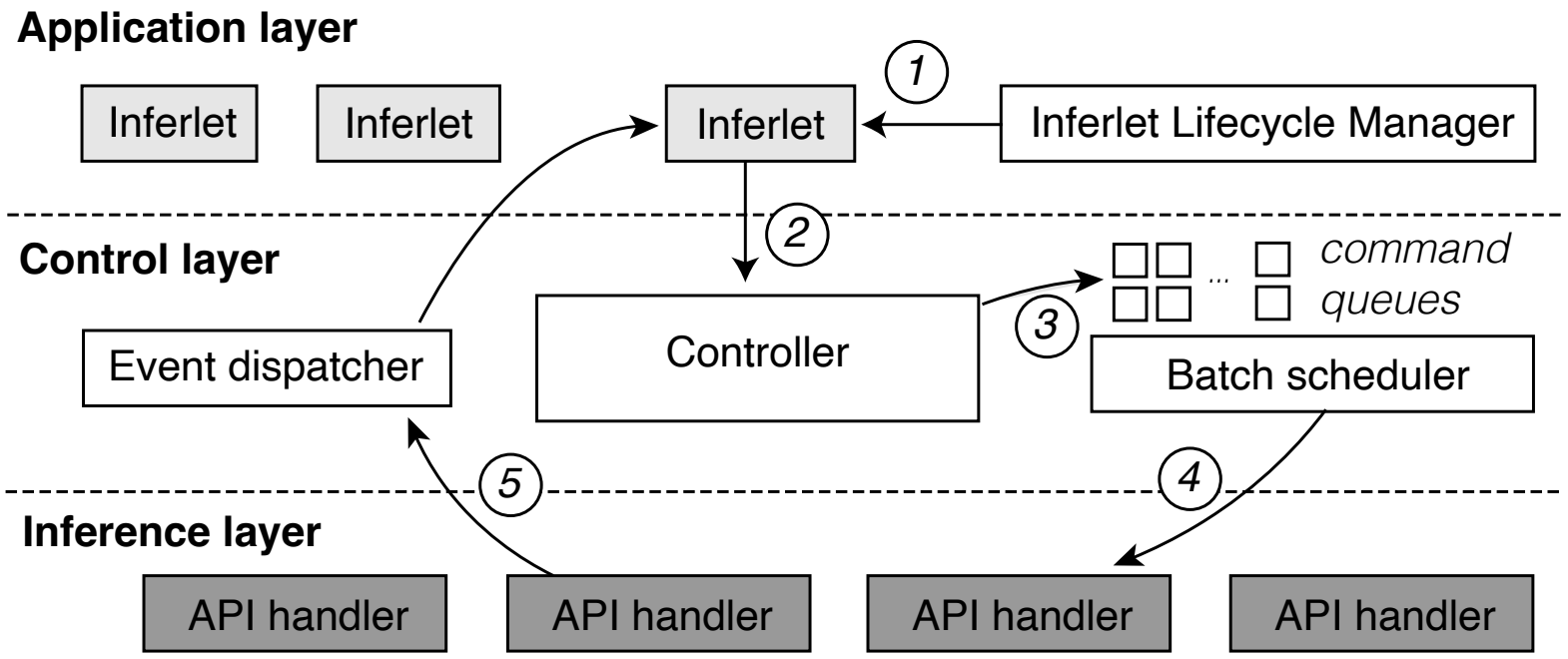


Pie – System Architecture

❑Control Layer

❖Resource Management

- ➢ Pre-allocate device memory at startup
- ➢ Manage virtual mapping for Embeds and KvPages
- ➢ Resolve resource contention: FCFS

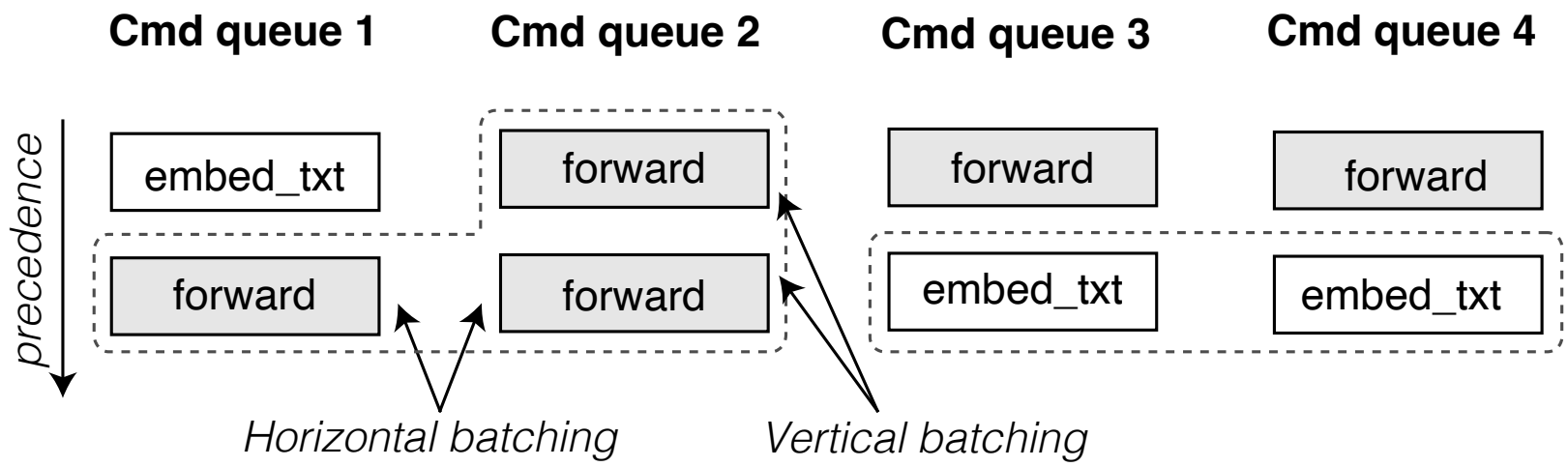


Pie – System Architecture

❑Control Layer

❖Batch Scheduling

- ➢ Batches compatible GPU API calls
- ➢ Command queue provides information about *data dependency* and *priority*
  - Vertical batching: batch non-conflict / data-independent calls
  - Horizontal batching: batch calls from different queue

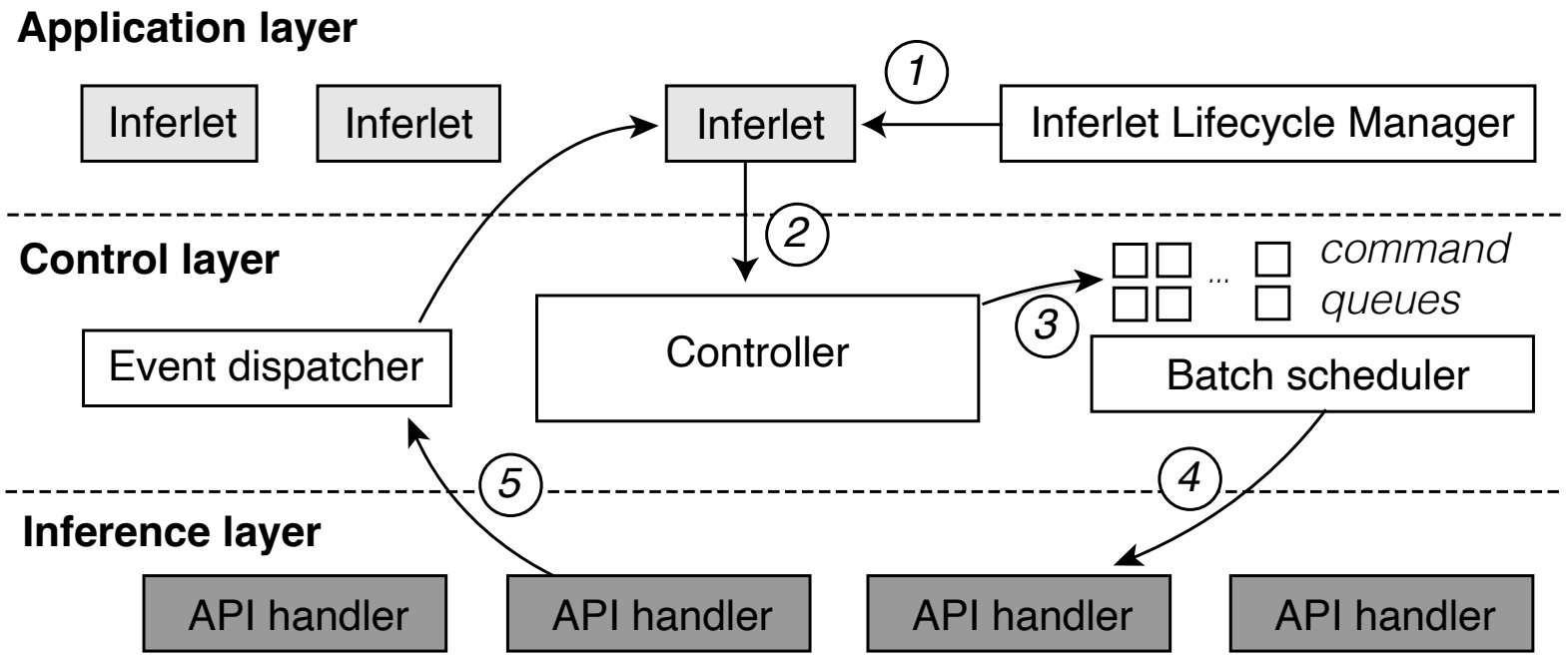


Pie – System Architecture

❑Control Layer

❖Batch Scheduling

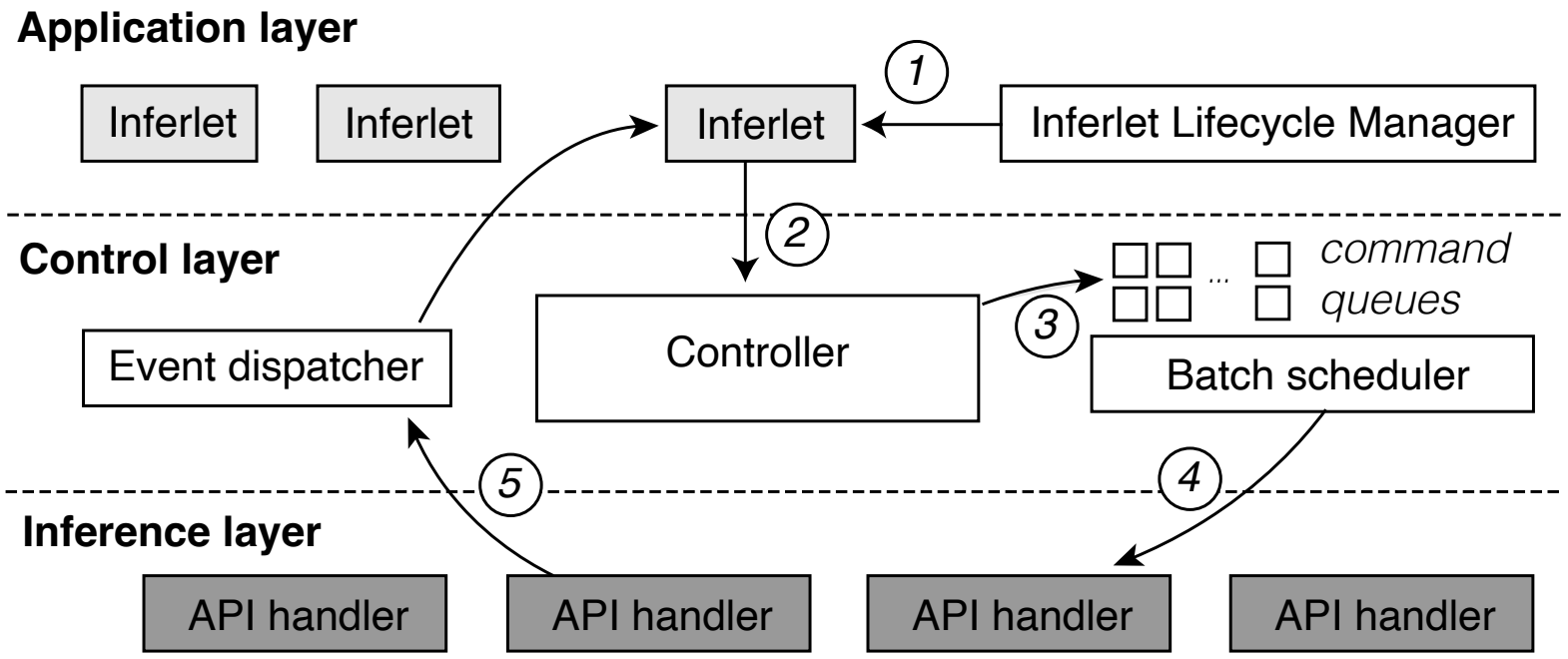
- ➢ Vertical Batching + Horizontal Batching
- ➢ Select the batch containing the longest-waiting call
- ➢ Dispatch Timing: as soon as GPU becomes idle



Pie – System Architecture

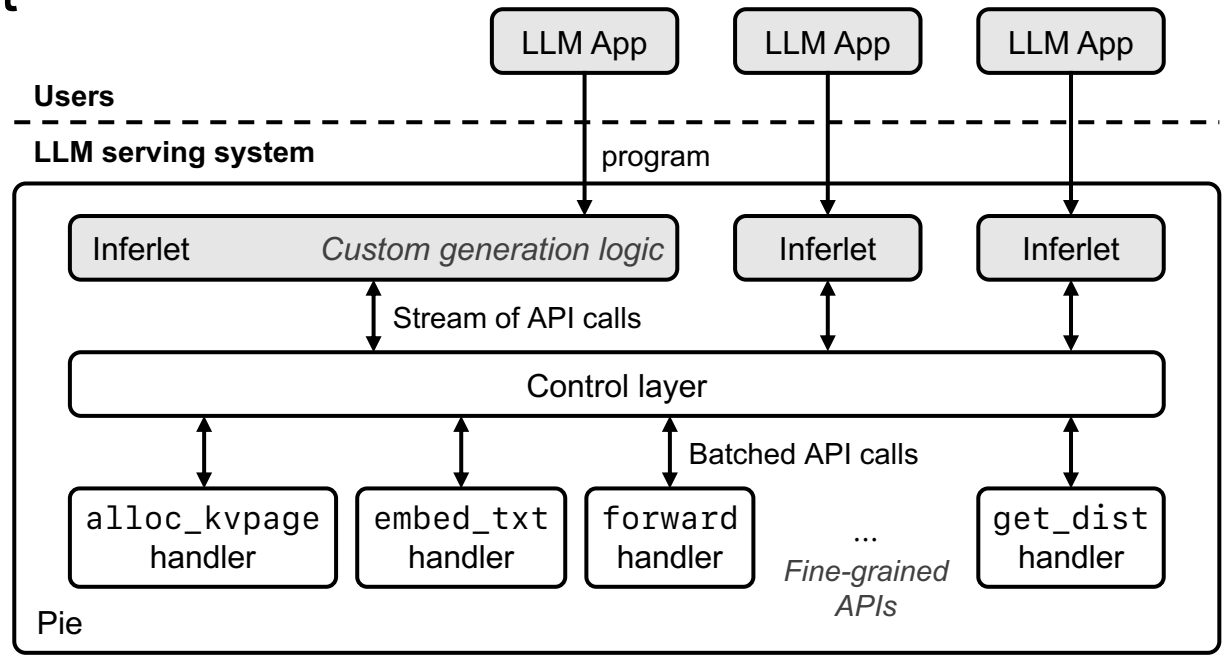
❑Inference Layer

- ❖Handle GPU API call / Invoke GPU computation
- ❖Backend: PyTorch + FlashInfer
  - ➢ An incomplete but faster C++/CUDA backend also available



Pie – Summary

- **Programming Model**
  - Expressive & Extensible
  - Explicit Resource Management
  - User-friendly Support Library
- **Three-layer Architecture**
  - Application Layer
  - Control Layer
  - Inference Layer



Outline

- ❑Background
- ❑Design
- ❑**Evaluation**
- ❑Discussion

Evaluation

❑Concerns questions

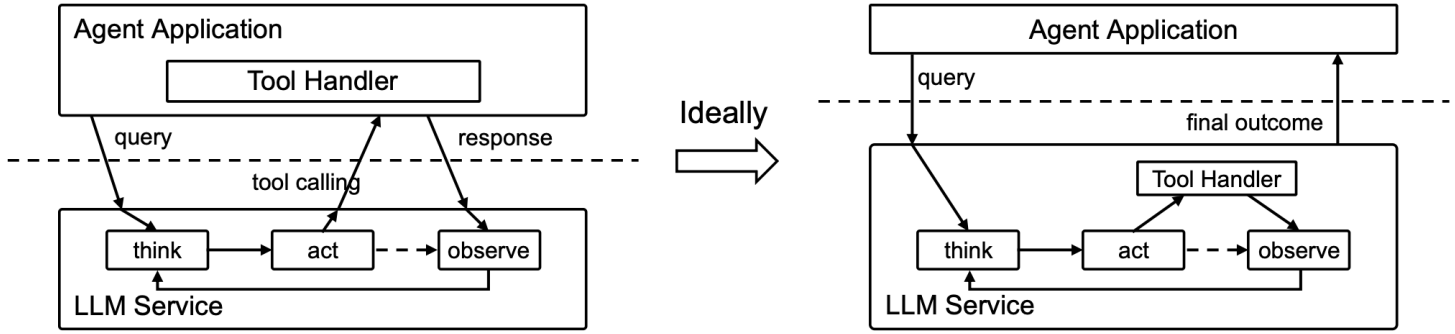
1. How does Pie **facilitate** the deployment of LLM applications?
2. How does Pie improve end-to-end **latency** and **throughput**?
3. What **overhead** is introduced?

❑Setup

- ❖ Hardware: A single NVIDIA L4 GPU (24 GB)
- ❖ Model: Llama 3 of 1B, 3B, and 8B

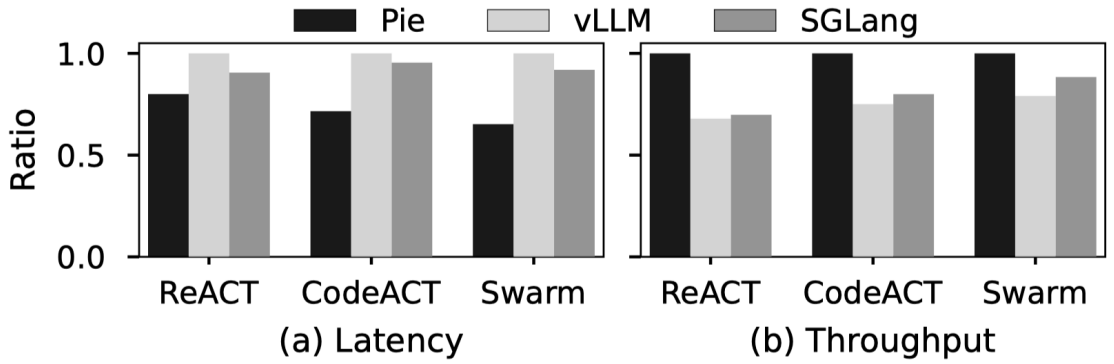
Evaluation – Agentic Workflow

❑Workflow comparison



❑E2E evaluation on a 1B model

- ❖up to 15% latency reduction
- ❖up to 30% throughput speedup



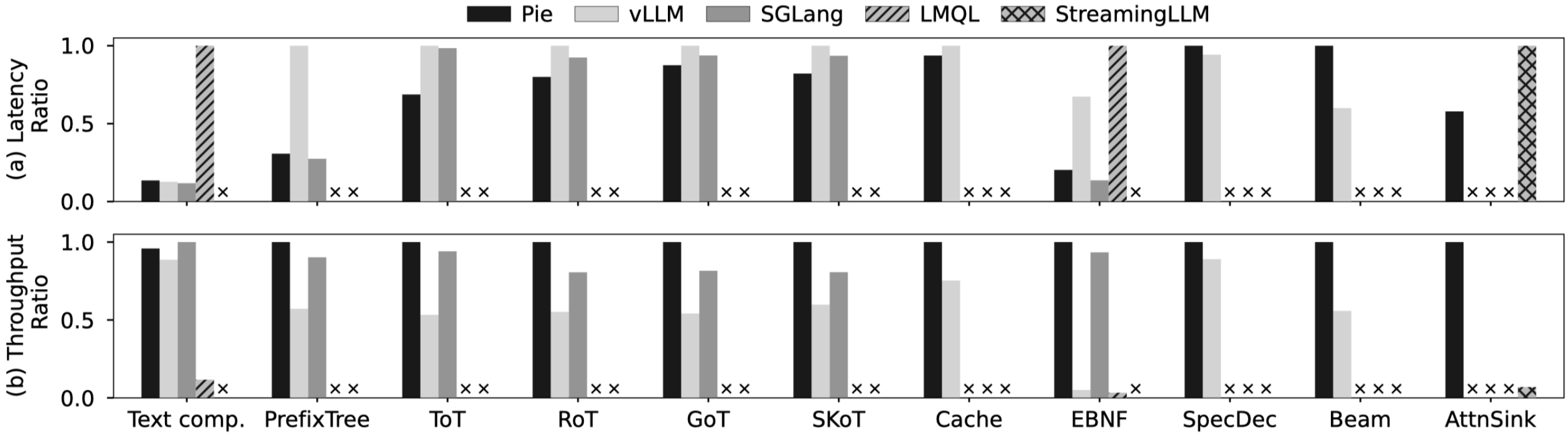
❑Speedup related to I/O

- ❖Smaller models (1,3B): reducing round-trip time
- ❖Larger models (8B): retaining the KV cache across I/O interactions

Evaluation – LLM inference techniques

❑Latency & Throughput

- Improved performance for tasks with various inference techniques
  - except pure text comp. without I/O

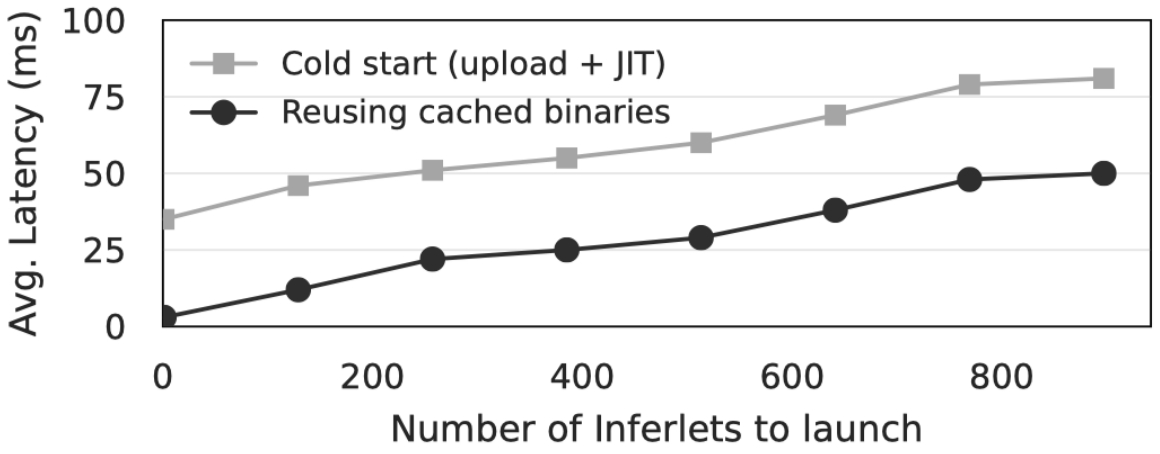


❑Effective expressiveness and implementation on Pie

Evaluation – System Overhead of Pie

❑Latency overhead of cold start

- Just like FaaS, cold start exists for user-defined sandboxed code
- Duration measured for a text completion task
  - T_inferlet_start_forward – T_inferlet_get_triggered
  - Negligible compared with per-token generation latency



- However, only the same inferlet is considered here
  - For different inferlets, uploading and JIT overhead would be much higher

Evaluation – System Overhead of Pie

❑Latency overhead per decoding step (warm start)

- Monolithic vLLM is faster for pure text completion task without I/O

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

❑Reduced overhead ratio

- for larger models

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%)

Evaluation – Batch Strategy

- **Baselines**
  - **Eager:** no batching
  - **K-only:** fixed-size batching based on queue length K
  - **T-only:** tiemout-based batching based on wat time
- Adaptive batching in Pie improves throughput
  - by 17x vs Eager baesline
  - by 8-40% vs K-only and T-only

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

Outline

- ❑Background
- ❑Design
- ❑Evaluation
- ❑**Discussion**

Discussion

❑Contributions

- ❖Fine-grained, customizable KV cache management
- ❖Ability to provide client-side information to speed-up inference
- ❖Flexible generation loop → heterogeneous workload at the same time

❑Discussions

- ❖Scalability: only single-instance/GPU inference is supported
- ❖Security: handling malicious inferlet
- ❖The inherit complexity of LLM service cannot be removed (Tesler's law)