Serve Programs, Not Prompts

In Gim and Lin Zhong

Yale University

Future LLM serving systems should

Serve Programs, Not Prompts

To efficiently support emerging AI applications

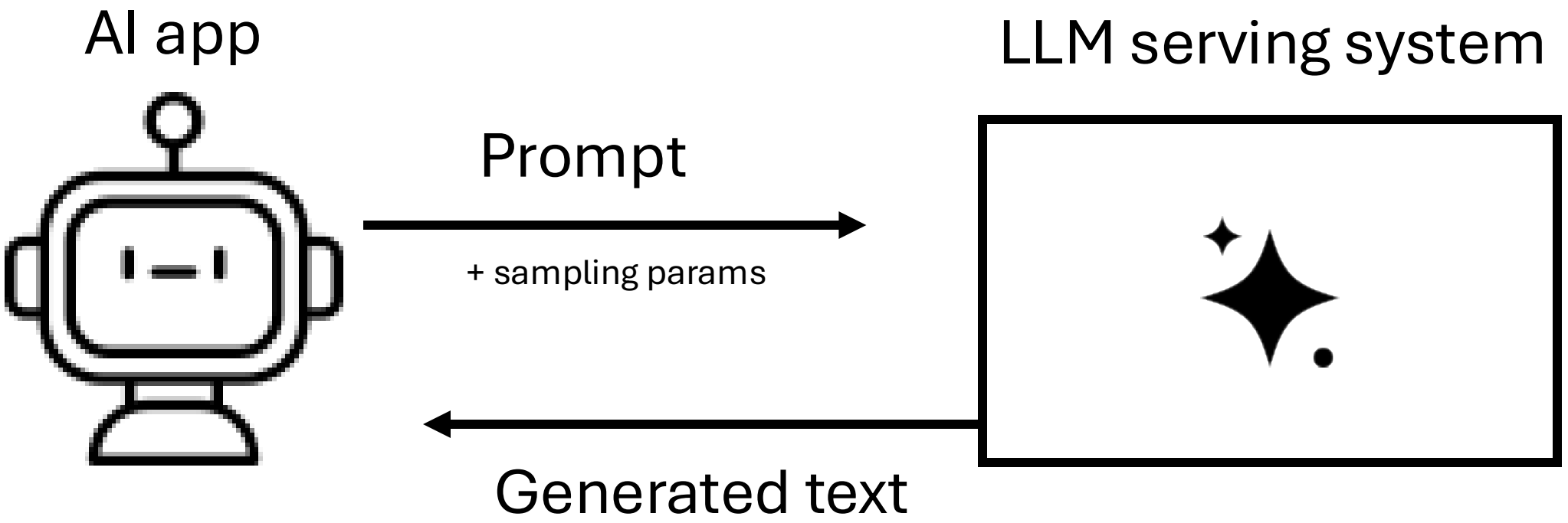
In Gim and Lin Zhong

Yale University

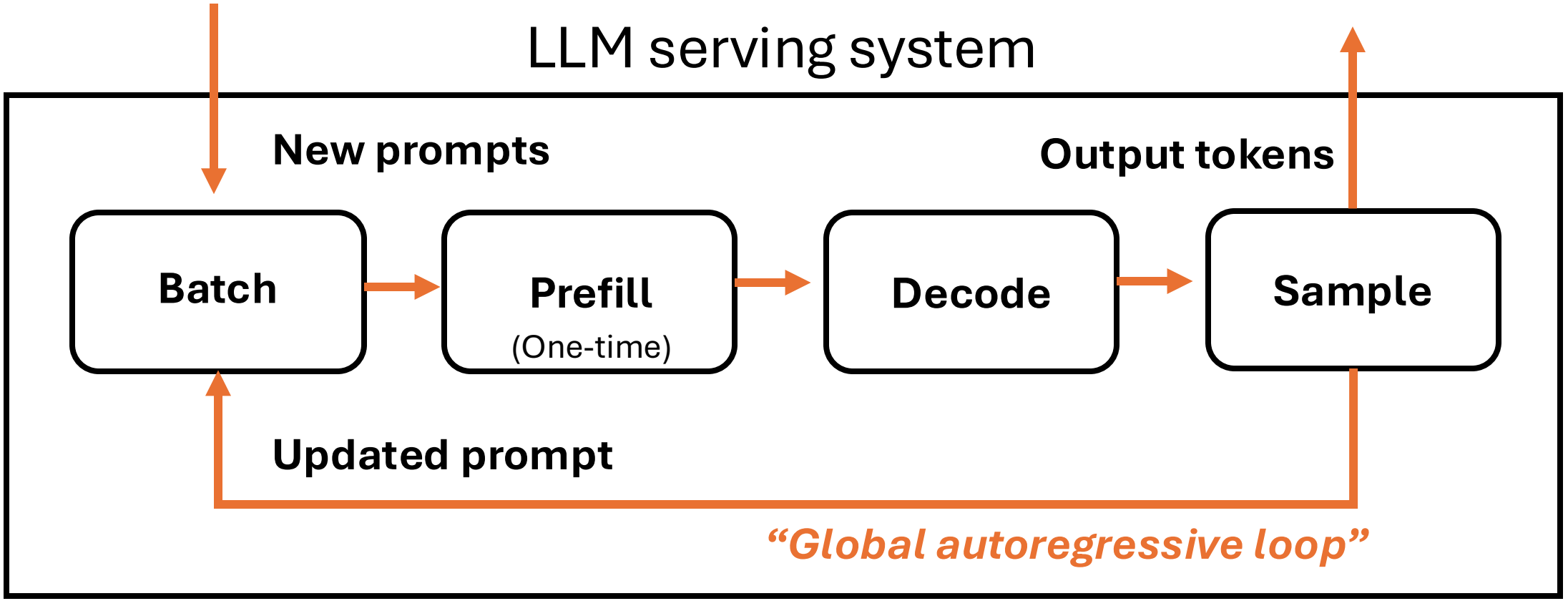
The only API you need to build a million-dollar startup

```
curl https://api.openai.com/v1/responses \
  -d "Write a one-sentence bedtime story about a unicorn."
```

Today's LLM serving systems are designed for text completion

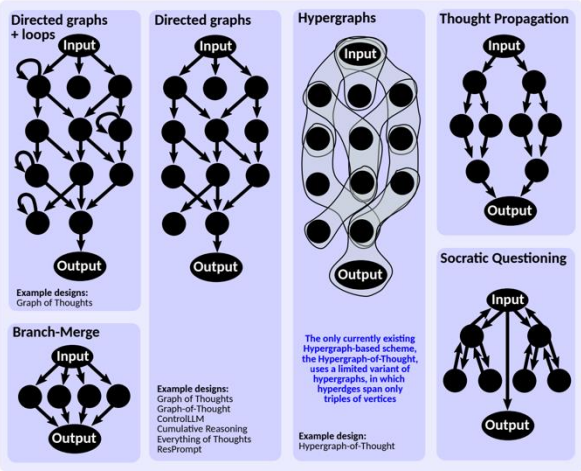


Today's LLM serving systems are designed for text completion



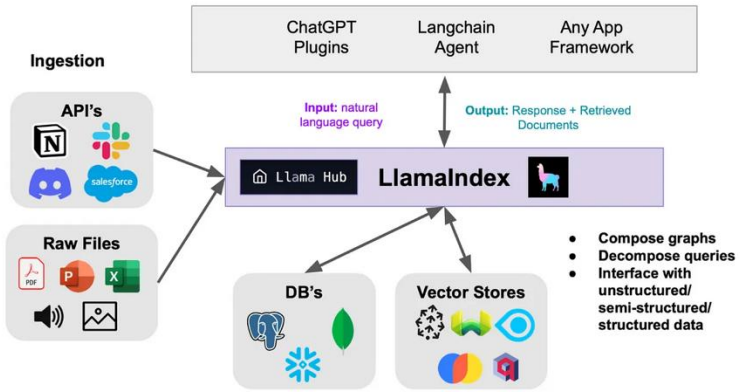
LLM usages are evolving beyond simple text completion

Non-linear generation



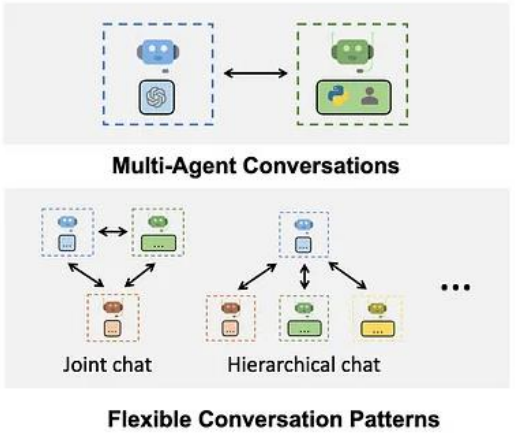
E.g., Graph-of-Thought

Augmented generation



E.g., LlamaIndex

Agentic problem solving



E.g., AutoGen

LLM usages are evolving beyond simple text completion

Non-linear generation

Augmented g

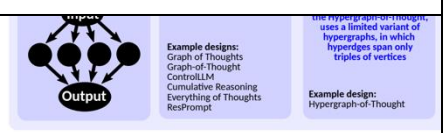
Type-Constrained Code Generation with Language Models

NIELS MÜNDLER*, ETH Zurich, Switzerland
JINGXUAN HE*, UC Berkeley, USA
HAO WANG, UC Berkeley, USA
KOUSHIK SEN, UC Berkeley, USA
DAWN SONG, UC Berkeley
MARTIN VECHEV, ETH Z

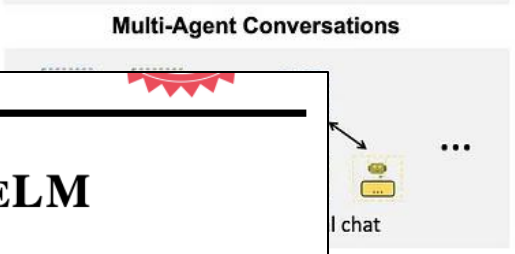
Language Model Cascades: Token-level uncertainty and beyond

krishna Narasimhan Wittawat Jitkrittum
Aditya Krishna Menon Sanjiv Kumar

ogle Research, New York



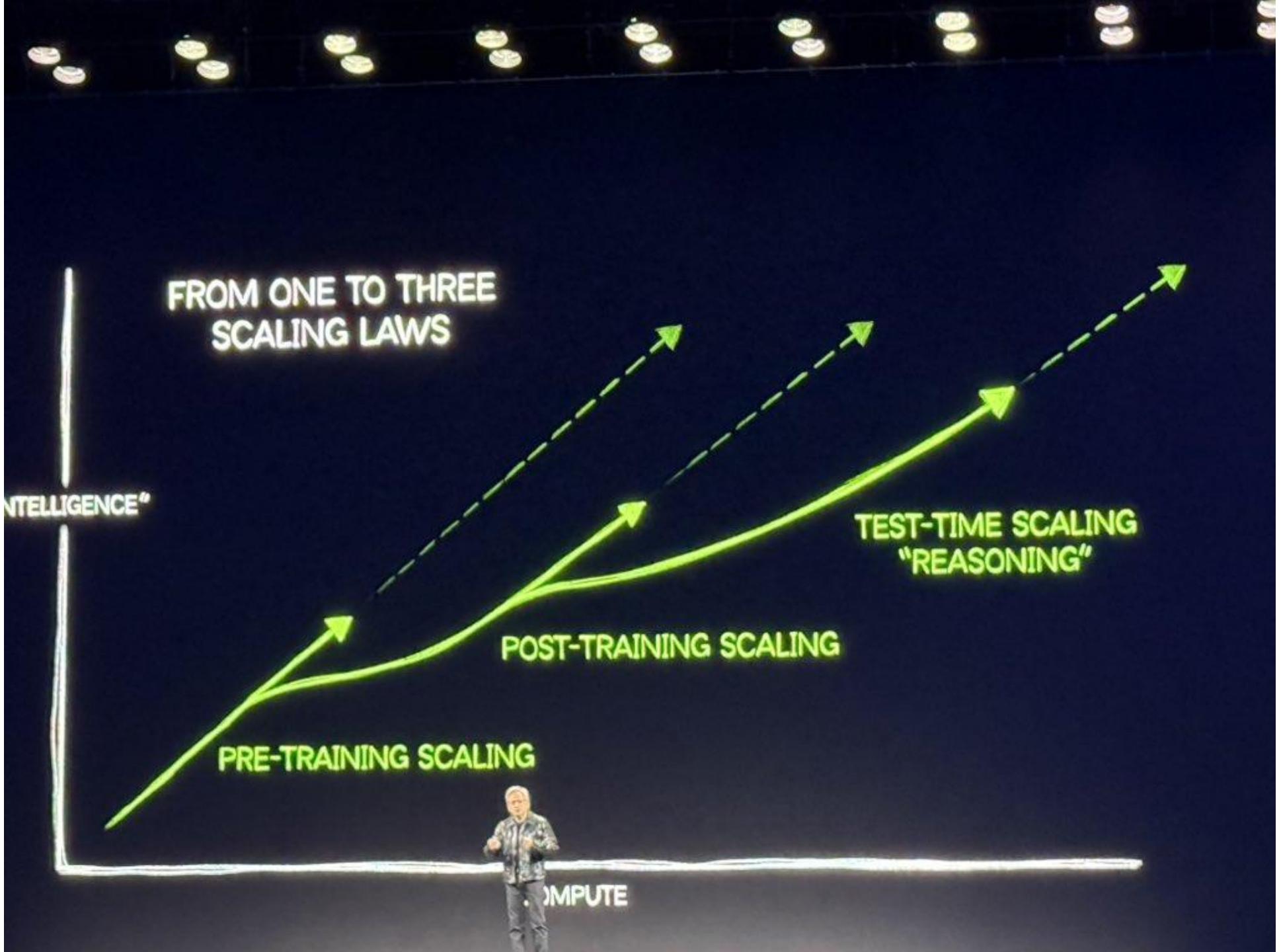
E.g., Graph-of-Thou



oGen

VALIDATING LARGE LANGUAGE MODELS WITH RELM

Michael Kuchnik [1] Virginia Smith [1] George Amvrosiadis [1]


FROM ONE TO THREE
SCALING LAWS
NTELLIGENCE"
TEST-TIME SCALING
"REASONING"
POST-TRAINING SCALING
PRE-TRAINING SCALING
OMPUTE

The disconnect between LLM applications and serving systems

1. **Inference** inefficiency from missing application-level optimizations
2. **Integration** friction with the external data and tools
3. **Implementation** challenges for the custom generation workflows

The only API you need to build a million-dollar startup?

```
curl https://api.openai.com/v1/responses \
  -d "Write a one-sentence bedtime story about a unicorn."
```

Imagine building an AI code completion tool

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

class Sequence(nn.Module):
    def __init__(self):
        super(Sequence, self).__init__()
        self.lstm1 = nn.LSTMCell(1, 51)
        self.lstm2 = nn.LSTMCell(51, 51)
        self.linear = nn.Linear(51, 1)

    def forward(self, input, future = 0):
        outputs = []
        h_t = torch.zeros(input.size(0), 51, dtype=torch.double)
```

Challenge 1. Inference efficiency

Scenario: User's input updates the suggestion.

Prompt =

System prompt

Your job is to predict the most probable next line of code given the …

+

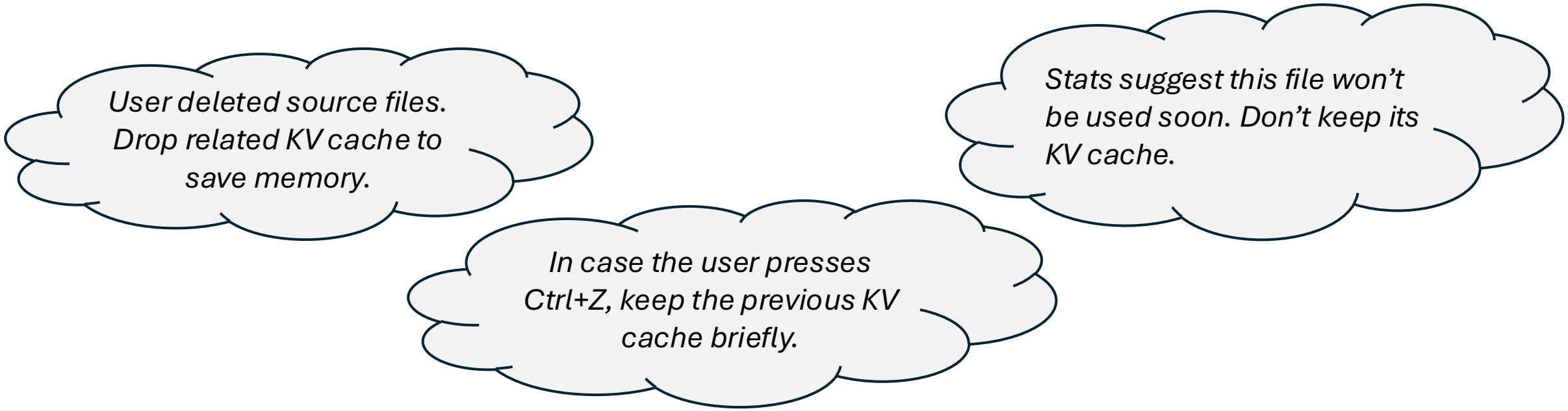
Context
(source code files)

Due to the large overlaps in prompts,
Inter-request KV cache reuse is essential for efficiency

Challenge 1. Inference efficiency

KV cache management is often automatic, system-wide policy
(e.g., OpenAI's prompt caching, vLLM's automatic prefix caching)

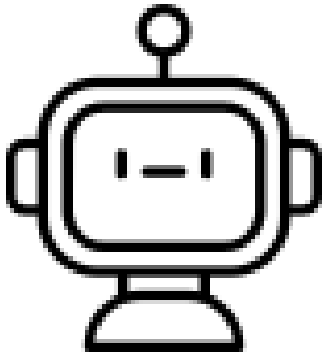
This precludes application-driven optimizations



Prompt Cache: Modular Attention Reuse for Low-Latency Inference [Gim et al., MLSys 24]

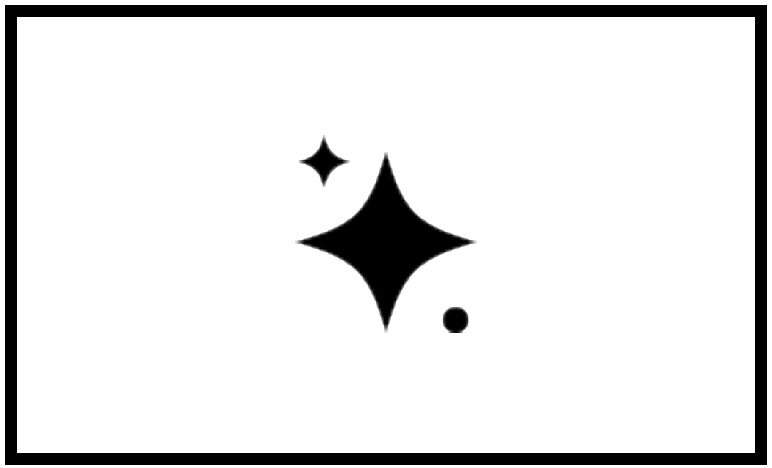
Challenge 2. Integration efficiency

Scenario: Augment generation with relevant API documentation



Prompt: “If unsure about an API during generation, output **<read>API name</read>** and pause”

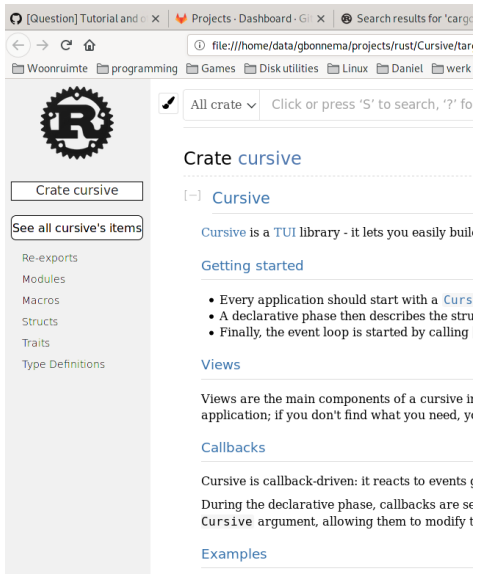
LLM serving system

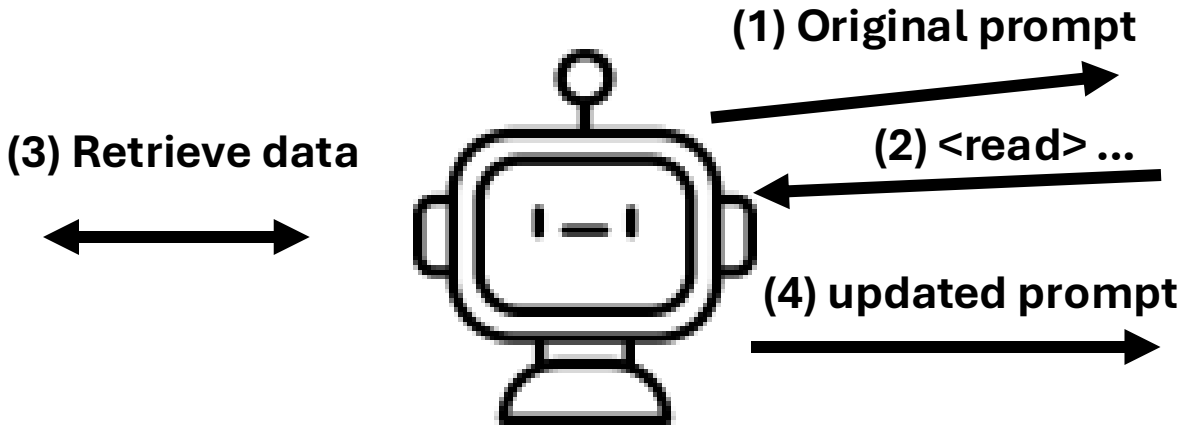


Challenge 2. Integration efficiency

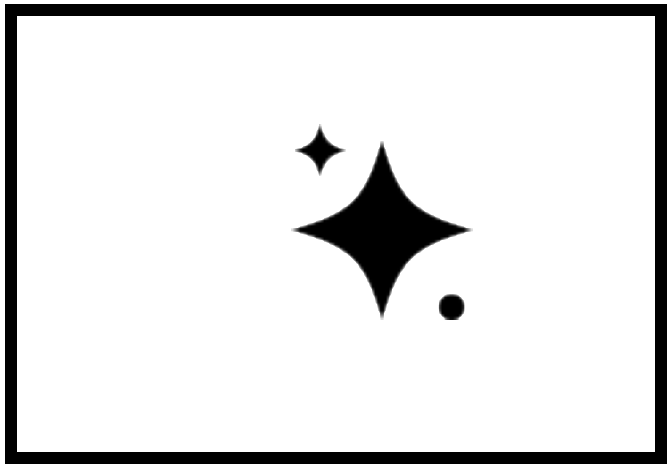
Each external interaction induces two extra boundary crossings

Data/Tool



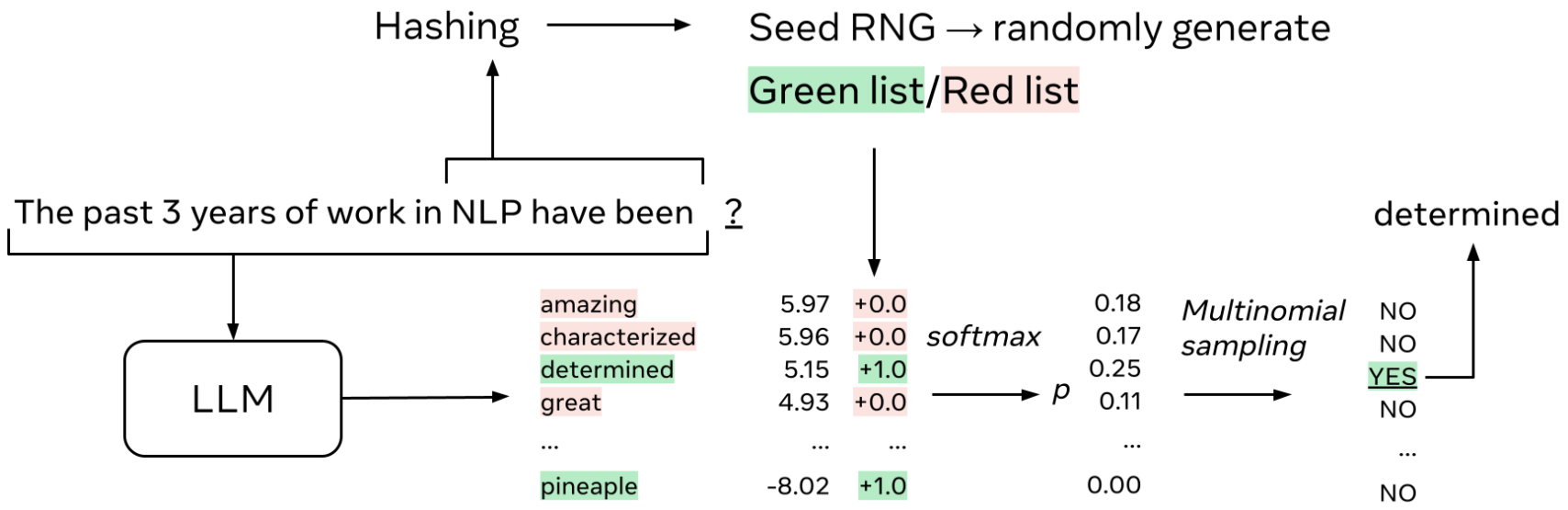


LLM serving system



Challenge 3. Implementation efficiency

Scenario: Watermark output suggestions for IP protection

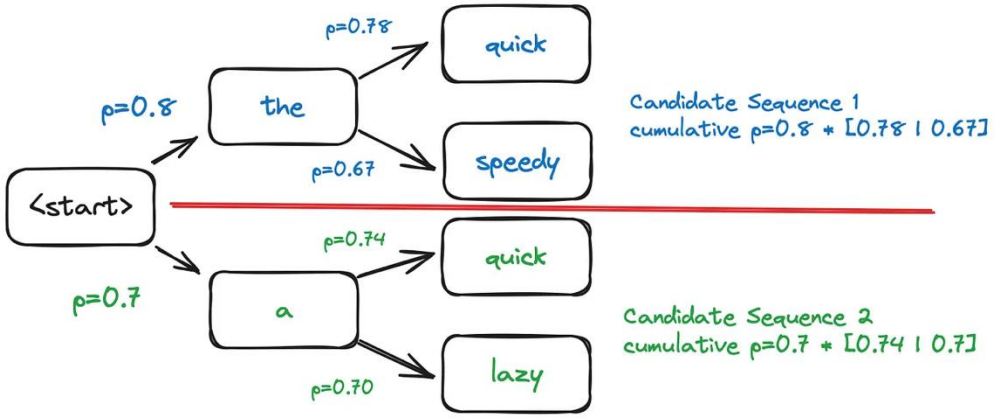


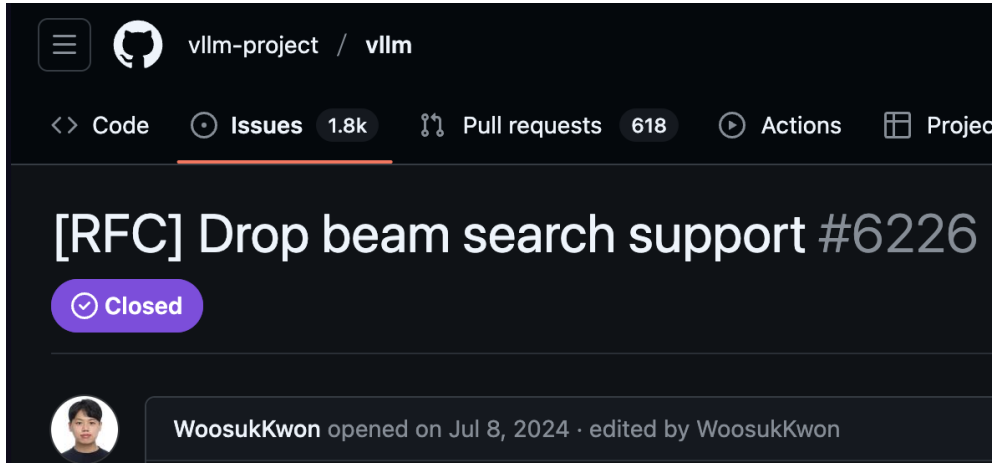
Watermarking Makes Language Models Radioactive [Sander et. al., NeurIPS 24]

Challenge 3. Implementation efficiency

Current LLM serving systems confound ***generation process*** and ***model computation*** by employing a global autoregressive loop.

This makes it difficult to implement & manage generation strategies that deviates from the usual generation process (e.g., beam search)





Challenge 3. Implementation efficiency

Optimizations are global policy

```
[--enable-chunked-prefill
 [--enable-prefix-caching
 [--disable-sliding-window
 [--disable-cascade-attn |
 [--skip-tokenizer-init | -
 [--enable-prompt-embeds |
 [--speculative-config SPECULATIVE_CONFIG]
```

LLM serving paradigm should change

Model-centric

How fast (and how many) can it generate new tokens?

Application-centric

How efficiently does it support the application logic?

Our vision: “programmable” LLM serving system

We have seen similar problems/solutions in networking, OS, and security before:

- eBPF
- SDN
- OPA

Can we take the same philosophy in building the LLM serving system?

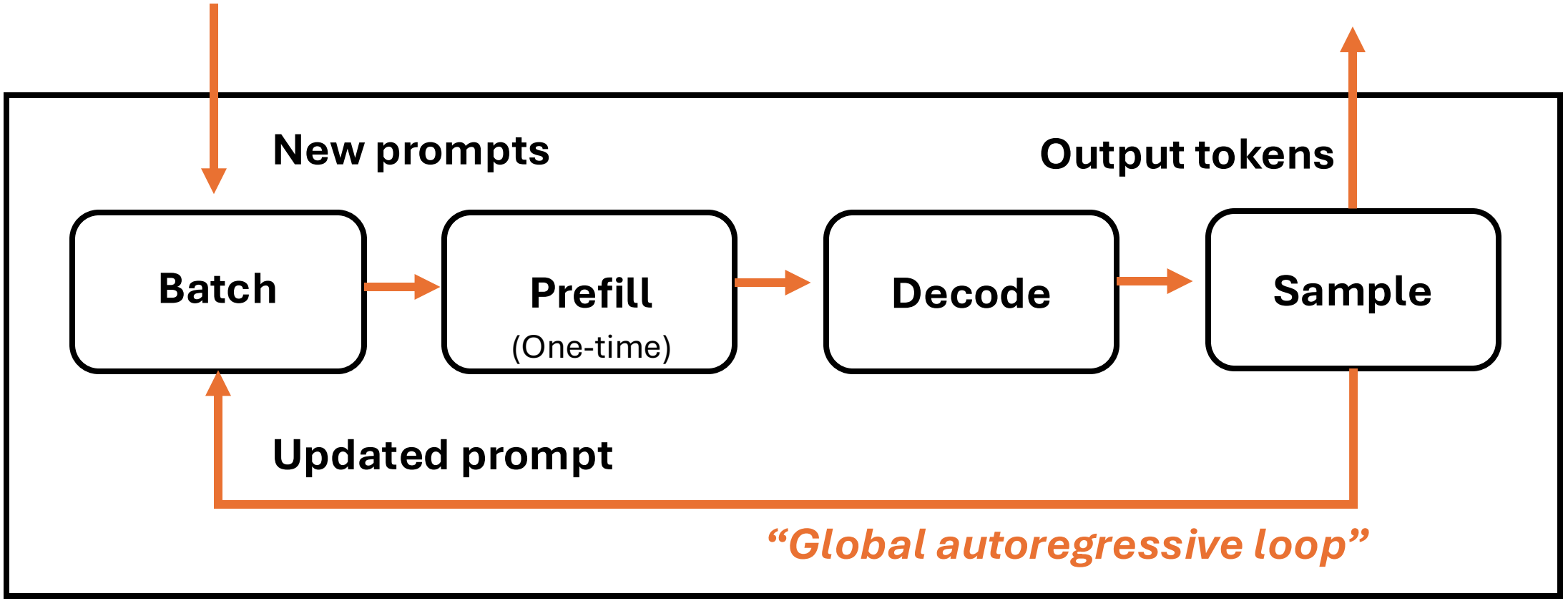
Our proposal

Symphony is a *programmable* LLM serving system that:

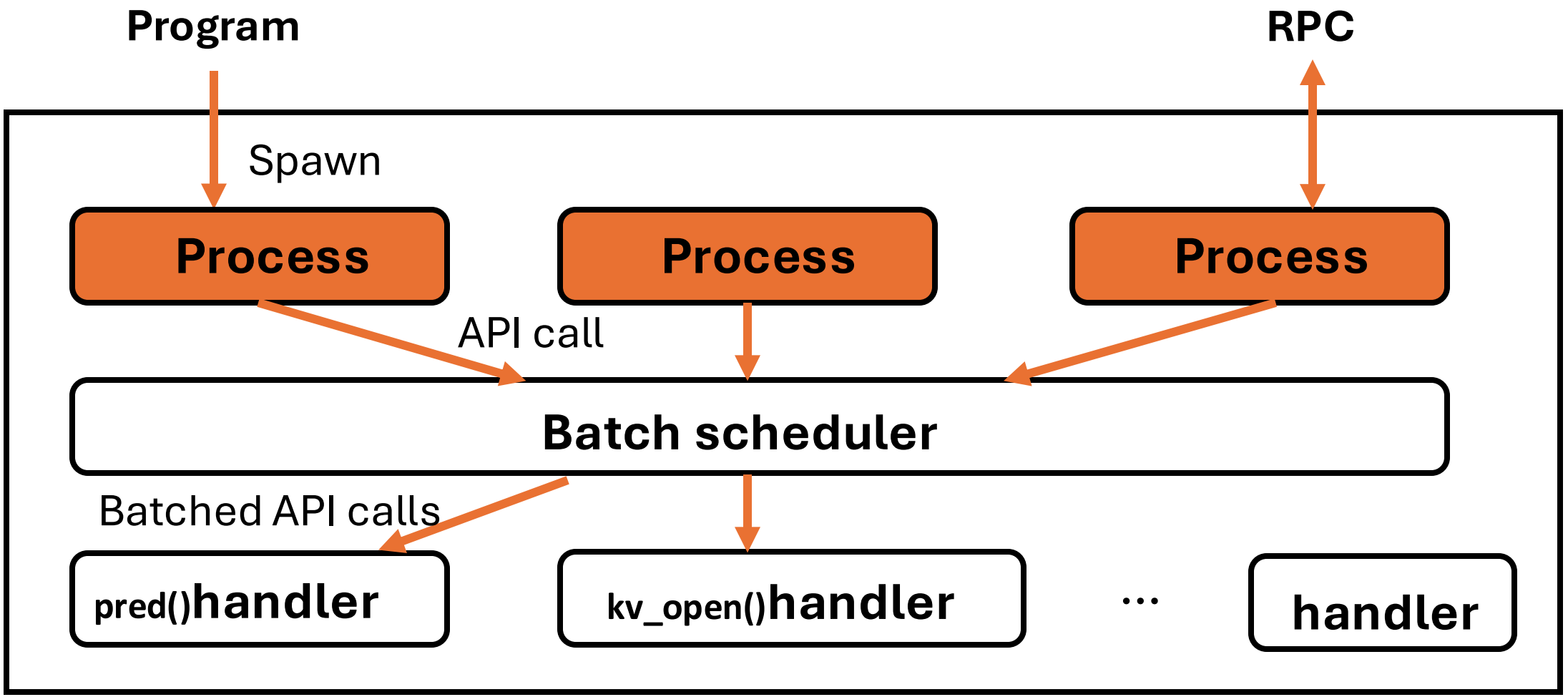
(1) **Decouples** control (e.g., autoregressive loop) and the model computation from the generation process

(2) **Delegates** the control entirely to the user-provided programs, which we refer to as *LLM Inference Programs (LIPs)*

Prompt-serving architecture



Symphony architecture



LLM Inference Programs (LIPs)

(Some) APIs provided by Symphony:

```
// model computation
pred(kv_cache, tok_ids, …)

// KV cache management
kv_open
kv_create
kv_remove

// tokenization
tokenize

// RPC
send_to_user
```

LIP code for basic autoregressive decoding

```
prompt = "Hello, how are you?"

kv = kv_create()
tok_ids = tokenize(prompt)

for (i = 0; i < MAX_TOKENS; i++) {
    dist = pred(kv, tok_ids, …)
    sampled = argmax(dist)
    send_to_user(detokenize(sampled))
    tok_ids = [sampled]
}
kv_remove(kv)
```

Inference efficiency

➔LIPs customize KV cache management per application

Interaction efficiency

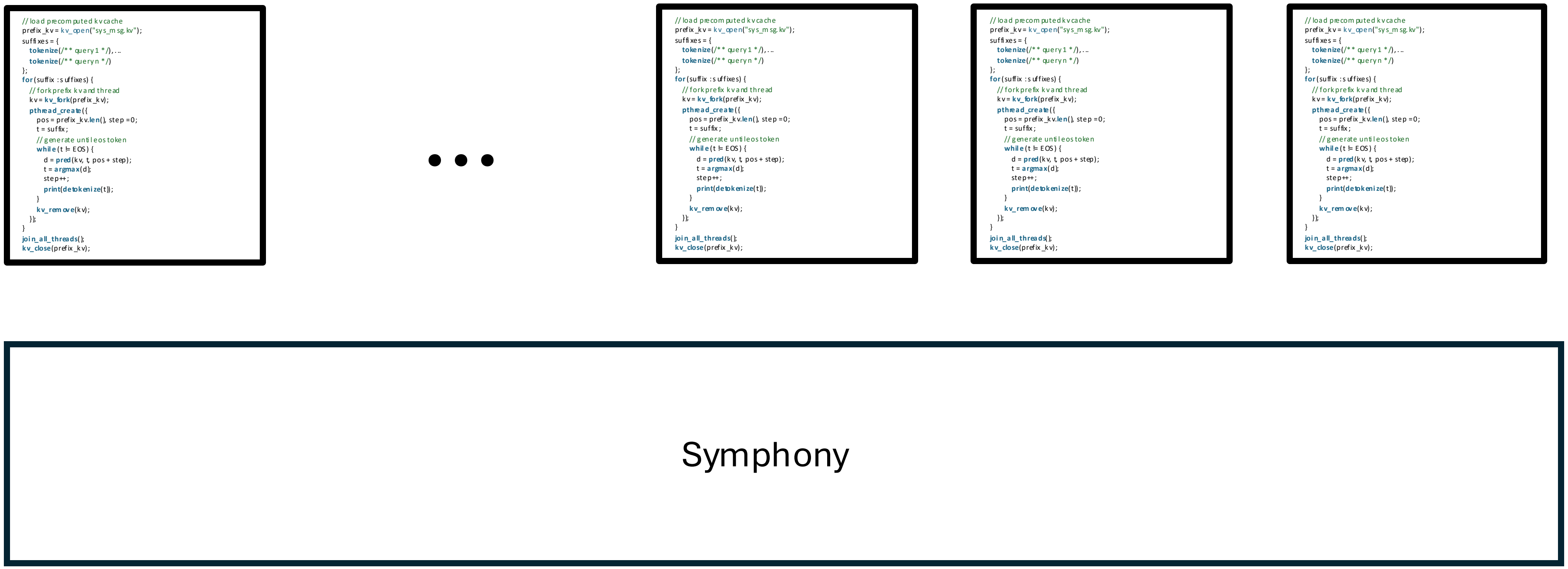
➔LIPs integrate arbitrary I/O and compute within the generation

Implementation efficiency

➔LIPs define novel decoding strategies without modifying the serving system

LIP code for prefix caching + parallel decoding

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


// load precomputed kv cache
prefix_kv = kv_open("sys_msg.kv");
suffixes = {
tokenize(/** query1 */), ...
tokenize(/** query n */)
};
for (suffix : suffixes) {
// fork prefix kv and thread
kv = kv_fork(prefix_kv);
pthread_create({
pos = prefix_kv.len(), step = 0;
t = suffix;
// generate until eos token
while (t != EOS) {
d = pred(kv, t, pos + step);
t = argmax(d);
step++;
print(detokenize(t));
}
kv_remove(kv);
});
}
join_all_threads();
kv_close(prefix_kv);
Symphony

LLM serving system as OS

Symphony extends the responsibility of the LLM serving system to:

(1) Abstracting resources (e.g., KV cache)

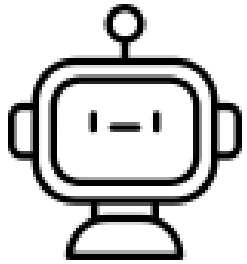
(2) Scheduling LIPs

(3) Handling I/O

We borrow familiar OS abstractions to describe Symphony's operation

AI app

Symphony

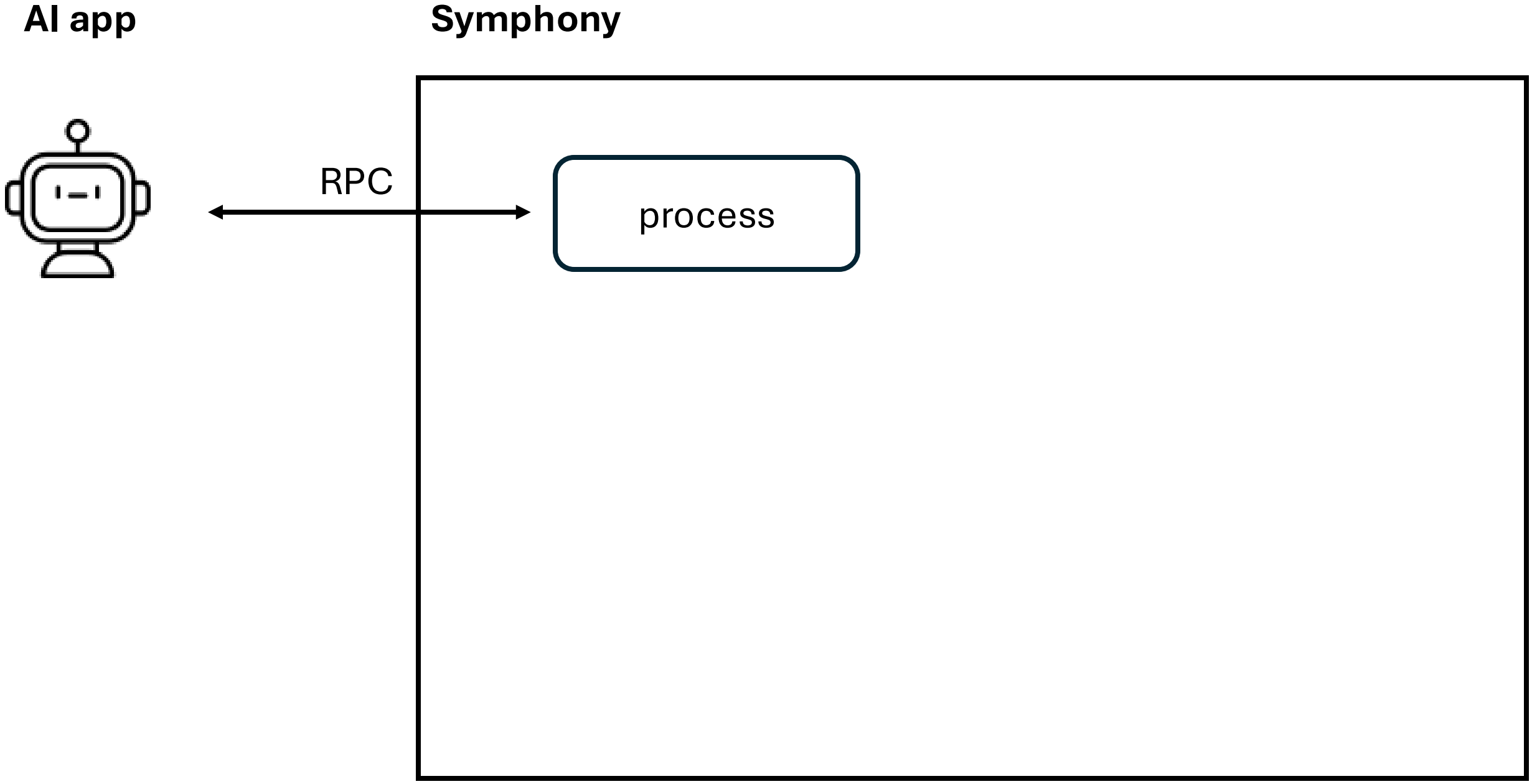


```
prompt = "Hello, how are you?"

kv = kv_create()
tok_ids = tokenize(prompt)

for (i = 0; i < MAX_TOKENS; i++) {
        dist = pred(kv, tok_ids, …)
        sampled = argmax(dist)
        send_to_user(detokenize(sampled))
        tok_ids = [sampled]
}
kv_remove(kv)
```

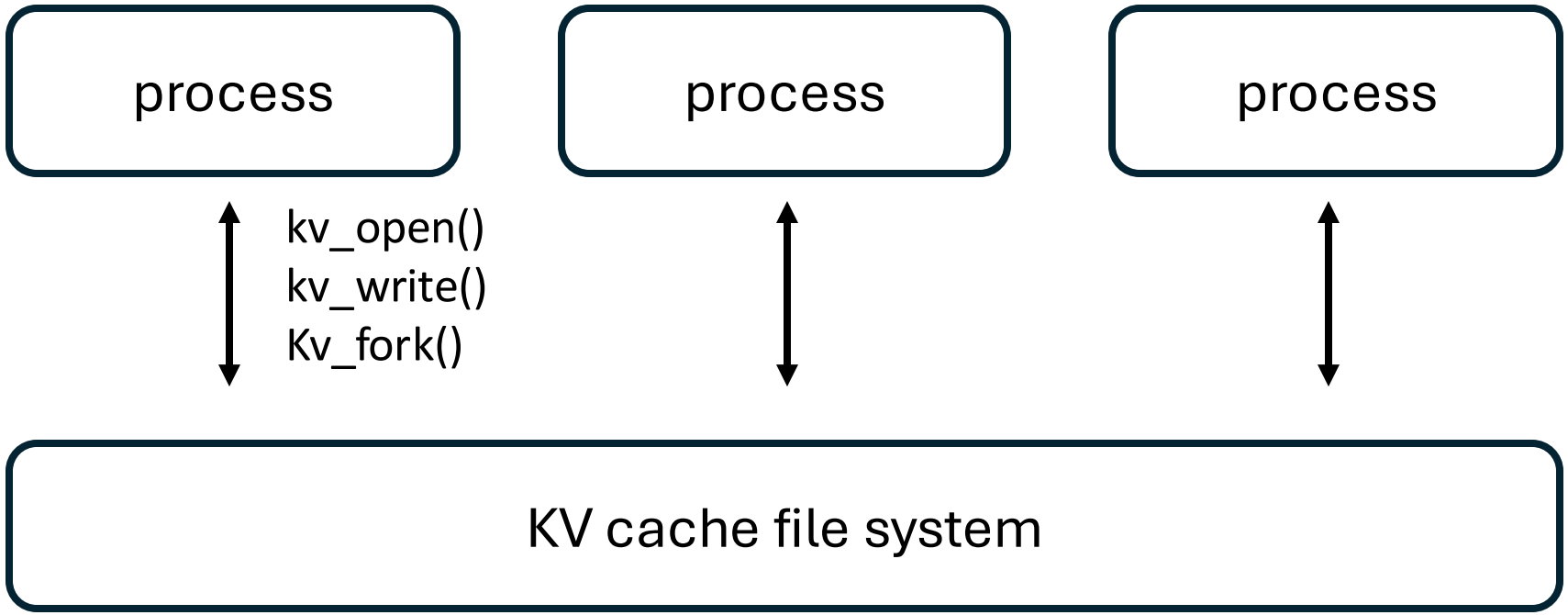
program

spawn

process

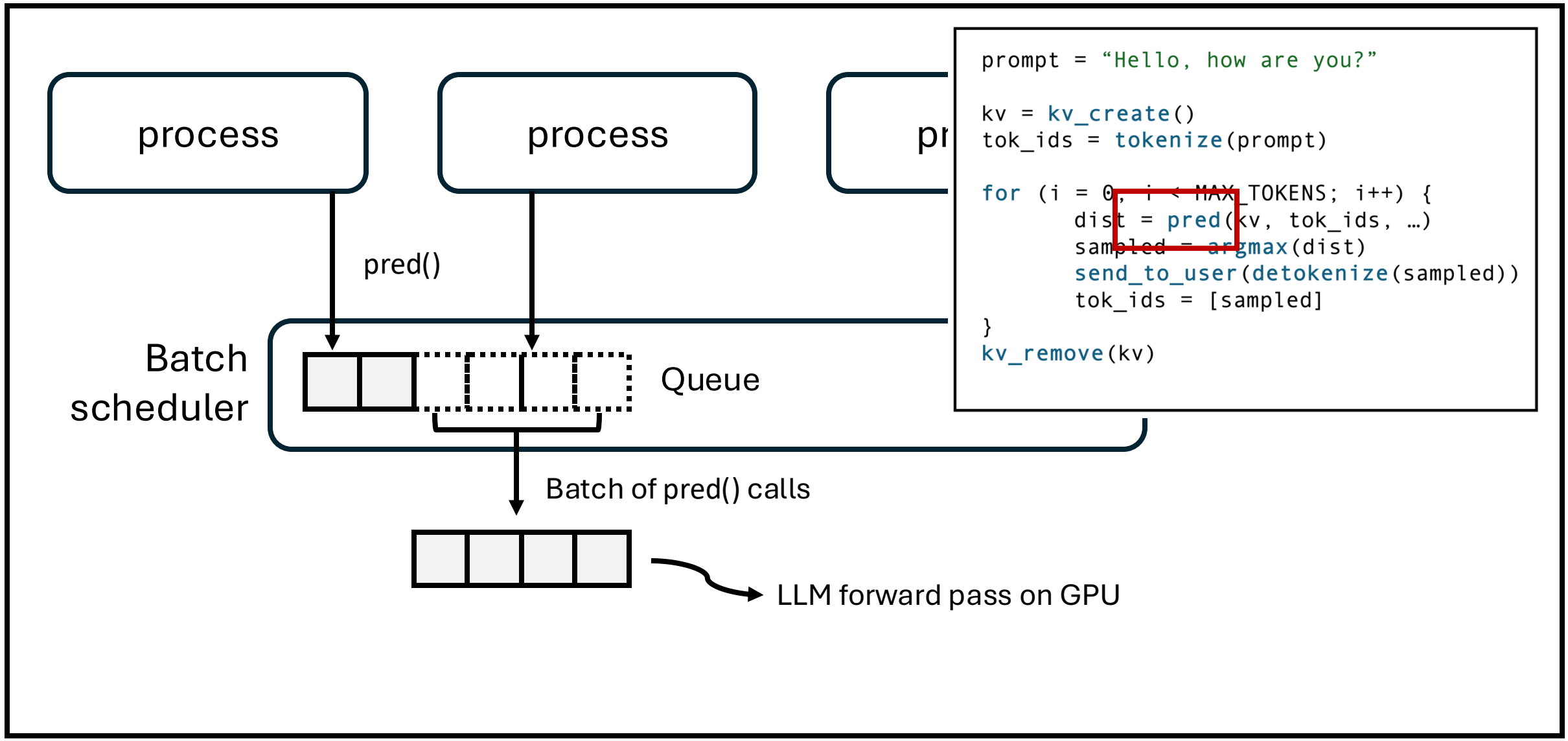

AI app
Symphony
RPC
process

Symphony



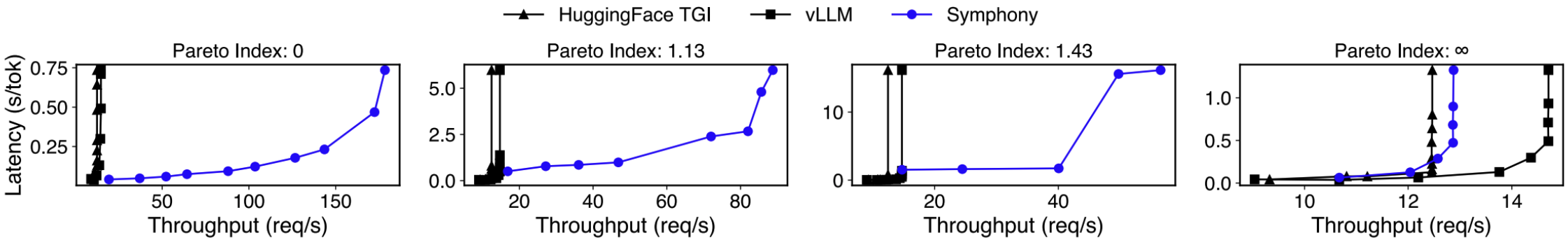
KV cache can be shared across processes

Symphony


process
process
pr
pred()
Batch scheduler
Queue
Batch of pred() calls
LLM forward pass on GPU
prompt = “Hello, how are you?”
kv = kv_create()
tok_ids = tokenize(prompt)
for (i = 0; i < MAX_TOKENS; i++) {
dist = pred(kv, tok_ids, …)
sampled = argmax(dist)
send_to_user(detokenize(sampled))
tok_ids = [sampled]
}
kv_remove(kv)

Preliminary results

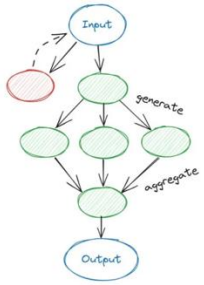
Benefits from application-specific optimization >> system overhead

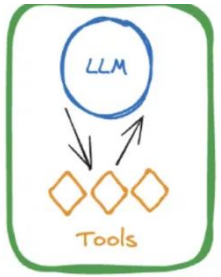


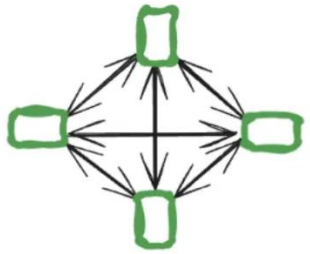
User can exploit workload characteristics (Pareto Index) for optimal performance

Preliminary results

LIPs can implement wide spectrum of emerging LLM applications







Text completion
-4%-11%

Graph-of-Thought
15%-43%

ReACT agent
24%-

SWARM agents
55%-

End-to-end throughput improvements (baseline: vLLM/SGLang)

Challenges and open questions

- Security - What if LIPs are malicious?

- API design - What is the right interface granularity?