Pie: A Programmable Serving System for Emerging LLM Applications

In Gim Zhiyao Ma Seung-seob Lee Lin Zhong

SOSP 2025

Yale University

马浩然

2026.1.19

章节介绍

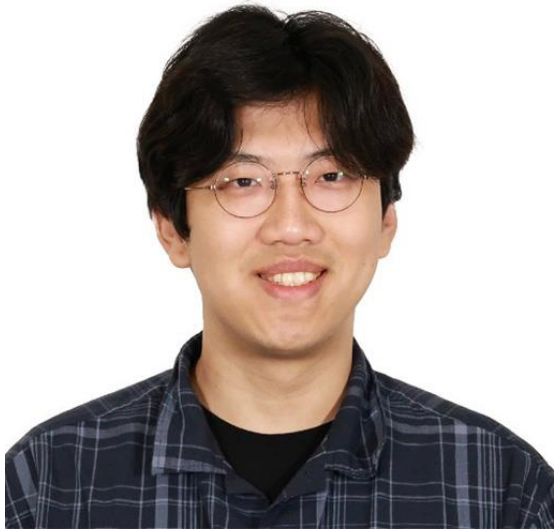
- 作者介绍
- 研究背景与动机
- PIE 系统设计
- 实验评估
- 总结与讨论

章节介绍

- 作者介绍
- 研究背景与动机
- Pie 系统设计
- 实验评估
- 总结与讨论

作者介绍

In Gim



耶鲁大学计算机科学系在读博士生，导师为钟林教授

主要研究方向

- 机器学习系统：专注于为 AI 构建可编程系统架构
- 大语言模型推理服务系统：研究如何提高 LLM 推理的效率、灵活性与可持续性
- 系统安全与隐私：涉及保护云端 LLM 用户提示词的机密性以及针对 LLM 服务的网络侧信道攻击研究

论文发表：

- [EMNLP'25] Cacheback: Speculative Decoding With Nothing But Cache
- [SOSP'25] Pie: A Programmable Serving System for Emerging LLM Applications
- [HotOS'25] Serve Programs, Not Prompts
- [MLSys'24] Prompt Cache: Modular Attention Reuse for Low-Latency Inference

章节介绍

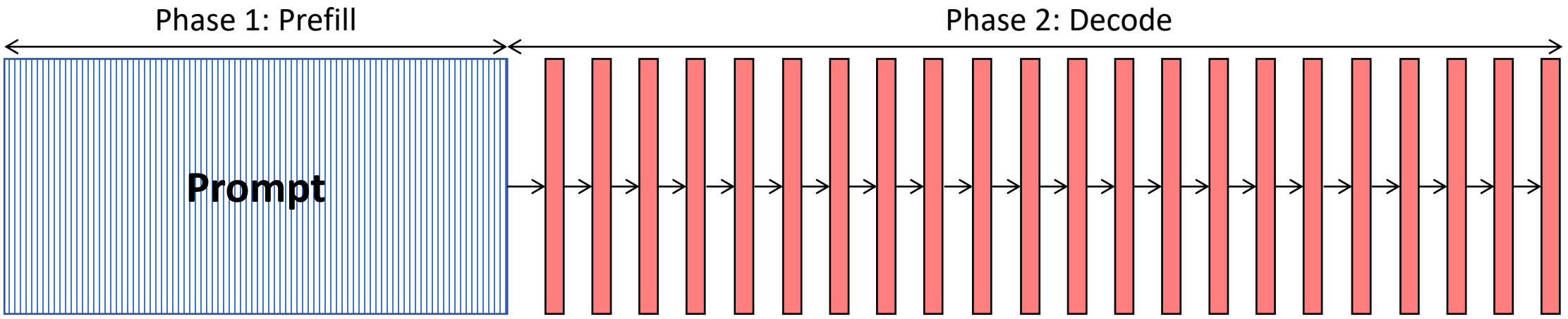
- 作者介绍
- **研究背景与动机**
- Pie 系统设计
- 实验评估
- 总结与讨论

研究背景与动机

LLM 生成过程

- **阶段一：Prefill**
  计算所有输入 Token 的表示，并生成初始的 KV Cache 供解码阶段使用

- **阶段二：Decode**
  1. 计算：输入上一个 Token + 历史 KV Cache 进行计算
  2 .采样：根据概率分布以及特定的策略选择下一个 Token
  3. 更新：将新 Token 的 KV 信息存入 KV Cache，准备下一次迭代

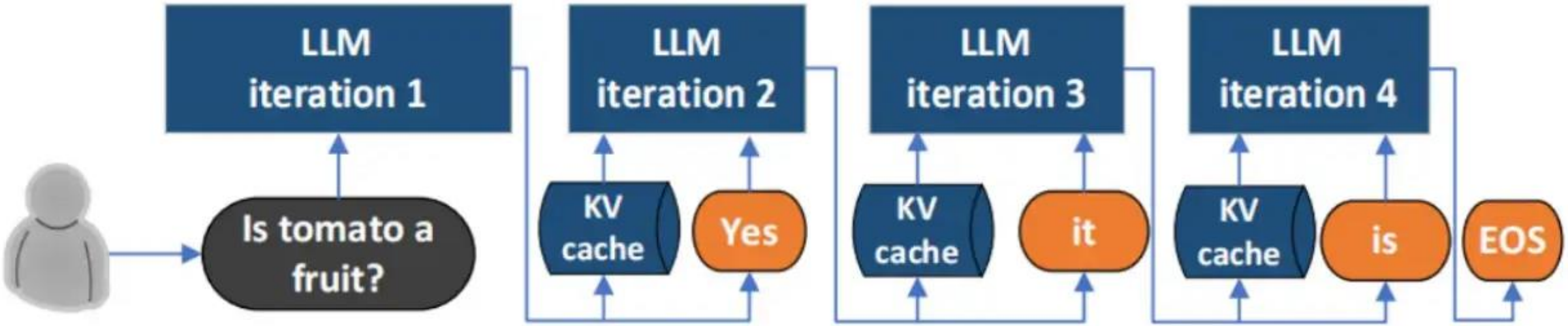


研究背景与动机

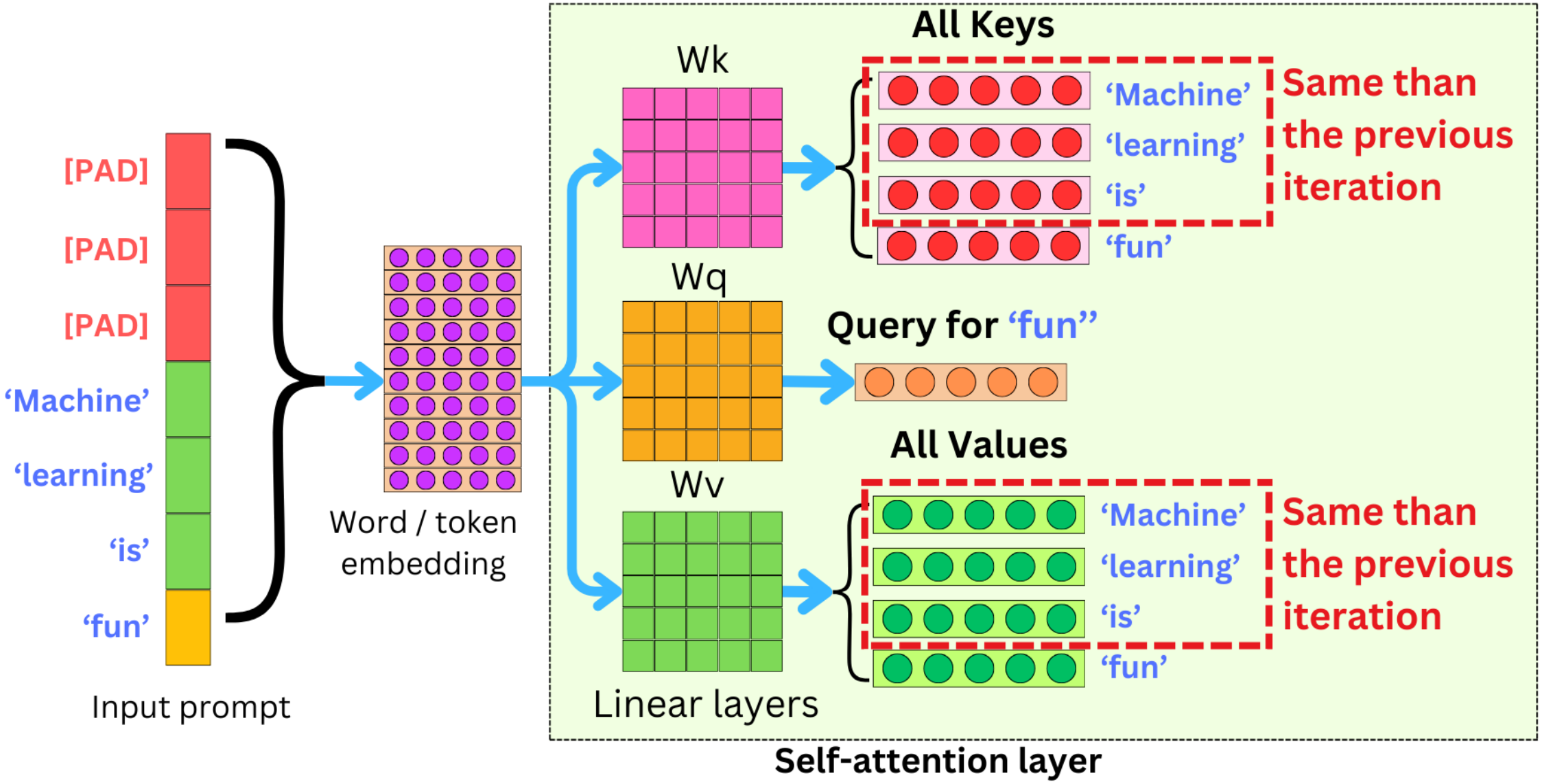
什么是 KV Cache

- **定义：**存储 Transformer 模型在推理过程中，先前 Token 生成的中间状态（Key 向量和 Value 向量）

- **核心目的：**避免重复计算，在自回归生成时，模型计算第 N 个词需要之前所有 N-1 个词的信息，缓存这些中间结果可以显著加速推理



研究背景与动机


[PAD]
[PAD]
[PAD]
‘Machine’
‘learning’
‘is’
‘fun’
Input prompt
Word / token embedding
Wk
Wq
Wv
Linear layers
All Keys
‘Machine’
‘learning’
‘is’
‘fun’
Same than the previous iteration
Query for ‘fun’’
All Values
‘Machine’
‘learning’
‘is’
‘fun’
Same than the previous iteration
Self-attention layer

研究背景与动机

主流 LLM Serving 系统概览

设计初衷：主要为大规模、高吞吐量的文本补全任务而设计
代表性系统：

- vLLM：引入 PagedAttention 解决显存碎片化问题
- SGLang: 引入了 RadixAttention，使用前缀树管理缓存
- TGI (Text Generation Inference)：Hugging Face 推出的生产级推理后端

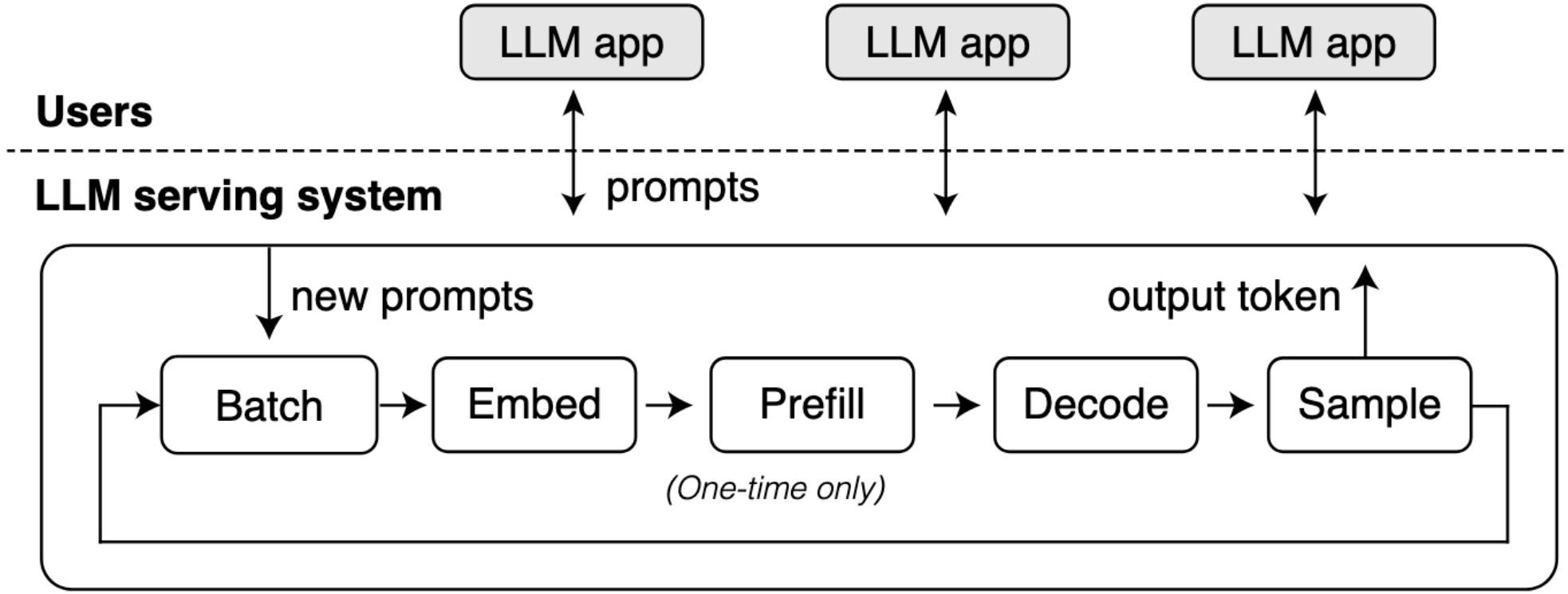
共同目标：通过优化显存管理和请求调度，尽可能压榨硬件的利用率

Text Generation Inference

研究背景与动机

主流 LLM Serving 系统核心设计

黑盒设计： 将 LLM 推理封装成一个预填充-解码的固定循环
固定流程： 每个请求按照“**批处理 -> 嵌入 -> 预填充 -> 解码 -> 采样**”的流水线运行



研究背景与动机

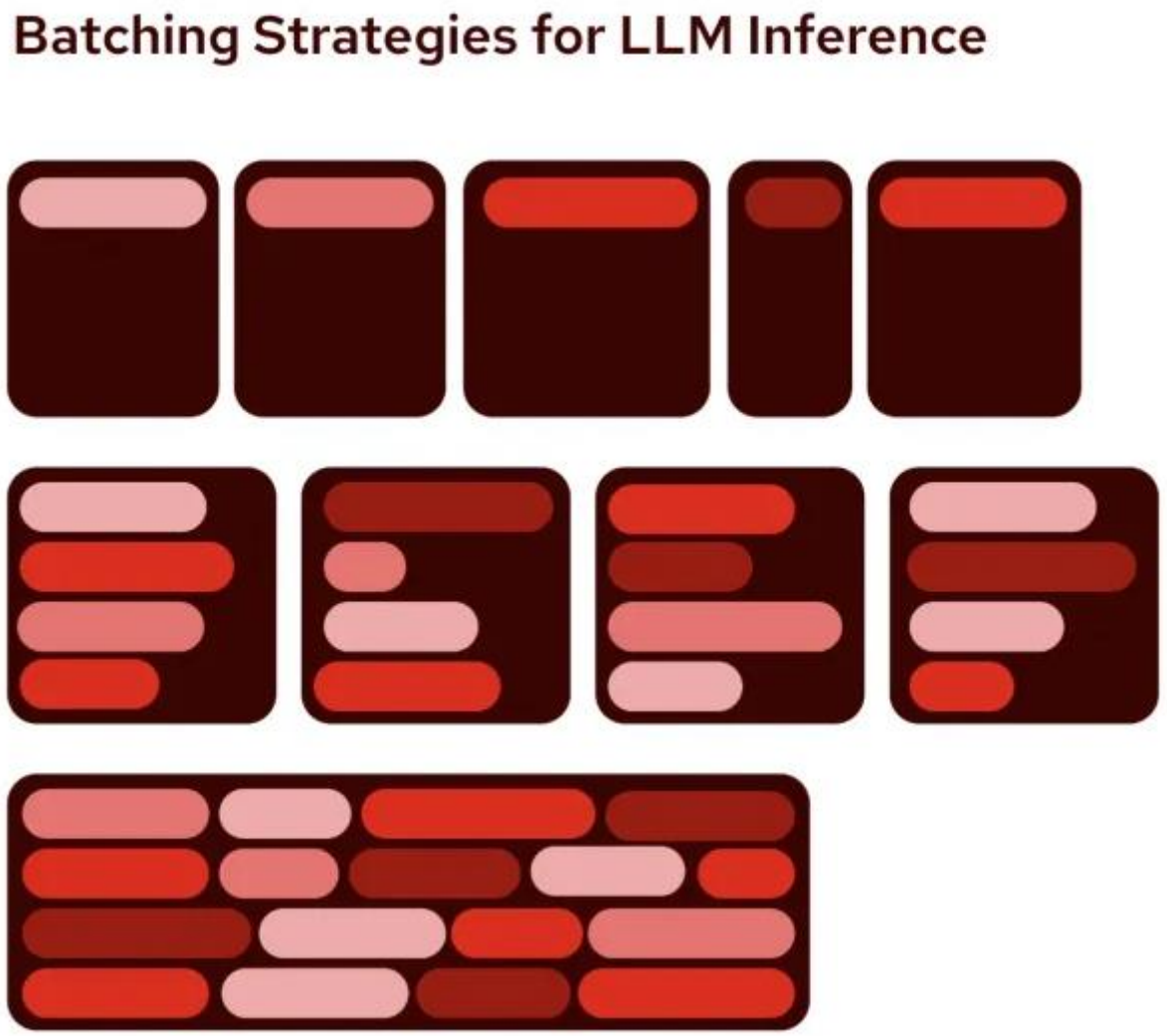
主流 LLM Serving 系统核心设计

请求批处理：中央调度器将多个独立的请求组合成一个批次

同步执行：

- 批次内的所有请求在 GPU 上并行计算
- 这种设计极大地提高了硬件利用率和整体吞吐量，最大化地利用 GPU 的并行计算能力

如图所示：独立请求，动态批处理，连续批处理



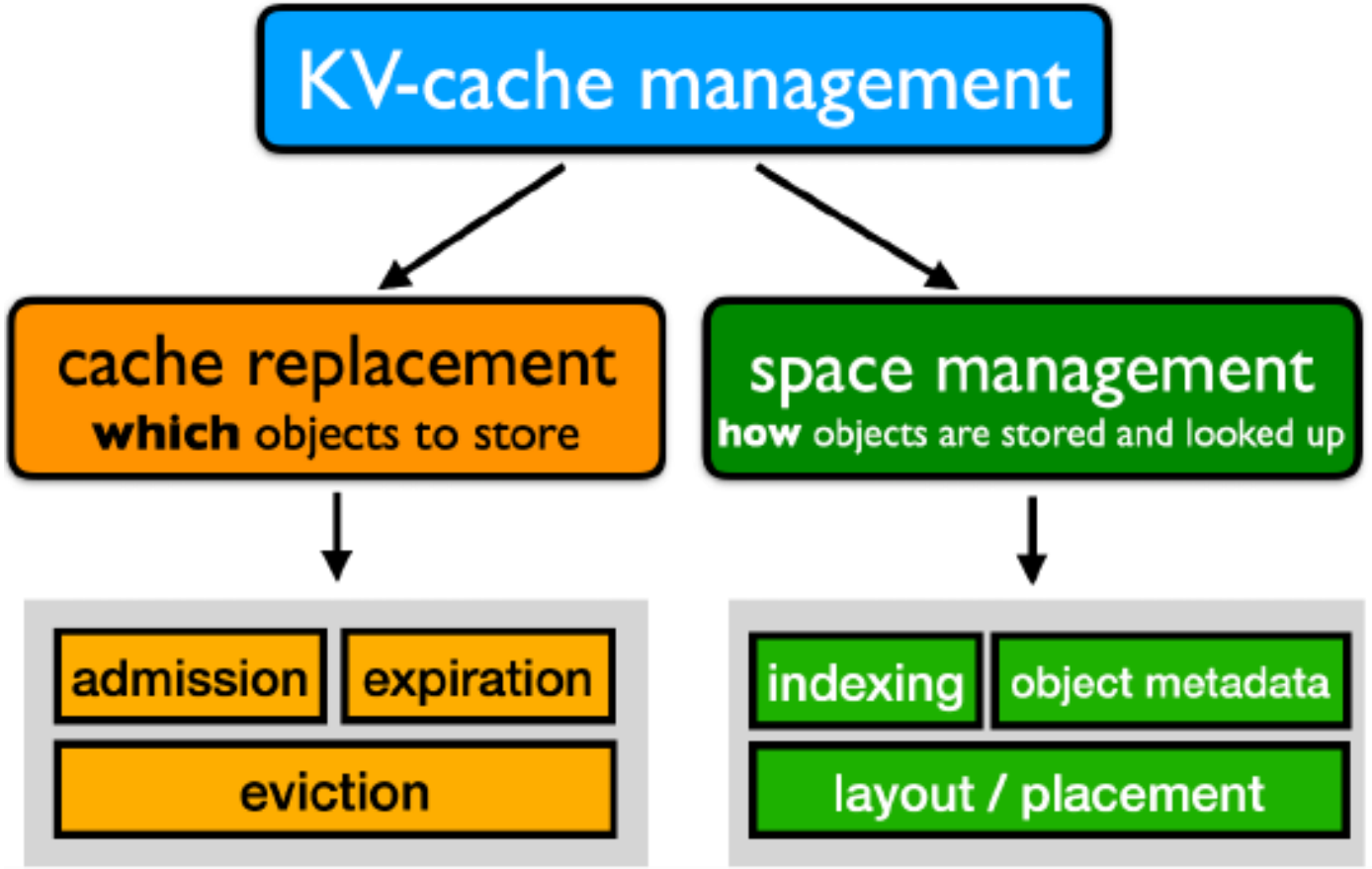
研究背景与动机

主流 LLM Serving 核心设计

隐式 KV Cache 管理
系统使用全局启发式算法自动决定缓存的管理，包括显存空间的分配，缓存块的剔除和缓存的复用等行为

典型策略

- LRU：自动清理最久未使用的缓存块
- 前缀缓存：识别公共开头的 Token 序列并共享缓存



研究背景与动机

LLM 新范式的发展

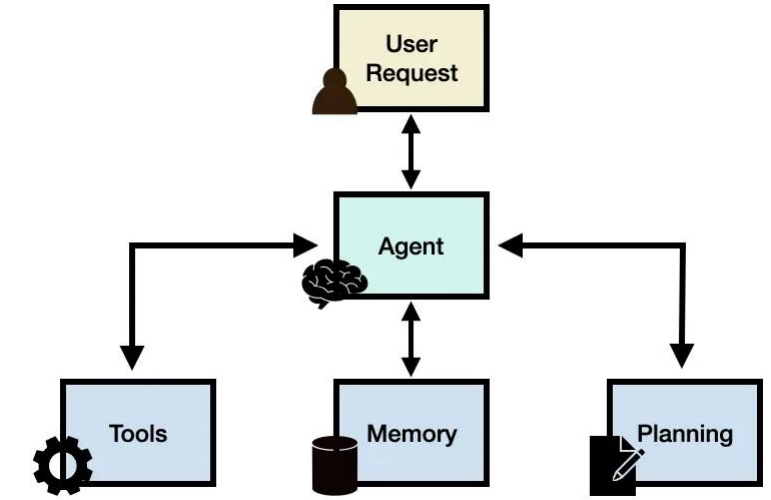
应用范式的三大演进趋势

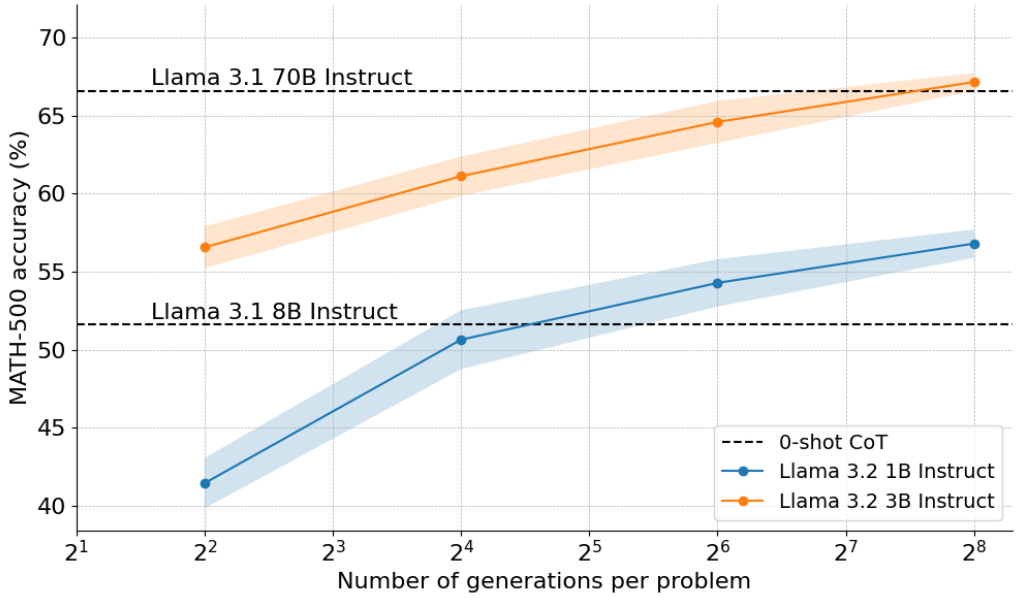
大语言模型正在从传统的单次文本补全，演进为具备复杂规划、深度推理及环境感知能力的计算中枢

计算范式-推理侧扩展(Test-Time Scaling)：推理阶段投入更多计算资源来提升模型的能力

控制范式-算法化/结构化生成：通过引入外部算法或约束条件，实现更高效或更规范的生成

交互范式-自主交互智能体：LLM 具备了规划意图并主动调用外部工具的能力



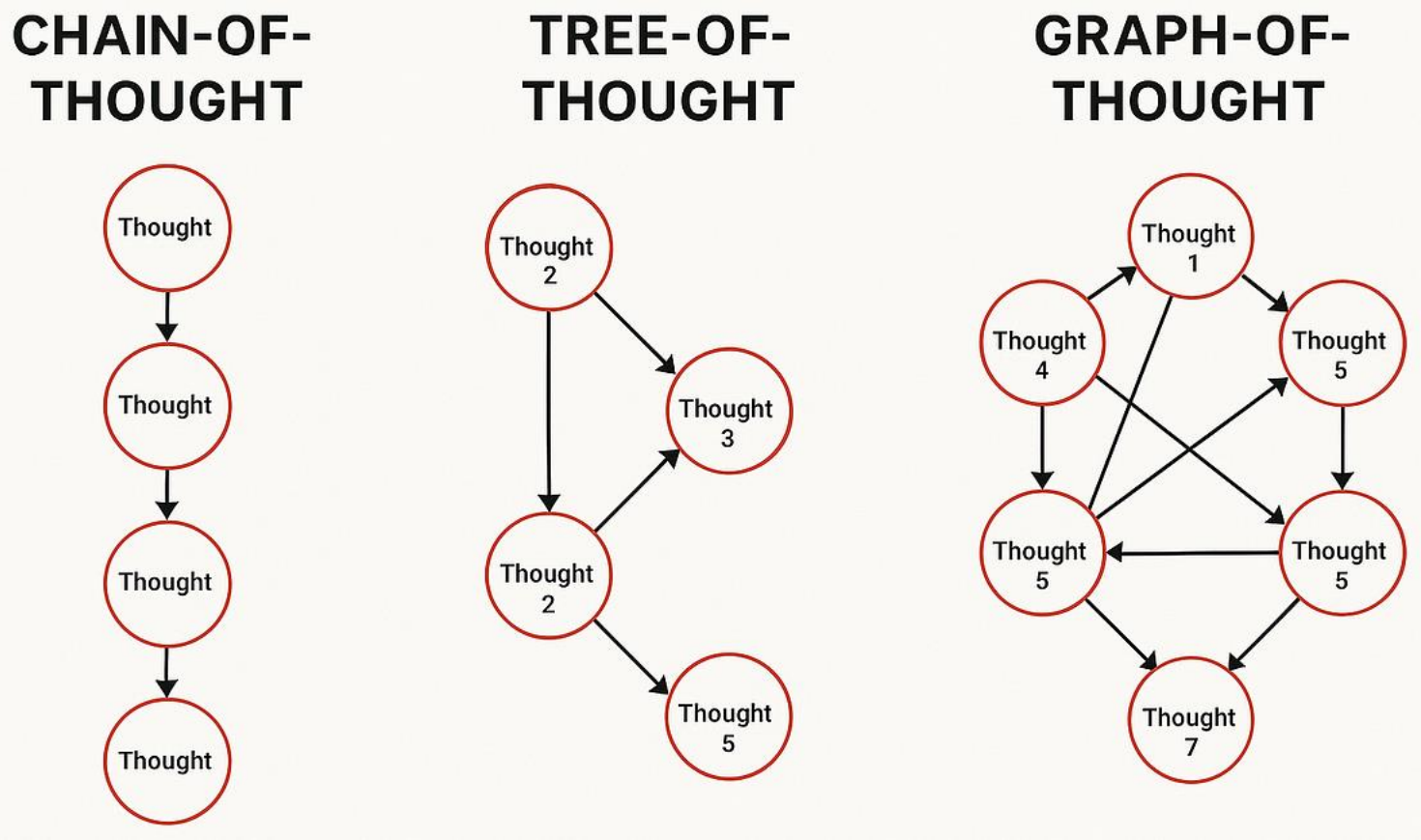


研究背景与动机

推理侧扩展 (Test-time Scaling)

核心理念：计算成本的再分配

- 不再仅仅依赖于预训练阶段的模型规模提升，转而通过在推理阶段投入更多计算资源来解决复杂逻辑问题
- 模仿人类的“慢思考”过程，在给出最终答案前，进行多路径尝试、自我评估与修正



研究背景与动机

推理侧扩展 (Test-time Scaling)

对现有系统挑战

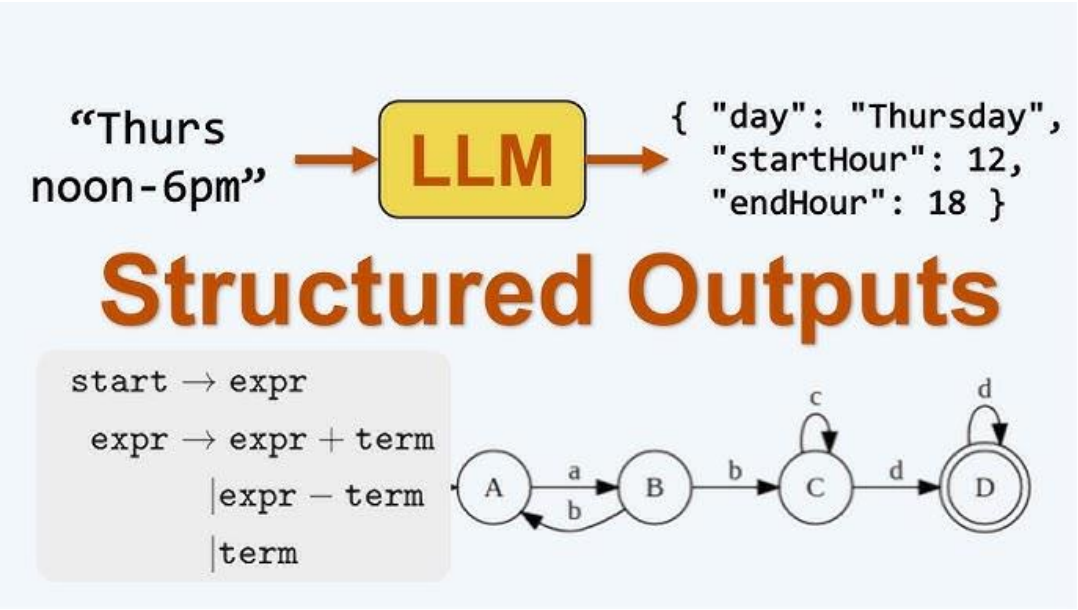
- 非线性路径的资源冲突：ToT 或 GoT 要求在运行时对特定的 KV Cache 块进行保留、丢弃、重用或复制，超出了传统系统的能力范围
- “根节点”误删风险：当 Agent 深入探索某个长分支时，全局策略可能会误认为起始的共享分支节点是“久未访问”的并将其剔除，导致后续其他分支必须重新进行昂贵的预填充计算 。
- 侵入式修改困难：在现有单体架构中实现复杂的 beam search 或递归推理，需要修改内存管理器和调度器，工程门槛高

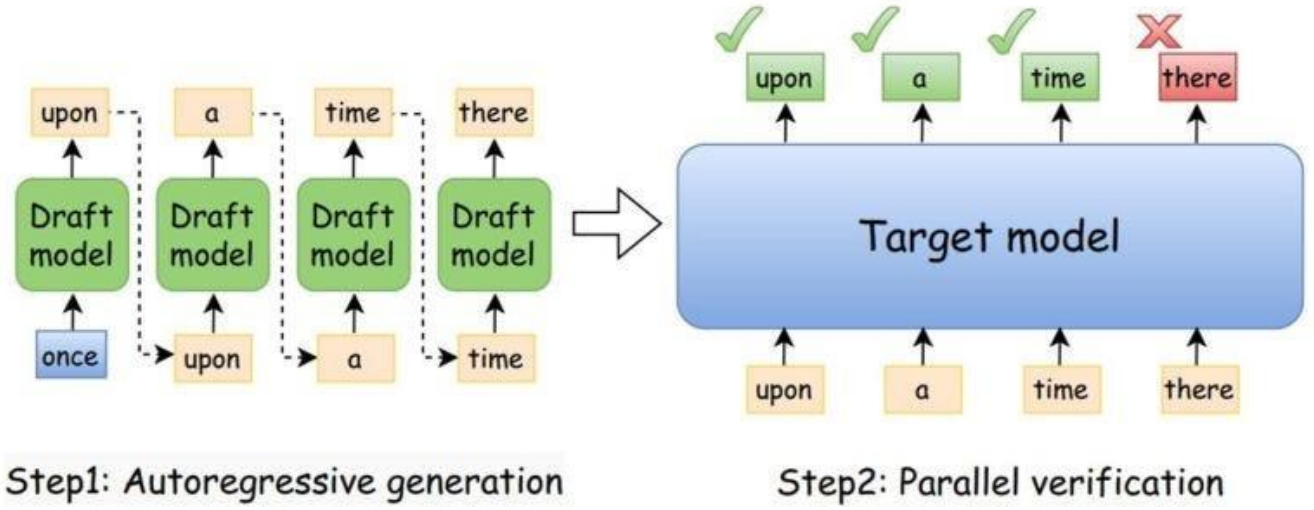
研究背景与动机

算法化/结构化生成

核心理念：从“概率抽样”转向“逻辑干预”

- 生成过程不再被动地依赖模型的概率分布，而是通过引入外部算法逻辑来提升性能（速度）或增强输出的规范性
- 要求在预测、采样及 Token 更新的每一个微观步骤中，都能动态地注入自定义业务逻辑





研究背景与动机

算法化/结构化生成

对现有系统的挑战

- “黑盒”循环的封闭性：现有的 Serving 架构将预测、采样和缓存更新封装在一个单体、不可分割的 Token 生成循环中
- 微观干预缺失：结构化输出（JSON/EBNF）或投机采样要求在“预测”之后、“采样”之前动态注入约束逻辑，而现有系统缺乏这种细粒度的 Hook
- 变长输出的冲突：投机采样一次可能产出多个 Token，这与现有 Serving 系统“每个批处理步骤产出单一 Token”的同步假设背道而驰

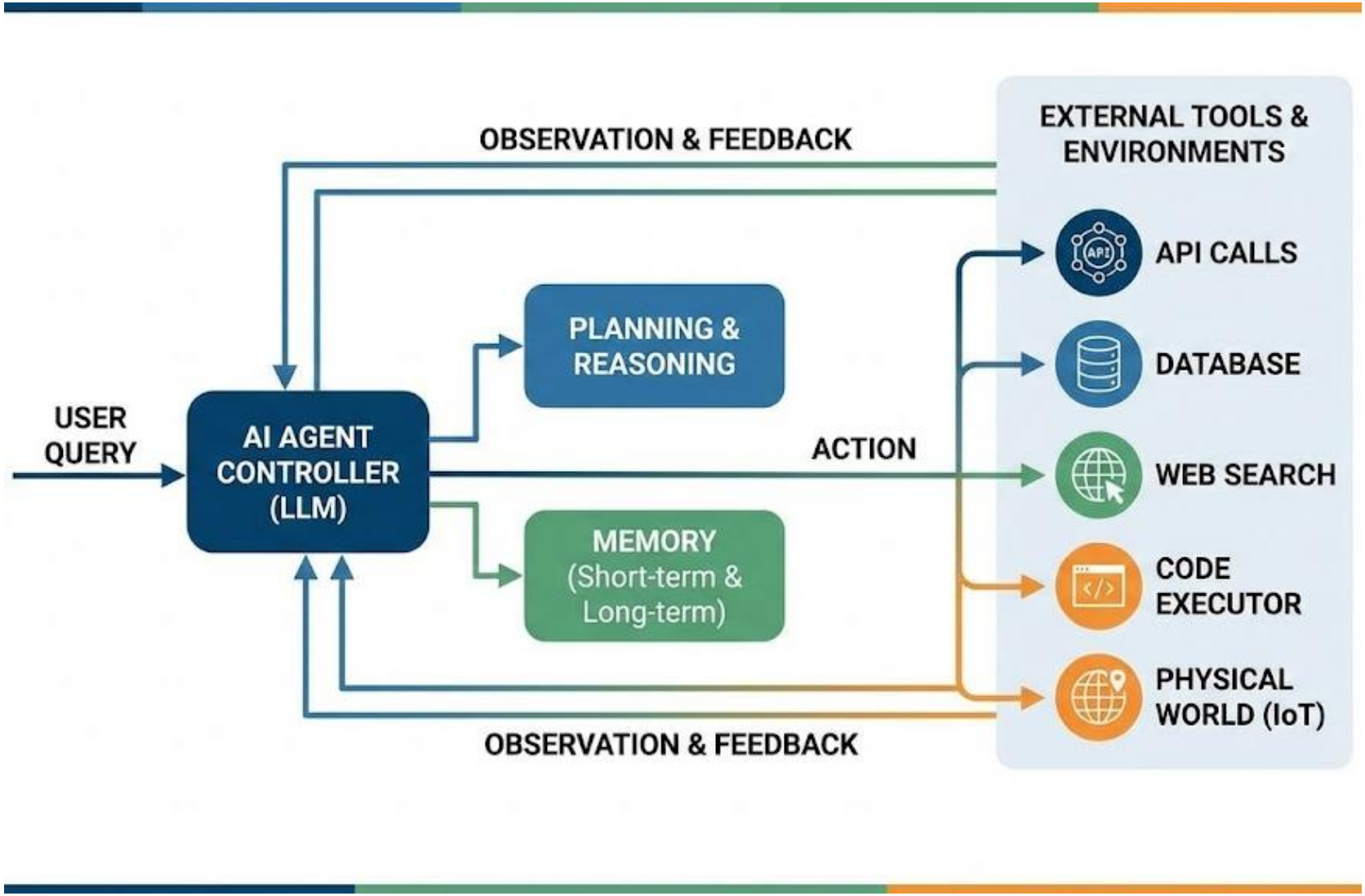
研究背景与动机

自主交互的智能体

核心理念：从“离线思考”到“在线执行”

- LM 不再仅仅是生成静态文本，而是作为中枢控制器，负责规划任务、调用外部资源并根据反馈调整决策。

- 形成“推理 -> 行动 -> 观察 -> 调整”循环，赋予模型解决现实世界问题的能力



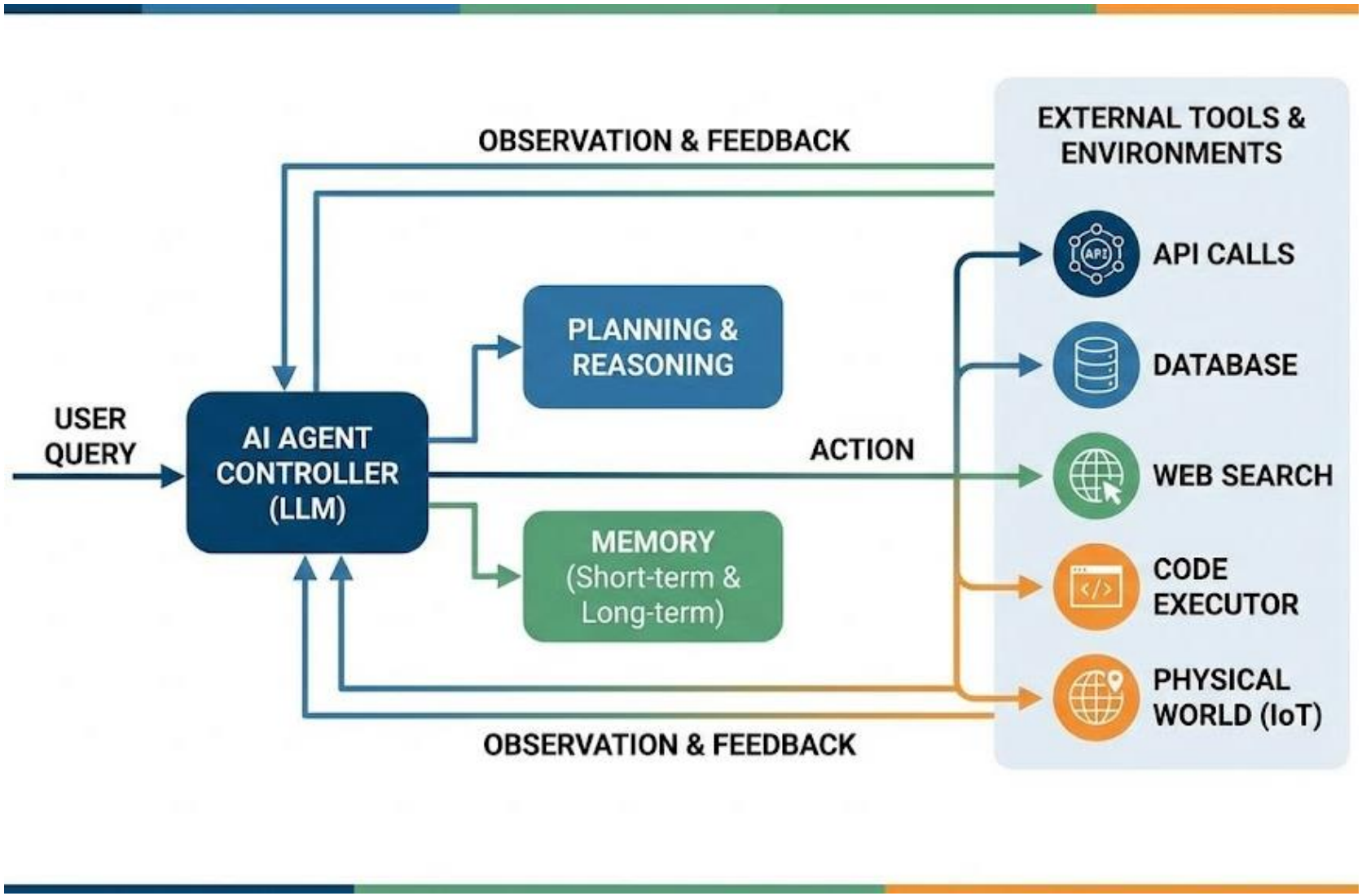
研究背景与动机

自主交互的智能体

对现有系统的挑战

- **推理与 I/O 的割裂**：推理过程在调用外部工具时必须中断，控制权在客户端与系统间频繁切换，产生巨大的延迟

- **状态丢失：**由于系统无状态，等待 I/O 反馈期间 KV Cache 往往被丢弃，继续任务时必须从头开始极其昂贵的重复预填充



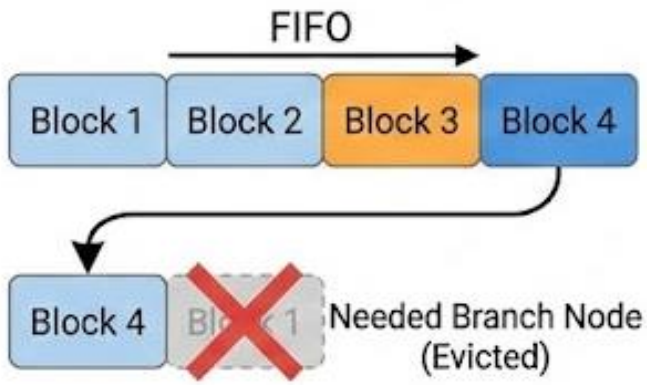
章节介绍

- 作者介绍
- 研究背景与动机
- **PIE 系统设计**
- 实验评估
- 总结与讨论

PIE 系统设计

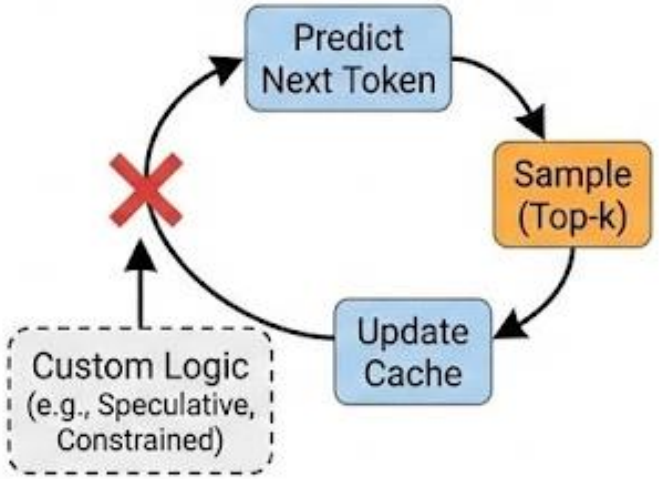
现有系统无法满足的三大需求

R1 细粒度 KV Cache 控制



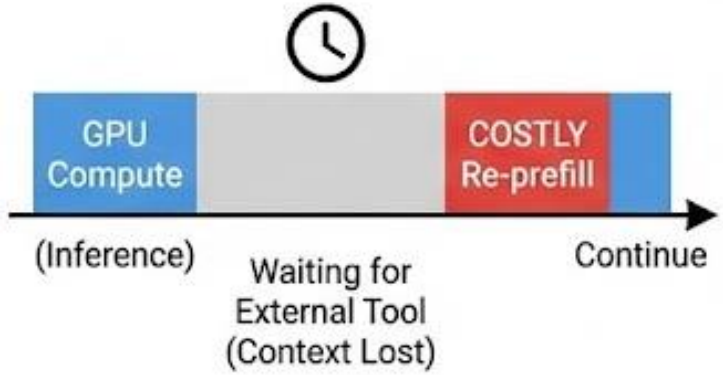
复杂推理需要保留特定分支的 KV Cache，现有系统只提供全局的 LRU，导致有效上下文被错误的清除

R2 自定义生成逻辑



投机采样/约束解码需要修改解码过程，但现有单体循环架构是封闭的，无法注入自定义逻辑

R3 计算与I/O 割裂

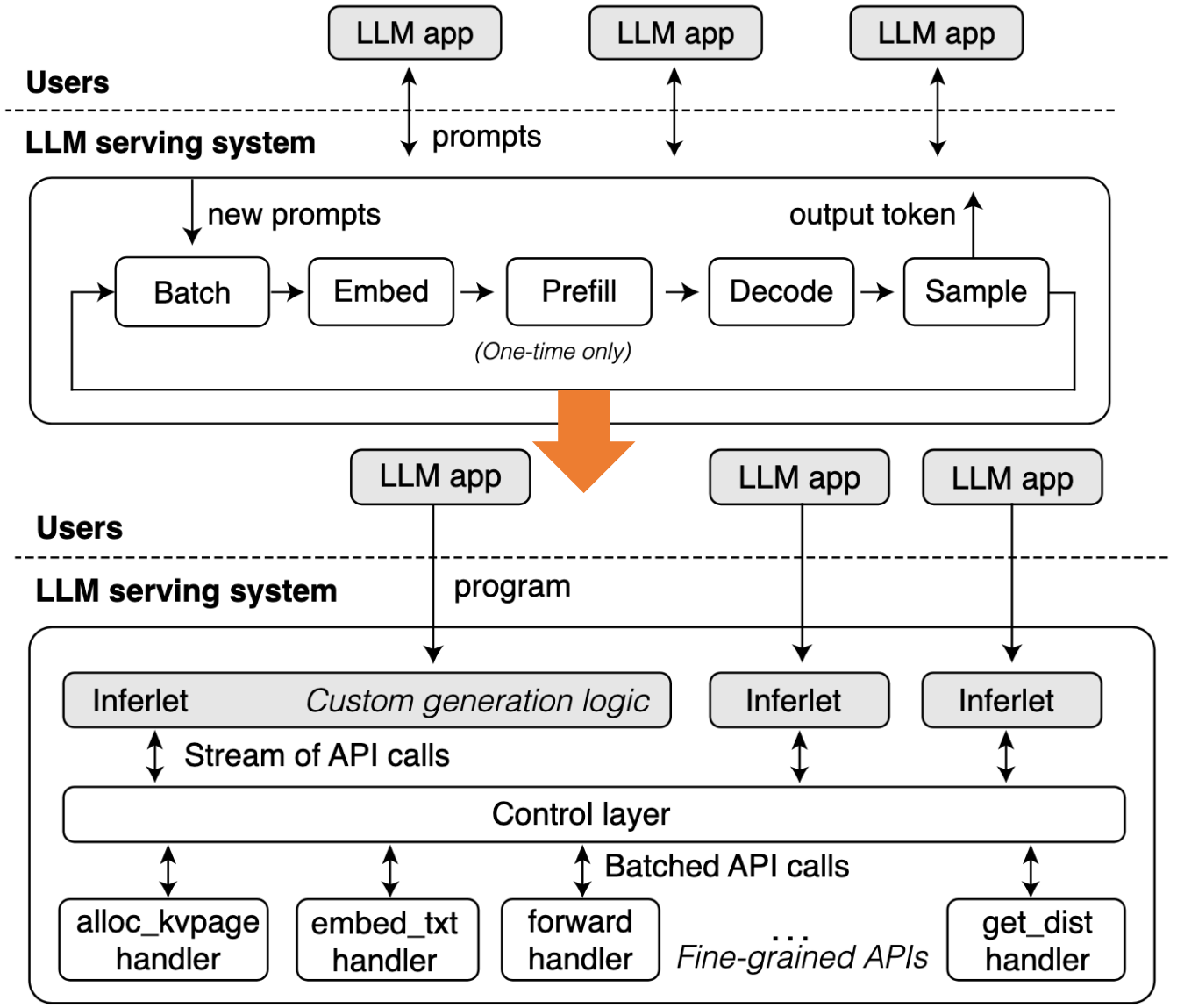


Agent 与外部工具/环境交互会导致推理中断，Cache 丢失，引发高延迟的 Round-Trip

PIE 系统设计

设计思想：解构与细粒度控制

- **核心逻辑：**打破死板的流水线，将推理过程拆解为细粒度的独立处理器
- **核心单位：**不再服务于简单的“提示词”，而是服务于用户编写的“Inferlets”
- **核心目标：**将 KV Cache 的管理、生成逻辑以及 I/O 的集成，交给开发者



PIE 系统设计

PIE 系统三层架构

应用层

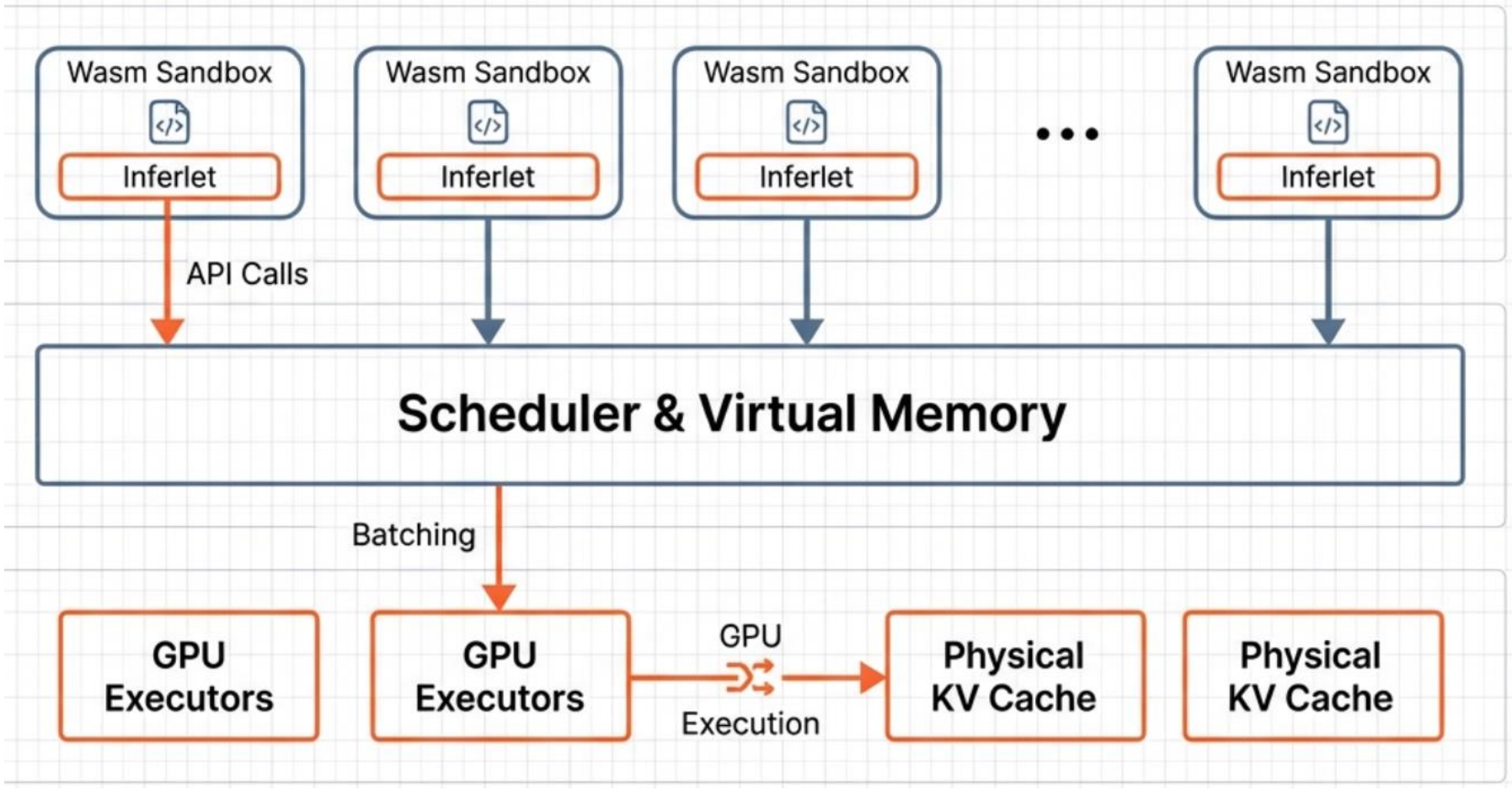
- 运行并隔离用户的 Inferlet 程序

控制层

- 接收来自不同 Inferlet 的异步 API 调用流，并进行全局调度

推理层

- 通过专用的 Handlers 执行具体的算子计算



PIE 系统设计

应用层：Inferlet 推理程序化能力

定义：将模型推理程序化

- Inferlet 是用户提供的、运行在 PIE 环境中的推理小程序
- 范式转变：系统不再以简单的 Prompt 作为服务单位，而是服务于一个完整的推理程序

Inferlet 作用：

- 控制权委派：PIE 将端到端的生成控制权从系统内核委派给 Inferlet

- 资源编排：Inferlet 通过 API 显式调用底层处理程序，灵活编排 Embed、Forward和 Sample 等步骤

```
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)
```

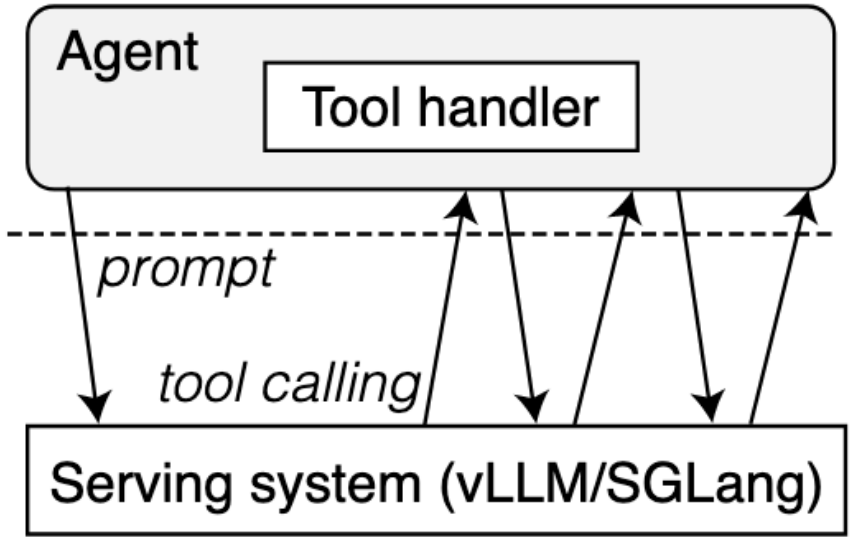
PIE 系统设计

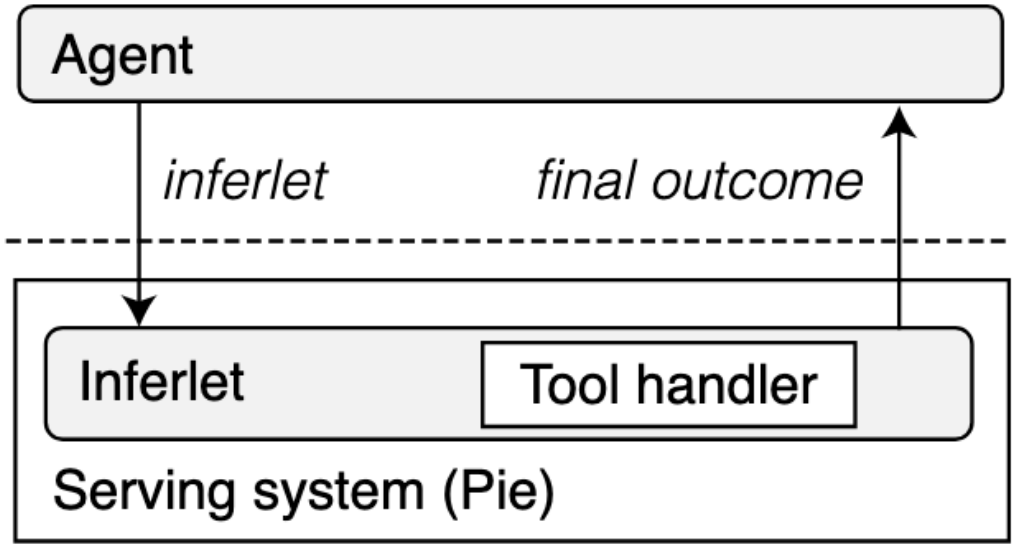
应用层：Inferlet 推理程序化能力

定义：将模型推理程序化

- Inferlet 是用户提供的、运行在 PIE 环境中的推理小程序

- 范式转变：系统不再以简单的 Prompt 作为服务单位，而是服务于一个完整的推理程序





PIE 系统设计

应用层：API 编程模型

核心抽象：基于特性的 APIs

- PIE 将复杂的模型能力拆解为 42 个细粒度接口，并按功能划分为不同的 Trait

- 解耦设计：推理流程不再是一个死板的闭环，而被解构为“资源分配、张量计算、结果采样”等独立积木块

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 系统设计

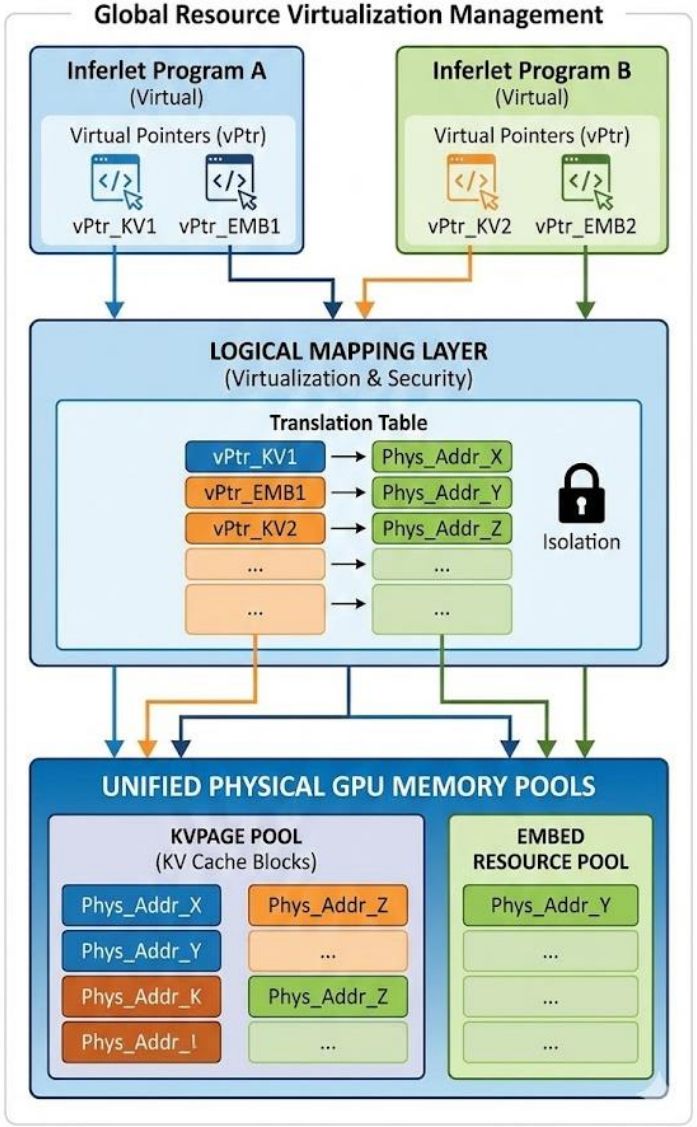
控制层：系统的调度中枢

不同 Inferlet 的 API 调用流是异步且非确定的，控制层作为中介，将凌乱的 API 请求转化为高效的批处理指令

控制层核心功能一：全局资源虚拟化管理

- 内存池化：统一管理物理显存中的 KV 缓存块和 EMBED 资源池

- 逻辑映射：将 Inferlet 程序中的虚拟指针映射到物理内存，确保不同程序间的资源隔离与安全



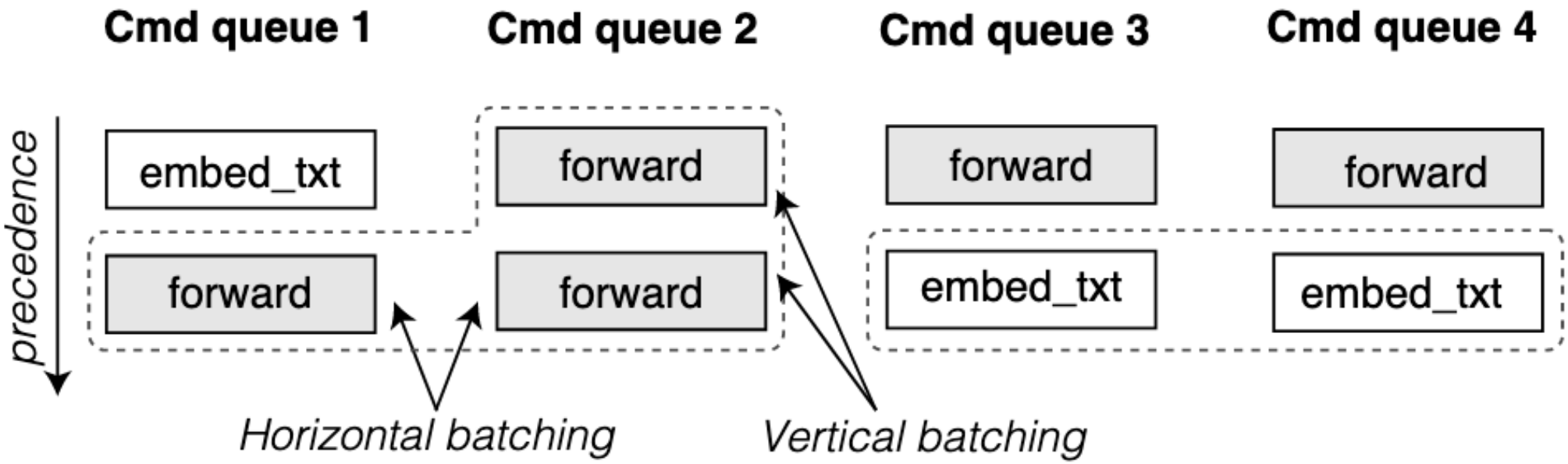
PIE 系统设计

控制层：系统的调度中枢

控制层核心功能二：自适应批处理调度

将来自多个异步运行的 Inferlet 的乱序 API 调用流进行实时重组

- 垂直合并：识别并合并单个 Inferlet 内部连续且无依赖的指令，减少通信开销
- 水平合并：跨不同请求对齐同类型的 API 调用，将其组装成大批量的 GPU 任务



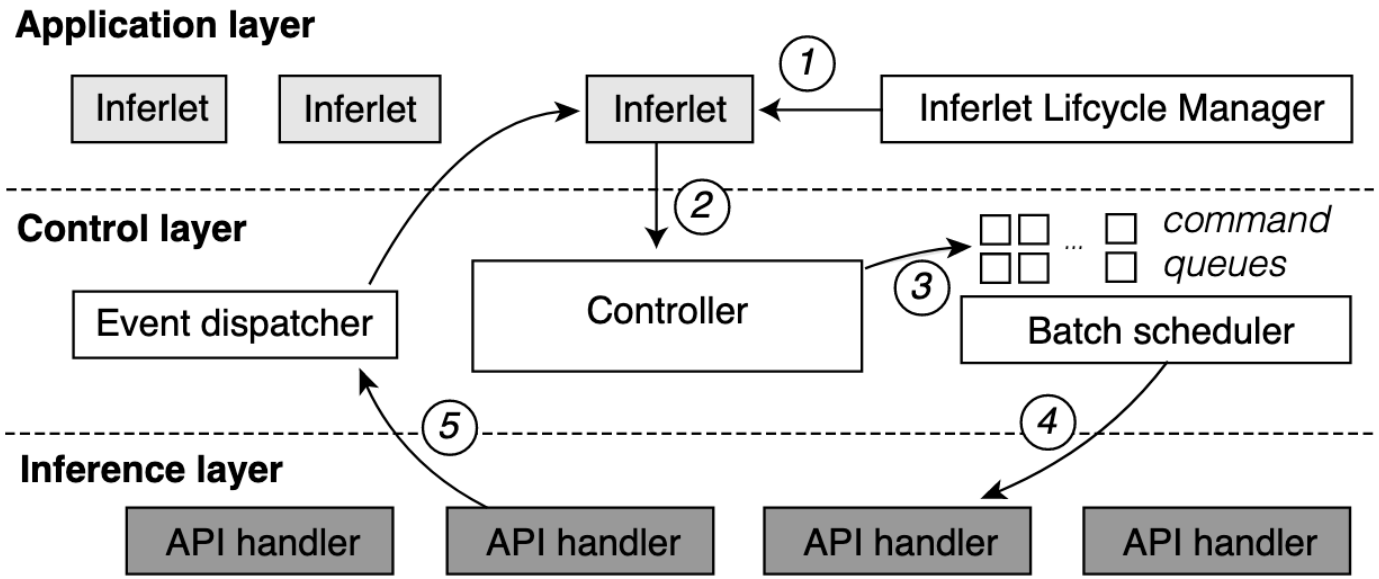
PIE 系统设计

控制层：系统的调度中枢

控制层核心功能三：异步分发与事件分拣

作为执行中转站，将打包好的批次分发给底层的硬件 Handlers，并分拣计算结果

- 分发：根据批处理后的指令集，调用相应的算子处理器如 forward handler
- 异步回传：当 GPU 计算完成，控制层通过事件驱动机制将结果如 Logits 分布精准地推回给最初发起申请的特定 Inferlet 实例



PIE 系统设计

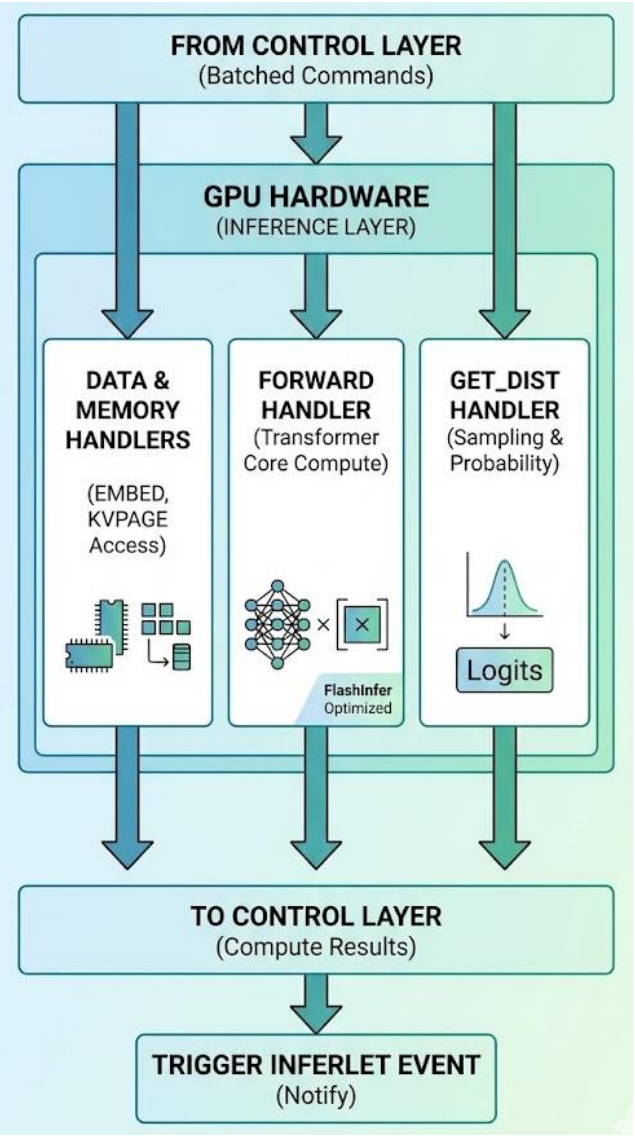
推理层：高性能执行后端

接收控制层分发的批处理指令并在硬件上执行

PIE 将大模型的推理过程解耦为一系列独立的算子（Handlers），每个算子对应应用层的一个或多个 API

- Forward Handler：执行 Transformer 的核心计算
- Get_Dist Handler：负责采样得到概率分布
- Data & Memory Handler：处理 Embed 以及物理显存

Handler 执行完毕后，会将结果返回给控制层，触发事件通知 Inferlet



章节介绍

- 作者介绍
- 研究背景与动机
- PIE 系统设计
- **实验评估**
- 总结与讨论

实验评估

实验设置与基准对比

硬件环境

- GCP VM G2 instance, NVIDIA L4 GPU (24GB)

测试模型

- Meta Llama-3 系列 (1B, 3B, 8B)

基线

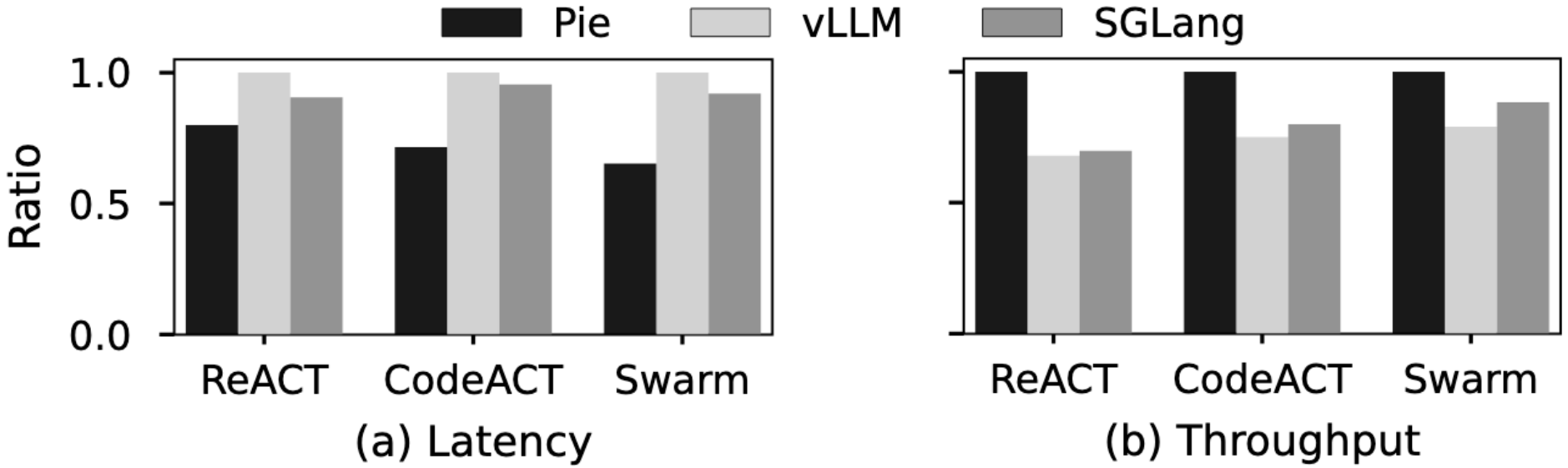
- vLLM:
- SGLang
- vLLM+LMQL

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	

实验评估

实验设置与基准对比

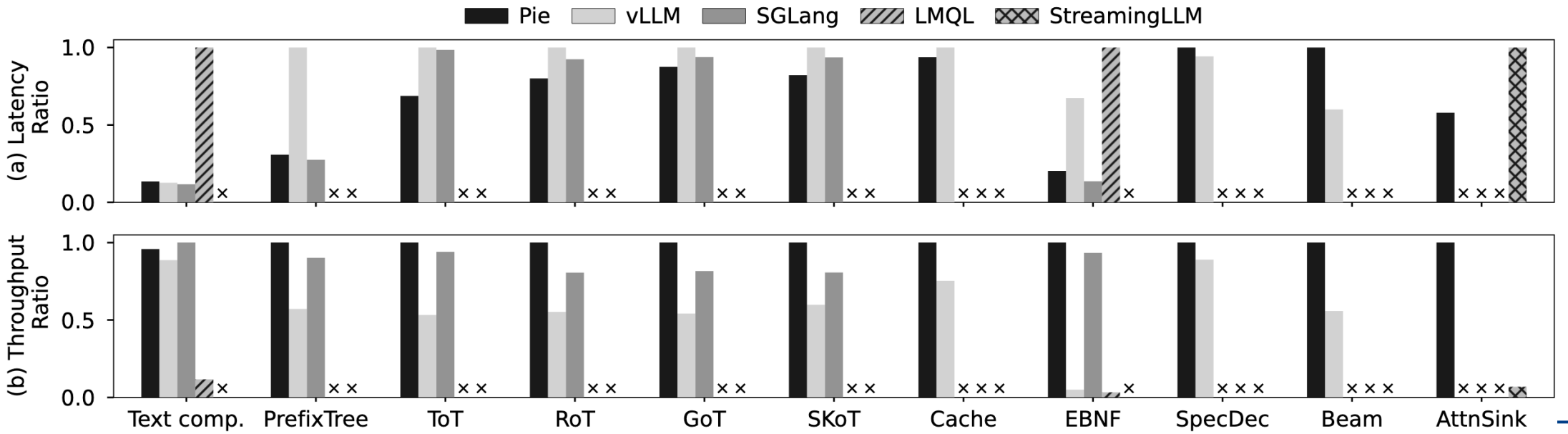
- **延迟对比：**在 ReACT、CodeACT 和 Swarm 三种典型的 Agent 框架下，PIE 的归一化延迟显著低于 vLLM 和 SGLang，特别是在 Swarm（多智能体协作）场景下，延迟优化最为明显
- **吞吐量对比：**PIE 的吞吐量实现了 1.3x 至 3.4x 的提升，相比之下，vLLM 和 SGLang 由于无法有效处理 I/O 等待，性能大幅受限



实验评估

实验设置与基准对比

复杂推理：PIE 在非线性推理任务中，延迟较 SGLang 降低了约 28%
约束生成：观察 EBNF 柱状图，LMQL的延迟开销巨大。而 PIE 原生支持的约束解码逻辑，在保持灵活性的同时，性能几乎没有损耗
自定义逻辑：开发者在 PIE 上用几行代码实现的投机采样（SpecDec），其性能达到了与原生硬编码实现相当的水平



实验评估

实验设置与基准对比

架构开销评估：

尽管引入了 Wasm 沙箱和三层解耦架构，但在主流的 8B 参数模型上，PIE 引入的额外延迟开销仅为 2.39%。

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%)

自适应调度收益：

对比不同的调度策略：Eager (无批处理) 模式下吞吐量仅为 5.61 reqs/s Adaptive (自适应批处理) 模式下吞吐量跃升至 84.85 reqs/s

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

章节介绍

- 作者介绍
- 研究背景与动机
- PIE 系统设计
- 实验评估
- **总结与讨论**

实验评估

一、能不能用到我们的场景

当前 Memory Agent 的相关工作存在的问题是：记忆在每轮对话中会被在线增删改查并动态重组，导致推理端反复处理“看似全新”的上下文；同时算法携带的层级/重要性/关系/作用域等语义被埋在应用逻辑里，底层推理引擎无法获取相关语义，难以做语义约束下的调度与一致性管理，导致大量的重复计算，增大延迟

我们借鉴 PIE 的两点核心思想来切入：统一编程模型/API（让记忆算法通过 API 实现从而把记忆需求与语义显式暴露）+ 中间层 IR 编译（把这些语义编译成可执行的运行时计划），用“可编译语义接口”打通 Agent Memory 与 LLM Serving 系统之间的断层

总结与讨论

二、PIE 研究的问题还可以进一步泛化吗

PIE 提出的这种解构主义思想，本质上是从“为特定算法设计专用执行器”向“为复杂推理流设计通用操作系统”的范式转移。在 LLM 领域，这种思想将原本封闭的预填与解码循环拆解为嵌入、前向传播和采样等独立算子，而这种“原子化能力+用户编排”的模式在跨模态推理和具身智能中同样具有巨大的泛化潜力。目前的机器人控制也受限于固定的感知-执行循环，这种将硬件资源的所有权和调度权从系统内核下放到应用层程序的方法，解决了复杂 AI 应用中长路径交互带来的通信开销和延迟瓶颈

实验评估

三、PIE 当前的工作还能不能进一步提高

自动化优化与编译：
目前 Inferlet 的编写仍需要开发者手动平衡 forward 和资源分配，这有一定的门槛，提升点，可以引入编译器优化技术，让系统自动分析 Inferlet 的逻辑，自动进行指令重排、算子融合或预测性缓存

硬件感知的动态调度：
目前的自适应批处理主要在软件逻辑层面，在生产环境下，计算集群普遍存在层次化存储（GPU HBM / Host RAM）与异构算力（H100 / L4 / CPU）的复杂组合，如何做到资源的层次化存储管理和异构算力的负载均衡调度