Composable Effect Handling for Programming LLM-Integrated Scripts

Di Wang
Programming Languages Lab, Peking University

2025 / 10 / 15

What are the Differences

for **humans** writing code, since the LLM era?

What are the Differences

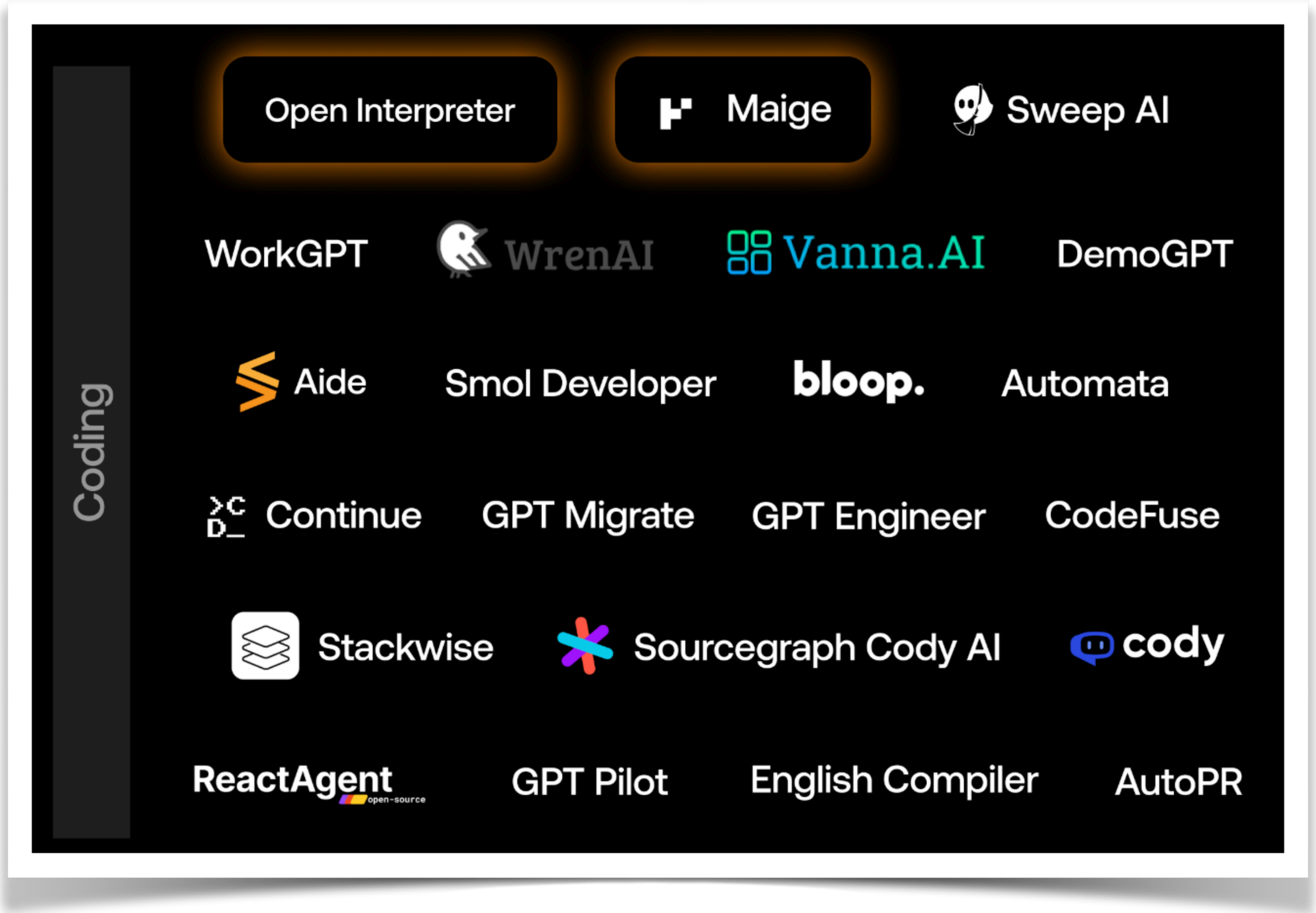
for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!

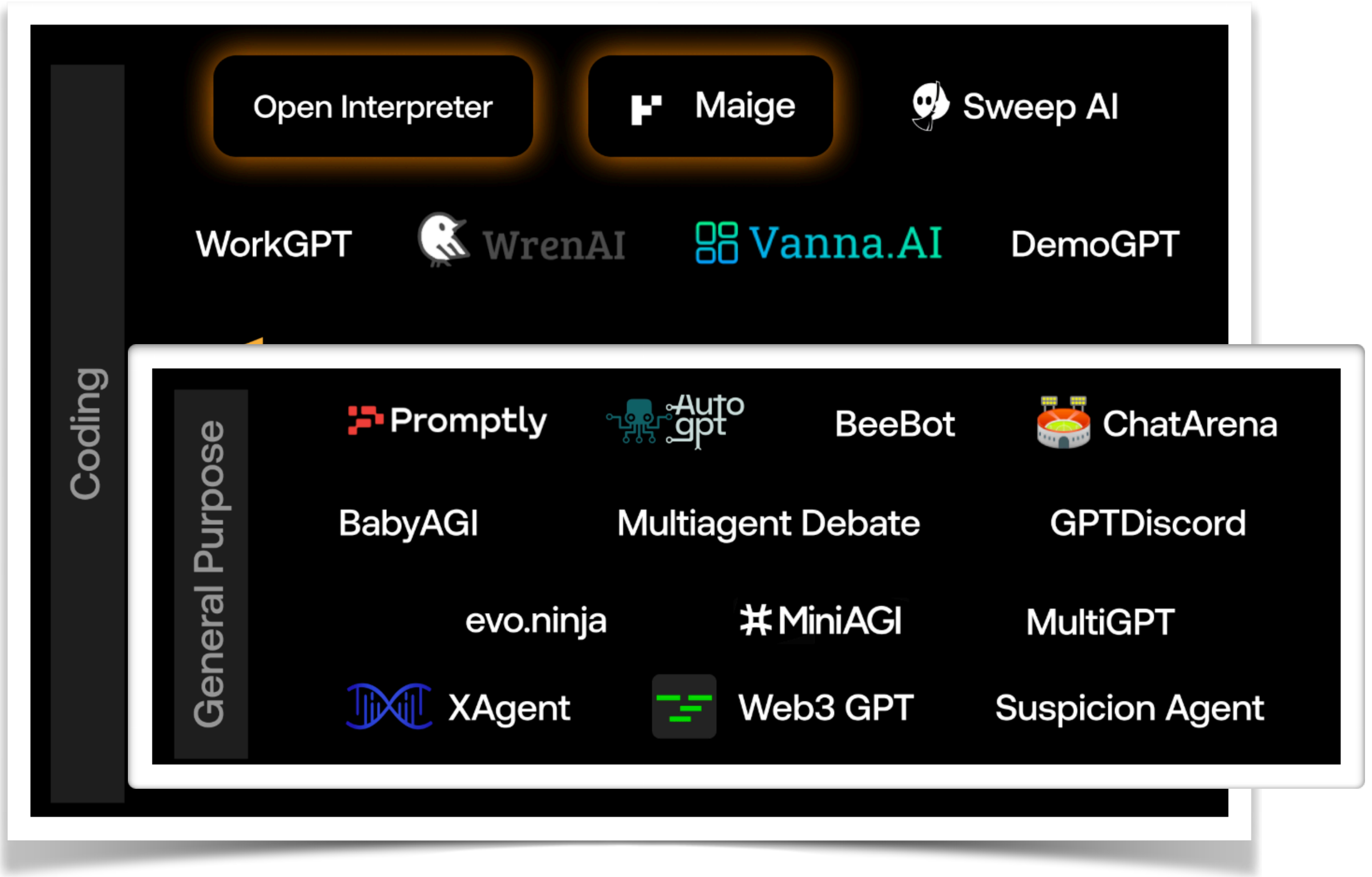


Source: https://github.com/e2b-dev/awesome-ai-agents

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!

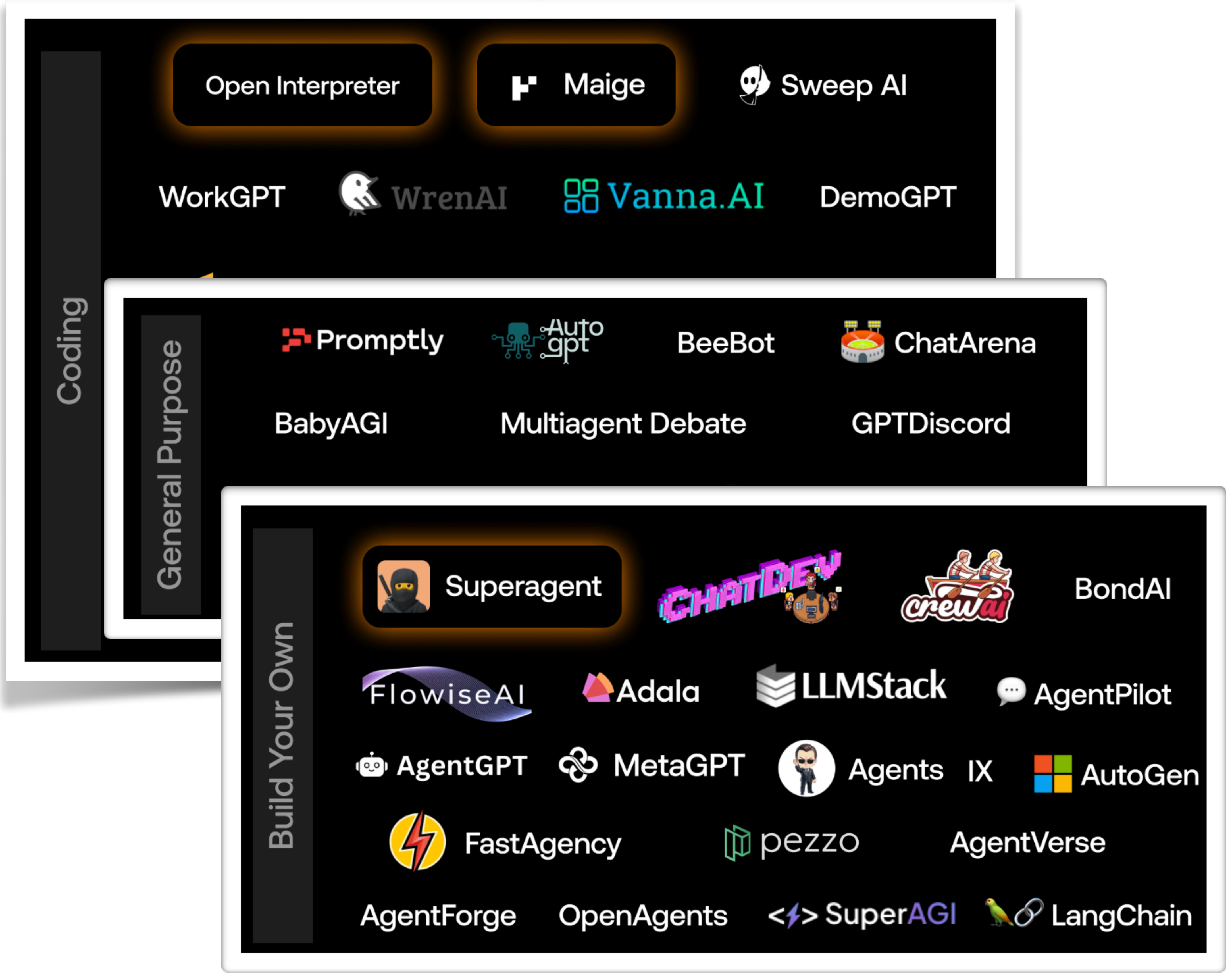


Source: https://github.com/e2b-dev/awesome-ai-agents

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!

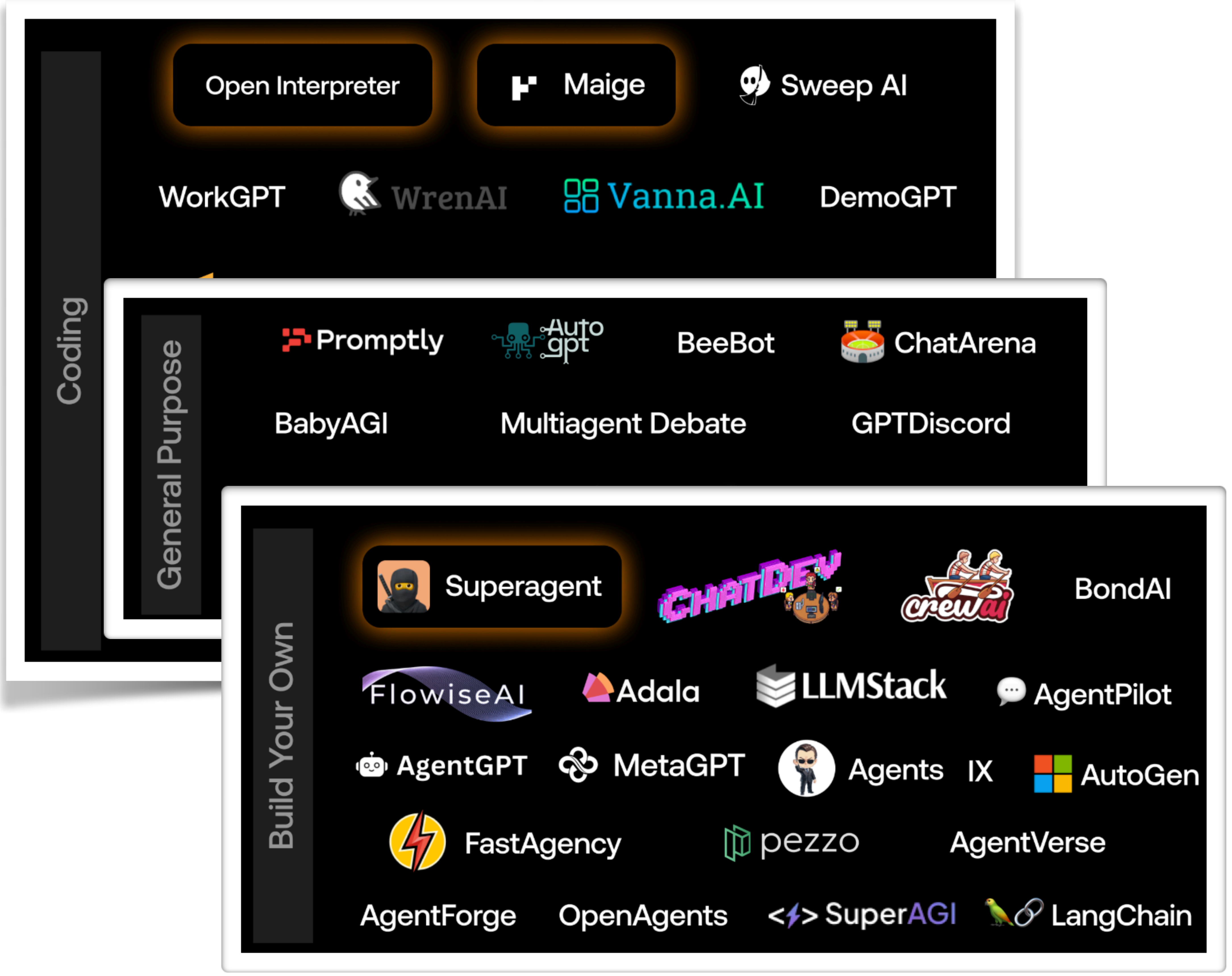


Source: https://github.com/e2b-dev/awesome-ai-agents

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!
  - or: **LLM-integrated** software

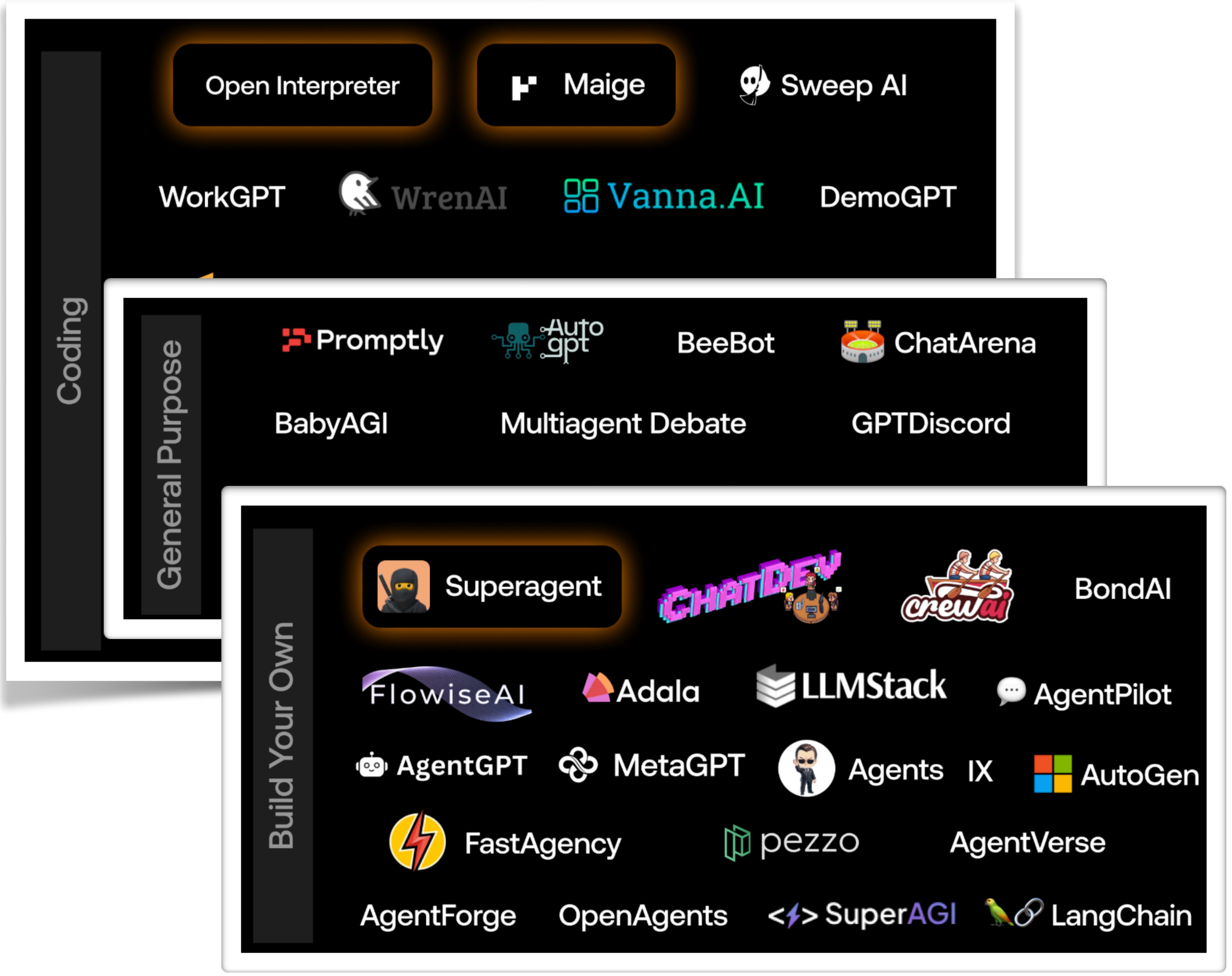


Source: https://github.com/e2b-dev/awesome-ai-agents

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!
  - or: **LLM-integrated** software
- Many of them are programmed by humans

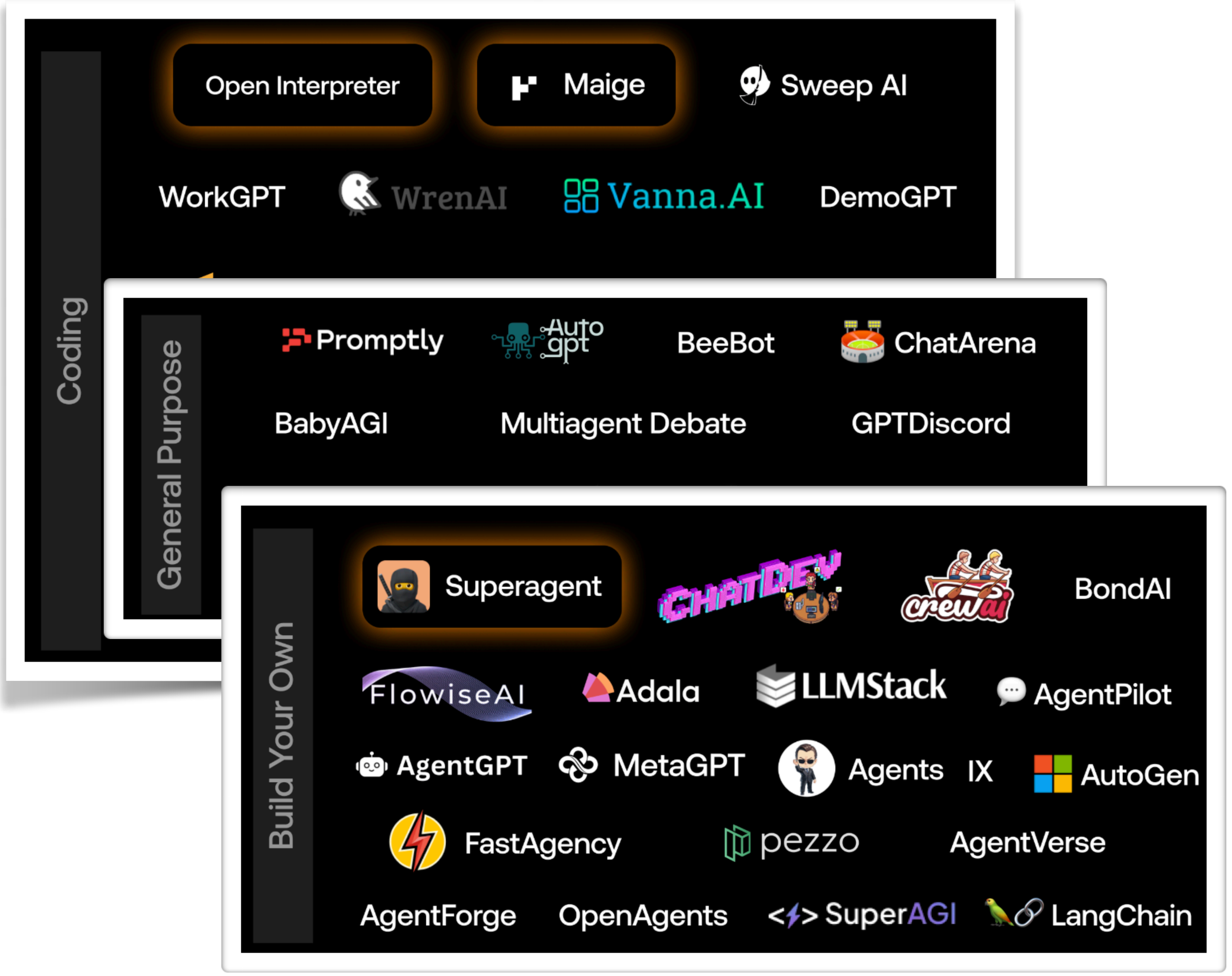


Source: https://github.com/e2b-dev/awesome-ai-agents

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!
  - or: **LLM-integrated** software
- Many of them are programmed by humans
  - at least for now …

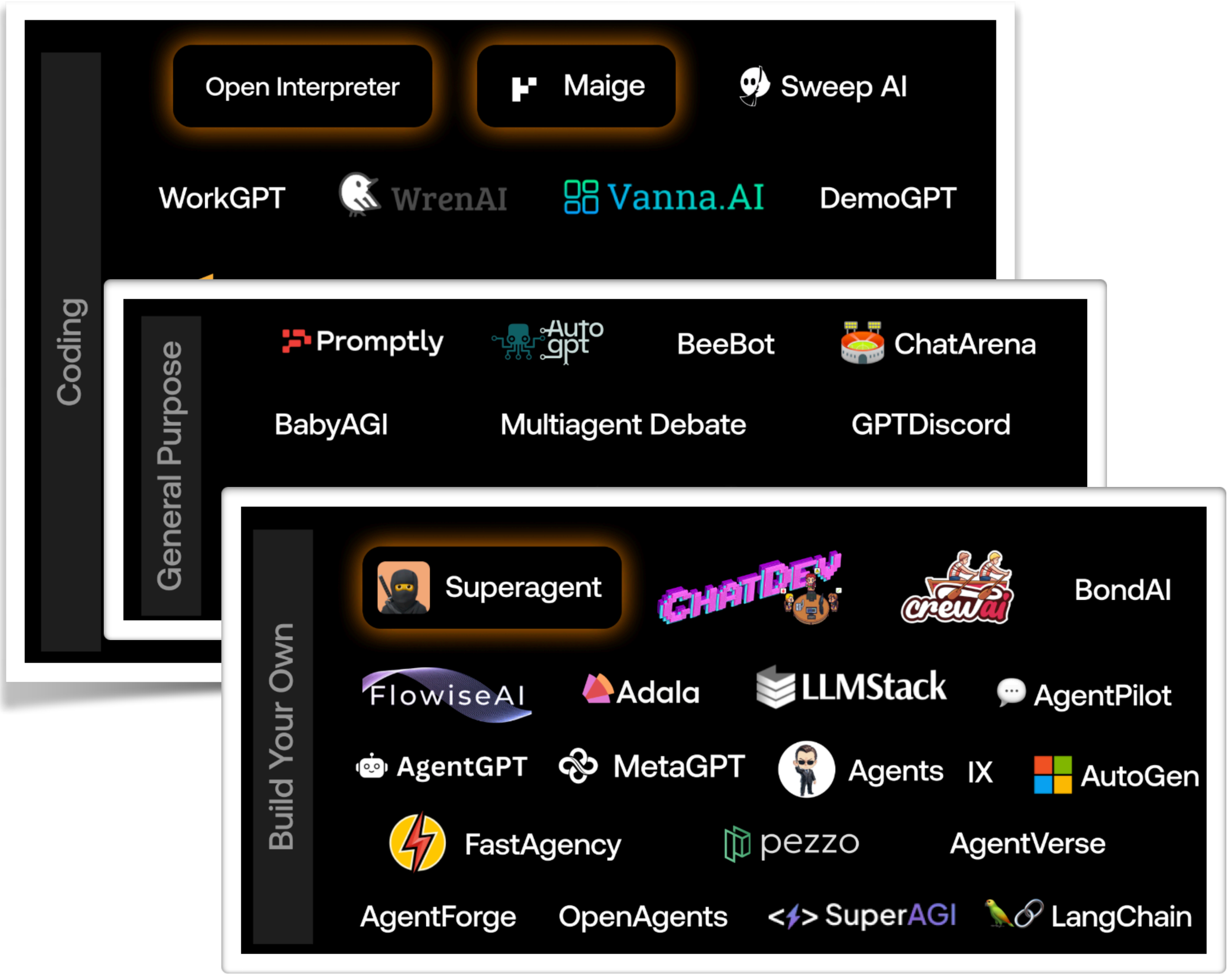


Source: https://github.com/e2b-dev/awesome-ai-agents

What are the Differences

for **humans** writing code, since the LLM era?

- A new kind of software: **agents**!
  - or: **LLM-integrated** software
- Many of them are programmed by humans
  - at least for now ...
- What about their
  - **correctness**?
  - **efficiency**?
  - **modularity**?



Source: https://github.com/e2b-dev/awesome-ai-agents

Correctness?

- Shao et al. 2025. *Are LLMs Correctly Integrated into Software Systems?* In ICSE'25.

- "Our study finds that integration defects are widespread, with **77% of these applications containing more than 3 types of defects** ... including unexpected fail-stops, incorrect software behaviors, slow execution, unfriendly UI, increased token cost, and secure vulnerabilities."

Are LLMs Correctly Integrated into Software Systems?

Yuchen Shao§, Yuheng Huang†, Jiawei Shen§, Lei Ma†‡, Ting Su§, Chengcheng Wan*§
§ East China Normal University, Shanghai, China † The University of Tokyo, Tokyo, Japan
‡ University of Alberta, Edmonton, AB, Canada
ycshao@stu.ecnu.edu.cn, yuhenghuang42@g.ecc.u-tokyo.ac.jp, javishen@stu.ecnu.edu.cn,
ma.lei@acm.org, tsu@sei.ecnu.edu.cn, ccwan@sei.ecnu.edu.cn

***Abstract*—Large language models (LLMs) provide effective solutions in various application scenarios, with the support of retrieval-augmented generation (RAG). However, developers face challenges in integrating LLM and RAG into software systems, due to lacking interface specifications, various requirements from software context, and complicated system management. In this paper, we have conducted a comprehensive study of 100 open-source applications that incorporate LLMs with RAG support, and identified 18 defect patterns. Our study reveals that 77% of these applications contain more than three types of integration defects that degrade software functionality, efficiency, and security. Guided by our study, we propose systematic guidelines for resolving these defects in software life cycle. We also construct an open-source defect library HYDRANGEA [1].**

***Index Terms*—LLM, defects, empirical software engineering**

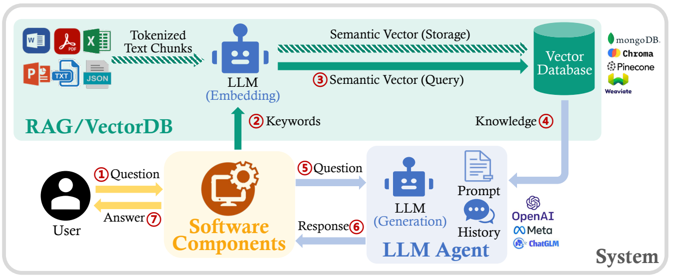


Fig. 1. Components and workflow of LLM-enabled software.

I. Introduction

A. Motivation

Large language models (LLMs) offer effective solutions for a spectrum of language-processing tasks. Retrieval-augmented generation (RAG) techniques further enhance their capabilities by providing relevant information from external data sources. Together, LLM and RAG serve as efficient and cost-effective proxies of artificial general intelligence (AGI). Consequently, an increasing number of software systems are integrating LLMs with RAG support to realize intelligence features, which this paper refers to as *LLM-enabled software*. Indeed, more than 36,000 open-source LLM-enabled software projects have been created on GitHub in the past six months, to solve a variety of real-world problems.

Various frameworks [2]–[8] offer LLM and RAG solutions as third-party APIs, significantly reducing developers' burden of incorporating them. However, challenges still remain in building correct, efficient, and reliable LLM-enabled software. In fact, developers may overlook integration failures, due to insufficient testing and the lack of LLM and RAG knowledge. Thus, understanding the defects and their root causes in LLM-enabled software has become urgent.

Challenge-1: Lacking interface specifications. Unlike AI tasks with categorical outputs, LLM performs generation tasks and typically lacks detailed specifications of their interfaces and behaviors. Given a particular input, LLMs cannot specify whether they could provide a correct answer in a certain format. Moreover, it is impractical to define the capability boundary of a certain LLM, especially when enhanced by RAG. Therefore, LLM-enabled software cannot formally describe the interface between LLM, RAG, and the remaining software components. Thus, developers have to tackle the under-specified interface and resolve potential failures.

Challenge-2: Various requirements from software context. As a generative model, an LLM enhanced by RAG could provide different responses for the same question. While these responses may all seem feasible, not all of them will match the software context and trigger the correct software behavior. For example, a user expects landscape descriptions from a travel agent and statistics from a data analyzer, with the question "how about Ottawa?". Furthermore, conventional software components typically have strict format requirements, whereas data-driven LLM supports various formats. Thus, developers have to instruct the general-purpose LLMs to perform specific tasks within the software context.

Challenge-3: Complicated system management. The LLM and RAG algorithms are resource-intensive and require system management to ensure performance. Even adopting cloud services to reduce computation costs, substantial memory is required for transferring and processing the intermediate results. Additionally, LLMs have vulnerabilities and could become security weak links after obtaining system privileges [9]–[11]. Thus, developers have to carefully manage resources and protect the security of the entire system.

Prior work studies the integration of AI components with categorical outputs [12]–[15]. Other work focuses on improving LLM and RAG algorithms [16]–[19]. However, to

* Chengcheng Wan is the corresponding author.

Efficiency?

- Mell et al. 2025. *Opportunistically Parallel Lambda Calculus*. In OOPSLA'25.

- "State-of-the-art LLMs are typically provided as remote services … their scale is so large … However, despite numerous **languages** and **frameworks** proposed to help developers write these [LLM glue] scripts, **none of them focus on automatic parallelization and streaming**."

Opportunistically Parallel Lambda Calculus

STEPHEN MELL, University of Pennsylvania, USA
KONSTANTINOS KALLAS, University of California, Los Angeles, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA
OSBERT BASTANI, University of Pennsylvania, USA

Scripting languages are widely used to compose *external calls* such as native libraries or network services. In such scripts, execution time is often dominated by waiting for these external calls, rendering traditional single-language optimization ineffective. To address this, we propose a novel *opportunistic* evaluation strategy for scripting languages based on a core lambda calculus that *automatically* dispatches independent external calls in parallel and streams their results. We prove that our approach is confluent, ensuring that it preserves the programmer's original intent, and that it eventually executes every external call. We implement this approach in a scripting language called Epic. We demonstrate the versatility and performance of Epic, focusing on programs that invoke heavy external computation through the use of large language models (LLMs) and other APIs. Across five scripts, we show that opportunistic evaluation improves total running time (up to 6.2×) and latency (up to 12.7×) compared to several state-of-the-art baselines, while performing very close (between 1.3% and 18.5% running time overhead) to hand-tuned manually optimized asynchronous Rust implementations. For Tree-of-Thoughts, a prominent LLM reasoning approach, we achieve a 6.2× performance improvement over the authors' own implementation.

arXiv:2405.11361v3 [cs.PL] 13 Jun 2025

1 INTRODUCTION

A key application of scripting languages such as Python and the shell is to "glue" together *external calls*, i.e., algorithms implemented as foreign calls to low-level languages or network calls to remote APIs, whose implementations are opaque to the interpreter [40]. Unlike other programs, performance bottlenecks in glue scripts are typically the external calls, rather than script evaluation itself. While each individual external call is usually heavily optimized by its developers, their composition is not—in glue scripts there are often opportunities to execute independent external calls in parallel and stream results between them.

Traditionally, it is up to the script developer to exploit parallelism or streaming across calls, but doing so may result in complex multi-threaded code, undermining the simplicity and usability of scripting languages. At the same time, automatically exposing parallelization and streaming opportunities across external calls is challenging because they are often interweaved with complex control flow. Traditional parallelizing compilers [6, 25] and evaluation strategies [12, 14] focus on identifying dependencies in a single language without external calls. Recent work on parallelizing shell glue scripts with external calls [28, 50] can only expose limited parallelism across control-flow, focusing instead on parallelizing contiguous pipelines.

We propose a novel, general-purpose, higher-order scripting language designed to automatically execute external calls in a parallel and streaming way, while remaining consistent with the sequential evaluation semantics programmers expect.

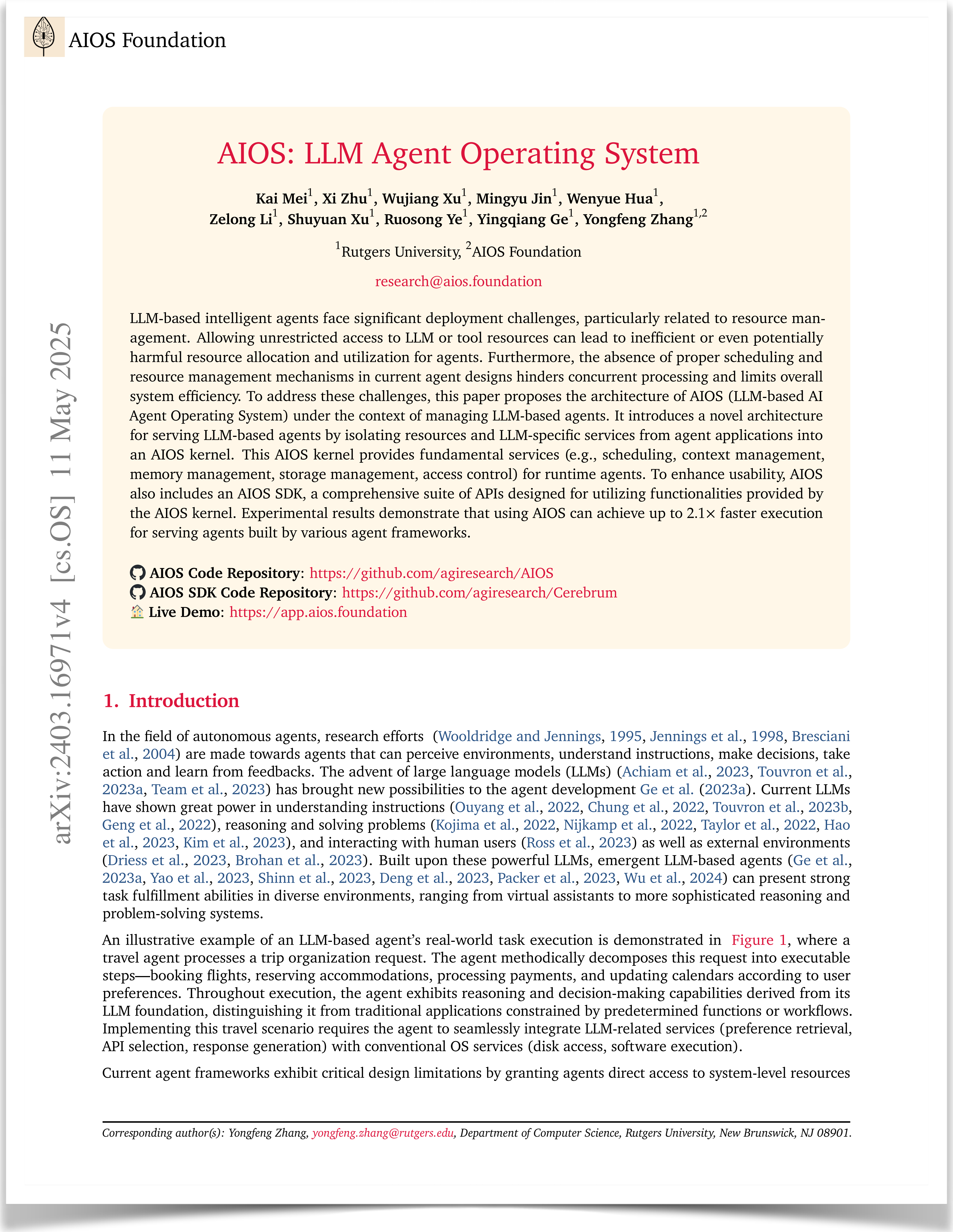
A key assumption that we make is that external call dependencies can be fully determined by the call arguments, which holds for many categories of external calls, e.g., API calls and command utilities. This assumption allows the language to evaluate two calls out of order and in parallel unless they are connected by an explicit data dependency. Our approach has three components: (1) a novel core calculus, λ^O that has external calls as first-class citizens, (2) a novel *opportunistic* evaluation strategy that parallelizes independent external calls during the execution of the program,

Authors' addresses: Stephen Mell, University of Pennsylvania, USA, sm1@cis.upenn.edu; Konstantinos Kallas, University of California, Los Angeles, USA, kkallas@ucla.edu; Steve Zdancewic, University of Pennsylvania, USA, stevez@cis.upenn.edu; Osbert Bastani, University of Pennsylvania, USA, obastani@seas.upenn.edu.

Modularity?

- Mei et al. 2025. *AIOS: LLM Agent Operating System.* In COLM'25.

- "Current agent frameworks exhibit **critical design limitations by granting agents direct access to system-level resources** … AIOS divides agent applications and resources into distinct layers … This separation facilitates systematic resource management, efficiency optimization, and safety enhancement."



AIOS Foundation

arXiv:2403.16971v4 [cs.OS] 11 May 2025

AIOS: LLM Agent Operating System

Kai Mei[1], Xi Zhu[1], Wujiang Xu[1], Mingyu Jin[1], Wenyue Hua[1], Zelong Li[1], Shuyuan Xu[1], Ruosong Ye[1], Yingqiang Ge[1], Yongfeng Zhang[1,2]

[1]Rutgers University, [2]AIOS Foundation

research@aios.foundation

LLM-based intelligent agents face significant deployment challenges, particularly related to resource management. Allowing unrestricted access to LLM or tool resources can lead to inefficient or even potentially harmful resource allocation and utilization for agents. Furthermore, the absence of proper scheduling and resource management mechanisms in current agent designs hinders concurrent processing and limits overall system efficiency. To address these challenges, this paper proposes the architecture of AIOS (LLM-based AI Agent Operating System) under the context of managing LLM-based agents. It introduces a novel architecture for serving LLM-based agents by isolating resources and LLM-specific services from agent applications into an AIOS kernel. This AIOS kernel provides fundamental services (e.g., scheduling, context management, memory management, storage management, access control) for runtime agents. To enhance usability, AIOS also includes an AIOS SDK, a comprehensive suite of APIs designed for utilizing functionalities provided by the AIOS kernel. Experimental results demonstrate that using AIOS can achieve up to 2.1× faster execution for serving agents built by various agent frameworks.

AIOS Code Repository: https://github.com/agiresearch/AIOS
AIOS SDK Code Repository: https://github.com/agiresearch/Cerebrum
Live Demo: https://app.aios.foundation

1. Introduction

In the field of autonomous agents, research efforts (Wooldridge and Jennings, 1995, Jennings et al., 1998, Bresciani et al., 2004) are made towards agents that can perceive environments, understand instructions, make decisions, take action and learn from feedbacks. The advent of large language models (LLMs) (Achiam et al., 2023, Touvron et al., 2023a, Team et al., 2023) has brought new possibilities to the agent development Ge et al. (2023a). Current LLMs have shown great power in understanding instructions (Ouyang et al., 2022, Chung et al., 2022, Touvron et al., 2023b, Geng et al., 2022), reasoning and solving problems (Kojima et al., 2022, Nijkamp et al., 2022, Taylor et al., 2022, Hao et al., 2023, Kim et al., 2023), and interacting with human users (Ross et al., 2023) as well as external environments (Driess et al., 2023, Brohan et al., 2023). Built upon these powerful LLMs, emergent LLM-based agents (Ge et al., 2023a, Yao et al., 2023, Shinn et al., 2023, Deng et al., 2023, Packer et al., 2023, Wu et al., 2024) can present strong task fulfillment abilities in diverse environments, ranging from virtual assistants to more sophisticated reasoning and problem-solving systems.

An illustrative example of an LLM-based agent's real-world task execution is demonstrated in Figure 1, where a travel agent processes a trip organization request. The agent methodically decomposes this request into executable steps—booking flights, reserving accommodations, processing payments, and updating calendars according to user preferences. Throughout execution, the agent exhibits reasoning and decision-making capabilities derived from its LLM foundation, distinguishing it from traditional applications constrained by predetermined functions or workflows. Implementing this travel scenario requires the agent to seamlessly integrate LLM-related services (preference retrieval, API selection, response generation) with conventional OS services (disk access, software execution).

Current agent frameworks exhibit critical design limitations by granting agents direct access to system-level resources

Corresponding author(s): Yongfeng Zhang, yongfeng.zhang@rutgers.edu, Department of Computer Science, Rutgers University, New Brunswick, NJ 08901.

What are the Features

for LLM-integrated software?

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**

- Correctness-related: **Uncertain behavior**

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**

May harm robustness; also affect reproducibility

- Correctness-related: **Uncertain behavior**

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**

May harm robustness; also affect reproducibility

- Correctness-related: **Uncertain behavior**
- Efficiency-related: **Long execution times**

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**

May harm robustness; also affect reproducibility

- Correctness-related: **Uncertain behavior**

Require the software to schedule LLM calls wisely

- Efficiency-related: **Long execution times**

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**
- Correctness-related: **Uncertain behavior**
- Efficiency-related: **Long execution times**
- Modularity-related: **Distributed execution**

May harm robustness; also affect reproducibility

Require the software to schedule LLM calls wisely

What are the Features

for LLM-integrated software?

- TL;DR: They interact with LLMs, **heavily**
- Correctness-related: **Uncertain behavior**
- Efficiency-related: **Long execution times**
- Modularity-related: **Distributed execution**

May harm robustness; also affect reproducibility

Require the software to schedule LLM calls wisely

Make it hard to perform testing and profiling

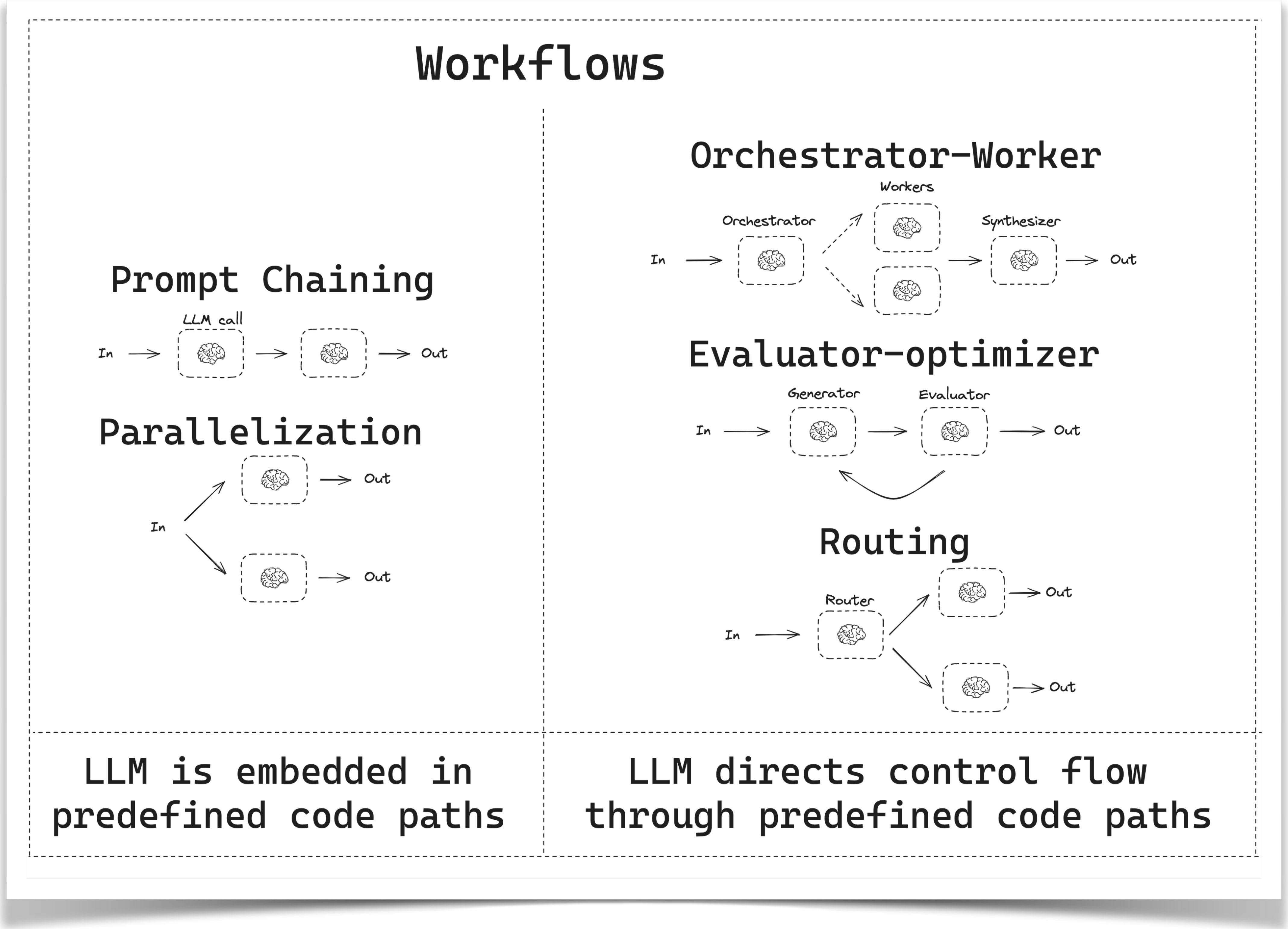
One Preliminary Thought

One Preliminary Thought

- The **logic** and **workflow** of agents are usually not very complex

One Preliminary Thought

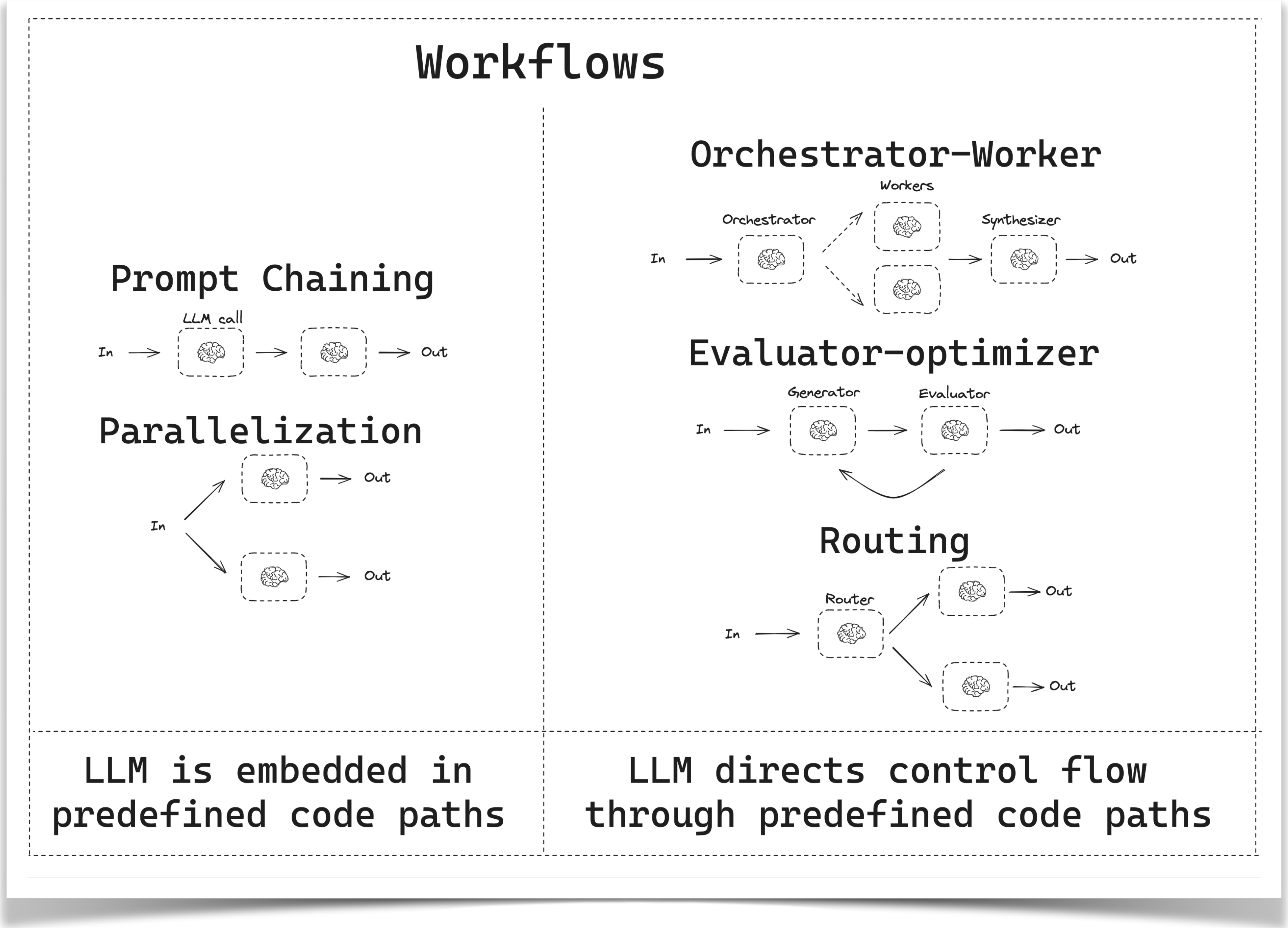
- The **logic** and **workflow** of agents are usually not very complex



Source: https://langchain-ai.github.io/langgraph/tutorials/workflows/

One Preliminary Thought

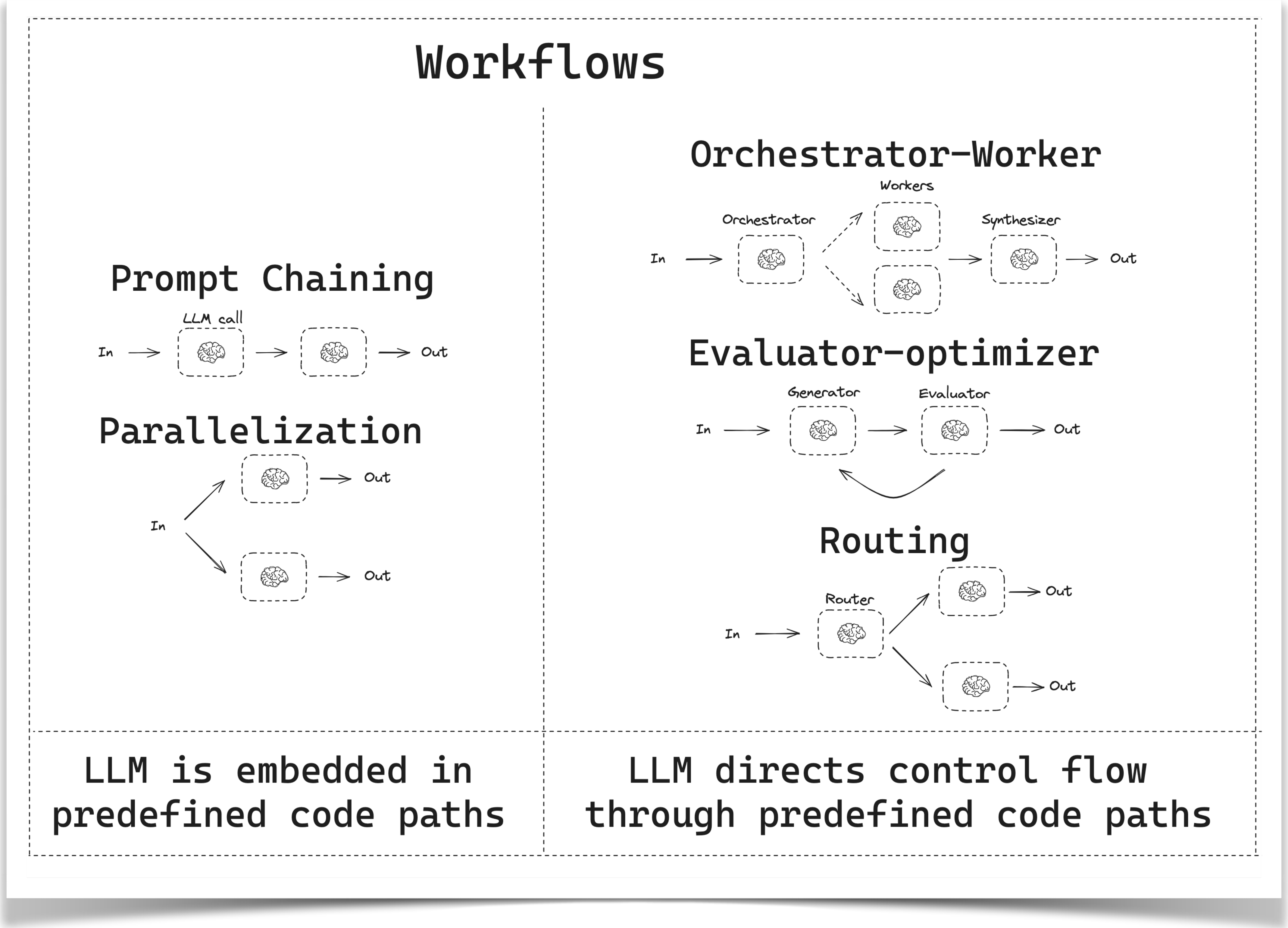
- The **logic** and **workflow** of agents are usually not very complex
- The **effects** make the engineering of agents complex
  - Uncertain behavior
  - Long execution times
  - Distributed execution



Source: https://langchain-ai.github.io/langgraph/tutorials/workflows/

One Preliminary Thought

- The **logic** and **workflow** of agents are usually not very complex
- The **effects** make the engineering of agents complex
  - Uncertain behavior
  - Long execution times
  - Distributed execution
- **Effect handler** oriented programming?



Source: https://langchain-ai.github.io/langgraph/tutorials/workflows/

An Example Workflow

orchestrator-worker

An Example Workflow

orchestrator-worker

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

An Example Workflow

orchestrator-worker

`get_topics` is a **distributed** call; it is **uncertain** as it may not return well-structured results

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

An Example Workflow

orchestrator-worker

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

`get_topics` is a **distributed** call; it is **uncertain** as it may not return well-structured results

`get_description` is a **distributed** call; it may take **long execution time**; sequential execution is inefficient

An Example Workflow

orchestrator-worker

`get_topics` is a **distributed** call; it is **uncertain** as it may not return well-structured results

`log` is a local **effectful** operation (I/O)

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

`get_description` is a **distributed** call; it may take **long execution time**; sequential execution is inefficient

An Example Workflow

orchestrator-worker

`get_topics` is a **distributed** call; it is **uncertain** as it may not return well-structured results

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

`log` is a local **effectful** operation (I/O)

`get_description` is a **distributed** call; it may take **long execution time**; sequential execution is inefficient

- The logic and workflow of agents are usually not very complex

An Example Workflow

orchestrator-worker

`get_topics` is a **distributed** call; it is **uncertain** as it may not return well-structured results

`log` is a local **effectful** operation (I/O)

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

`get_description` is a **distributed** call; it may take **long execution time**; sequential execution is inefficient

- The logic and workflow of agents are usually not very complex
- Idea: Separate **the logic and workflow** from **the implementation of these effects**

An Example Workflow

orchestrator-worker

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

An Example Workflow

orchestrator-worker

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

Synchronous version:
suitable for development

An Example Workflow

orchestrator-worker

Synchronous version:
suitable for development

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

```
with (
  # handler for async operations
  AsyncHandler(),
  # handler for LLM-call operations
  AsyncLLMHandler(**llm_kwargs),
  # handler for rendering async side effects sequentially
  AsyncSeqHandler(),
  # handler for application-specific operations
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

An Example Workflow

orchestrator-worker

Synchronous version:
suitable for development

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

Asynchronous version:
suitable for deployment

```
with (
  # handler for async operations
  AsyncHandler(),
  # handler for LLM-call operations
  AsyncLLMHandler(**llm_kwargs),
  # handler for rendering async side effects sequentially
  AsyncSeqHandler(),
  # handler for application-specific operations
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

An Example Workflow

orchestrator-worker

Synchronous version:
suitable for development

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

Asynchronous version:
suitable for deployment

```
with (
  # handler for async operations
  AsyncHandler(),
  # handler for LLM-call operations
  AsyncLLMHandler(**llm_kwargs),
  # handler for rendering async side effects sequentially
  AsyncSeqHandler(),
  # handler for application-specific operations
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

```
with (
  AsyncHandler(),
  # handler for mocking LLM-call operations
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

An Example Workflow

orchestrator-worker

Synchronous version:
suitable for development

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

Asynchronous version:
suitable for deployment

```
with (
  # handler for async operations
  AsyncHandler(),
  # handler for LLM-call operations
  AsyncLLMHandler(**llm_kwargs),
  # handler for rendering async side effects sequentially
  AsyncSeqHandler(),
  # handler for application-specific operations
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Mocking version:
suitable for testing & profiling

```
with (
  AsyncHandler(),
  # handler for mocking LLM-call operations
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

An Example Workflow

orchestrator-worker

Synchronous version:
suitable for development

```
with (
  # handler for LLM-call operations
  LLMHandler(**llm_kwargs),
  # handler for application-specific operations
  ResearchTopicsHandler(),
):
  research_topics()
```

Asynchronous version:
suitable for deployment

```
with (
  # handler for async operations
  AsyncHandler(),
  # handler for LLM-call operations
  AsyncLLMHandler(**llm_kwargs),
  # handler for rendering async side effects sequentially
  AsyncSeqHandler(),
  # handler for application-specific operations
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Mocking version:
suitable for testing & profiling

```
with (
  AsyncHandler(),
  # handler for mocking LLM-call operations
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

The workflow logic itself keeps the same!

Operations & Handlers

Operations & Handlers

```
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Operations & Handlers

Operations provides only **syntax**

```
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Operations & Handlers

Operations provides only **syntax**

```
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

```
# a handler is a manager for a set of operations
class LogHandler(Handler):
  def __init__(self):
    super().__init__()
    # this handler discharges the `log` operation
    self.register(log, self.log)

  def log(self, msg):
    print(f"[INFO] {msg}")
```

Operations & Handlers

Operations provides only **syntax**

```python
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Handlers determine the **semantics**

```python
# a handler is a manager for a set of operations
class LogHandler(Handler):
  def __init__(self):
    super().__init__()
    # this handler discharges the `log` operation
    self.register(log, self.log)

  def log(self, msg):
    print(f"[INFO] {msg}")
```

Operations & Handlers

Operations provides only **syntax**

```python
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Handlers determine the **semantics**

```python
# a handler is a manager for a set of operations
class LogHandler(Handler):
  def __init__(self):
    super().__init__()
    # this handler discharges the `log` operation
    self.register(log, self.log)

  def log(self, msg):
    print(f"[INFO] {msg}")
```

```python
class LogDateHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(log, self.log)

  def log(self, msg):
    print(f"[DATE] {datetime.now()}")
    # this handler invokes the `log` operation
    log(msg)
```

Operations & Handlers

Operations provides only **syntax**

```python
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Handlers determine the **semantics**

```python
# a handler is a manager for a set of operations
class LogHandler(Handler):
  def __init__(self):
    super().__init__()
    # this handler discharges the `log` operation
    self.register(log, self.log)

  def log(self, msg):
    print(f"[INFO] {msg}")
```

```python
class LogDateHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(log, self.log)

  def log(self, msg):
    print(f"[DATE] {datetime.now()}")
    # this handler invokes the `log` operation
    log(msg)
```

"Forward" the `log` operation to other handlers

Operations & Handlers

Operations provides only **syntax**

```
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Handlers determine the **semantics**

```
# a handler is a manager for a set of operations
class LogHandler(Handler):
  def __init__(self):
    super().__init__()
    # this handler discharges the `log` operation
    self.register(log, self.log)

  def log(self, msg):
    print(f"[INFO] {msg}")
```

```
class LogDateHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(log, self.log)

  def log(self, msg):
    print(f"[DATE] {datetime.now()}")
    # this handler invokes the `log` operation
    log(msg)
```

"Forward" the `log` operation to other handlers

```
with LogHandler(), LogDateHandler():
  log("Hello World!")
# [DATE] 2025-06-29 20:25:17.102486
# [INFO] Hello World!
```

Operations & Handlers

Operations provides only **syntax**

```python
# an operation is a callable object
get_topics = Operation()
get_description = Operation()
log = Operation()
```

Handlers determine the **semantics**

```python
# a handler is a manager for a set of operations
class LogHandler(Handler):
  def __init__(self):
    super().__init__()
    # this handler discharges the `log` operation
    self.register(log, self.log)

  def log(self, msg):
    print(f"[INFO] {msg}")
```

```python
class LogDateHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(log, self.log)

  def log(self, msg):
    print(f"[DATE] {datetime.now()}")
    # this handler invokes the `log` operation
    log(msg)
```

"Forward" the `log` operation to other handlers

Compose handlers seamlessly

```python
with LogHandler(), LogDateHandler():
  log("Hello World!")
# [DATE] 2025-06-29 20:25:17.102486
# [INFO] Hello World!
```

Composing Operations

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result
  - `await_(fut)`: wait for the completion of `fut`, created by an `async_` call

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result
  - `await_(fut)`: wait for the completion of `fut`, created by an `async_` call
  - `complete(prompt)`: request an LLM to generate text from `prompt`

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result
  - `await_(fut)`: wait for the completion of `fut`, created by an `async_` call
  - `complete(prompt)`: request an LLM to generate text from `prompt`
- to implement:

```
class AsyncResearchTopicsHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(get_topics, self.get_topics)
    self.register(get_description, self.get_description)
    self.register(log, self.log)
```

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result
  - `await_(fut)`: wait for the completion of `fut`, created by an `async_` call
  - `complete(prompt)`: request an LLM to generate text from `prompt`
- to implement:

```
class AsyncResearchTopicsHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(get_topics, self.get_topics)
    self.register(get_description, self.get_description)
    self.register(log, self.log)
```

```
def get_topics(area):
  content = await_(
    complete(f"Give a JSON list of topics in the research area {area}.")
  )
  ...  # try to parse `content` with error handling
```

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result
  - `await_(fut)`: wait for the completion of `fut`, created by an `async_` call
  - `complete(prompt)`: request an LLM to generate text from `prompt`
- to implement:

```
class AsyncResearchTopicsHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(get_topics, self.get_topics)
    self.register(get_description, self.get_description)
    self.register(log, self.log)
```

```
def get_topics(area):
  content = await_(
    complete(f"Give a JSON list of topics in the
research area {area}.")
  )
  ...  # try to parse `content` with error handling
```

```
def get_description(topic):
  return complete(f"Give a short description about
the topic {topic}.")
```

Composing Operations

- Consider using 3 operations:
  - `async_(coro, post_fn)`: schedule a coroutine `coro` with `post_fn` as a callback on the result
  - `await_(fut)`: wait for the completion of `fut`, created by an `async_` call
  - `complete(prompt)`: request an LLM to generate text from `prompt`
- to implement:

```
class AsyncResearchTopicsHandler(Handler):
  def __init__(self):
    super().__init__()
    self.register(get_topics, self.get_topics)
    self.register(get_description, self.get_description)
    self.register(log, self.log)
```

```
def get_topics(area):
  content = await_(
    complete(f"Give a JSON list of topics in the research area {area}.")
  )
  ...  # try to parse `content` with error handling
```

```
def get_description(topic):
  return complete(f"Give a short description about the topic {topic}.")
```

```
def log(msg):
  async def aux():
    return await msg if isinstance(msg, Awaitable) else msg

  # use `print` as a callback
  return async_(aux(), print)
```

Reusing Effect Handlers

to achieve modularity

Reusing Effect Handlers

to achieve modularity

Asynchronous version: suitable for deployment

```
with (
  AsyncHandler(),
  AsyncLLMHandler(**llm_kwargs),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Reusing Effect Handlers

to achieve modularity

Asynchronous version: suitable for deployment

```
with (
  AsyncHandler(),
  AsyncLLMHandler(**llm_kwargs),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

```
with (
  AsyncHandler(),
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Mocking version: suitable for testing & profiling

Reusing Effect Handlers

to achieve modularity

Asynchronous version: suitable for deployment

```
with (
  AsyncHandler(),
  AsyncLLMHandler(**llm_kwargs),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

```
with (
  AsyncHandler(),
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Mocking version: suitable for testing & profiling

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

Reusing Effect Handlers

to achieve modularity

Asynchronous version: suitable for deployment

```
with (
  AsyncHandler(),
  AsyncLLMHandler(**llm_kwargs),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

```
with (
  AsyncHandler(),
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Mocking version: suitable for testing & profiling

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

- The workflow looks sequential, but the handlers exploit LLM service's parallelization

Reusing Effect Handlers

to achieve modularity

Asynchronous version: suitable for deployment

```
with (
  AsyncHandler(),
  AsyncLLMHandler(**llm_kwargs),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

```
with (
  AsyncHandler(),
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Mocking version: suitable for testing & profiling

```
def research_topics():
  topics = get_topics("PL techniques for LLM applications")
  for topic in topics:
    log(topic)
    description = get_description(topic)
    log(description)
```

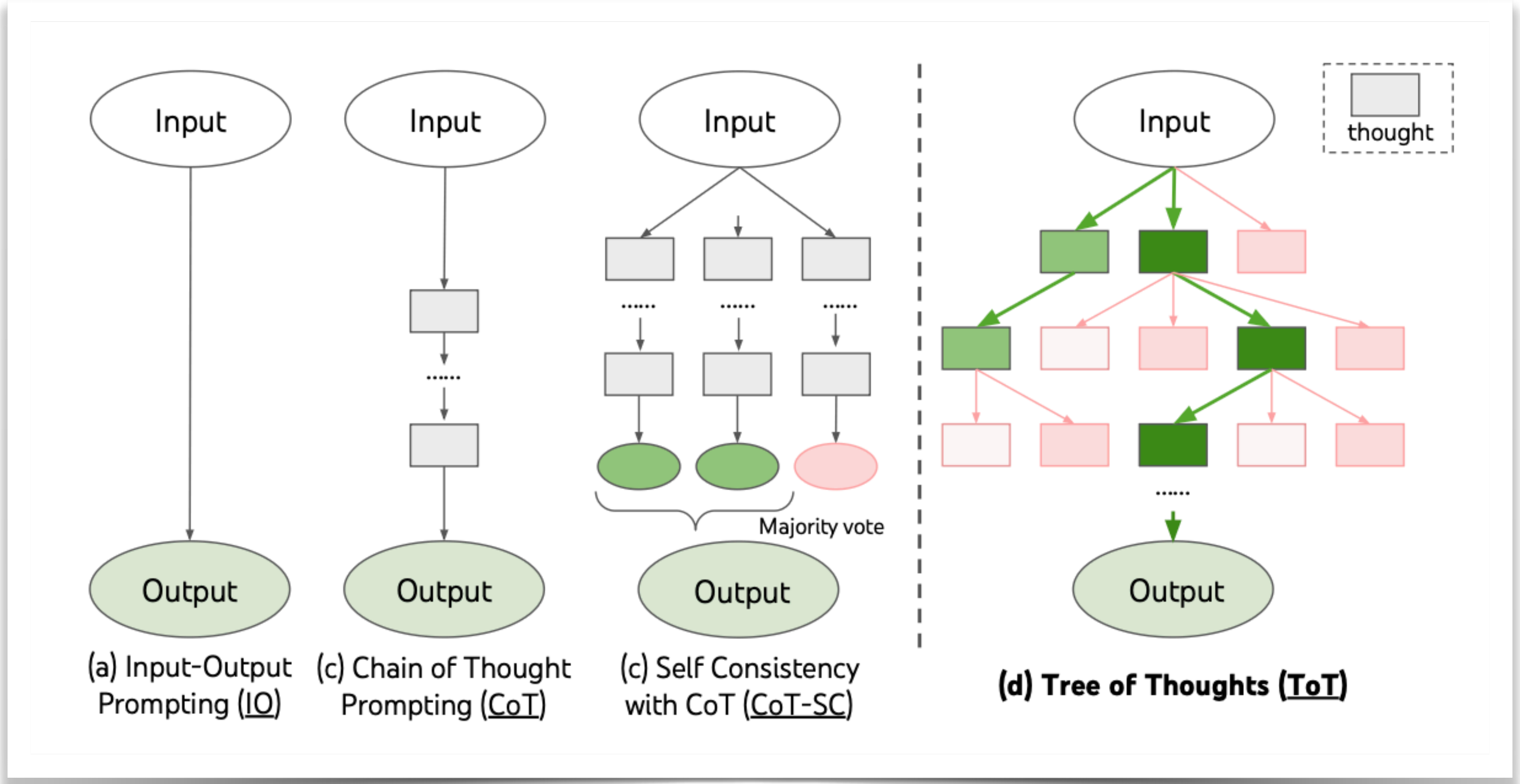
- The workflow looks sequential, but the handlers exploit LLM service's parallelization
- **Separating effects from workflow allows "modular" thinking of correctness & efficiency**

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

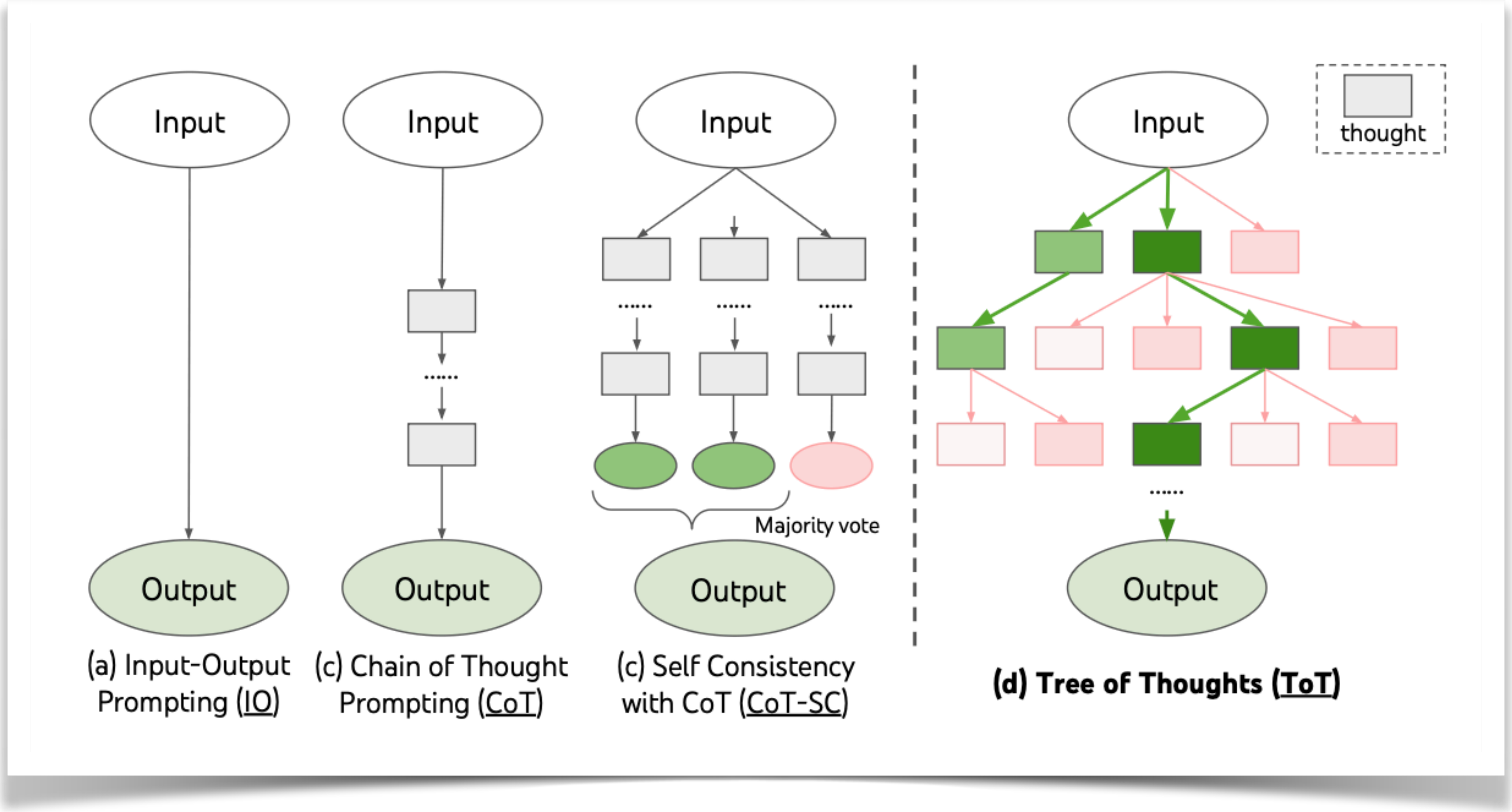
Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.



Case Study: Tree-of-Thoughts (ToT)

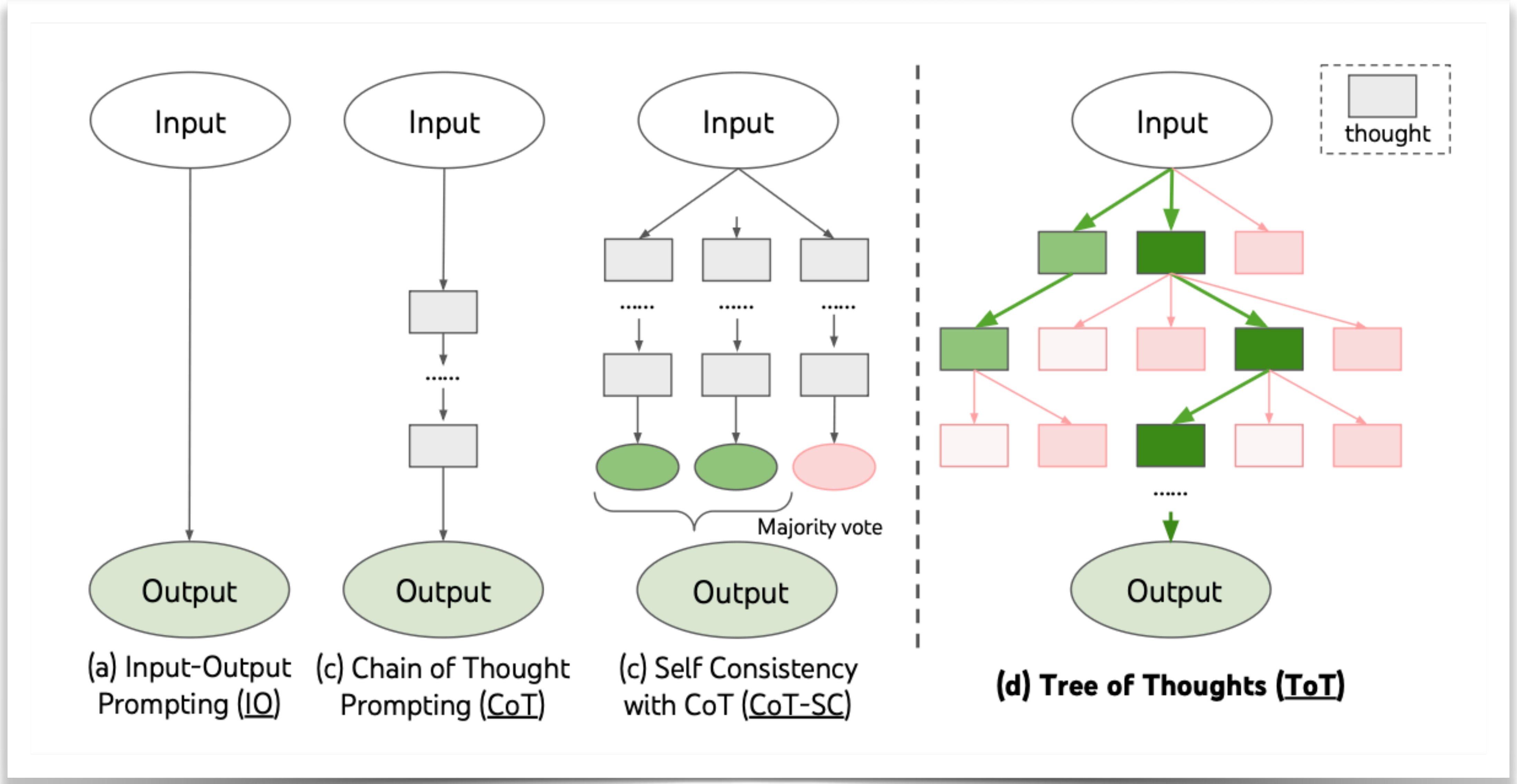
Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.



Treat reasoning as graph searching:
Use an LLM to generate thoughts (nodes)
and evaluate their fitness (weights)

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.



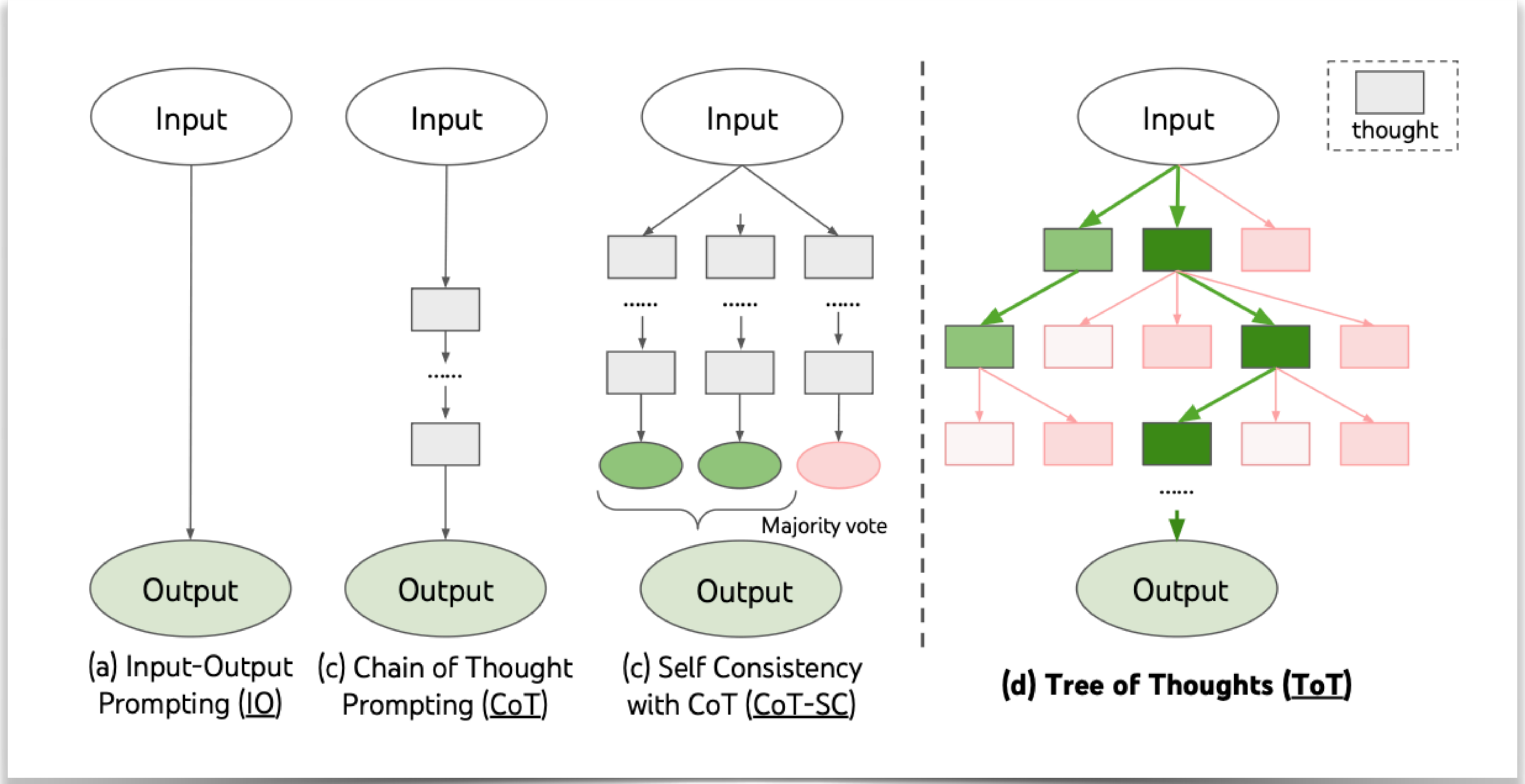

```
init, expand, score = Operation(), Operation(), Operation()


def tree_of_thoughts(n_steps, n_select, n_eval):
  frontier = [init()]
  for _ in range(n_steps):
    expanded = [expand(state) for state in frontier]
    candidates = chain(*expanded)  # flatten the list
    scored = [score(cand, n_eval) for cand in candidates]
    frontier = top_k(scored, n_select)  # select greedily
  print(frontier)
```

Treat reasoning as graph searching:
Use an LLM to generate thoughts (nodes)
and evaluate their fitness (weights)

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.




```
init, expand, score = Operation(), Operation(), Operation()


def tree_of_thoughts(n_steps, n_select, n_eval):
  frontier = [init()]
  for _ in range(n_steps):
    expanded = [expand(state) for state in frontier]
    candidates = chain(*expanded)  # flatten the list
    scored = [score(cand, n_eval) for cand in candidates]
    frontier = top_k(scored, n_select)  # select greedily
  print(frontier)
```

Treat reasoning as graph searching:
Use an LLM to generate thoughts (nodes)
and evaluate their fitness (weights)

Again: The logic and workflow of agents are usually not very complex

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

- Create effect handlers for the ToT operations to solve the Game of 24

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

- Create effect handlers for the ToT operations to solve the Game of 24

```
with (
  LLMHandler(**llm_kwargs),
  Game24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

- Create effect handlers for the ToT operations to solve the Game of 24

```
with (
  LLMHandler(**llm_kwargs),
  Game24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

→

```
with (
  AsyncHandler(), AsyncLLMHandler(**llm_kwargs),
  AsyncGame24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

- Create effect handlers for the ToT operations to solve the Game of 24

```
with (
  LLMHandler(**llm_kwargs),
  Game24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

```
with (
  AsyncHandler(), AsyncLLMHandler(**llm_kwargs),
  AsyncGame24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

Achieve ~10x speedup without changing the workflow logic

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

- Create effect handlers for the ToT operations to solve the Game of 24

```
with (
  LLMHandler(**llm_kwargs),
  Game24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

```
with (
  AsyncHandler(), AsyncLLMHandler(**llm_kwargs),
  AsyncGame24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

Achieve ~10x speedup without changing the workflow logic

- The official implementation did not exploit parallelization

Case Study: Tree-of-Thoughts (ToT)

Shunyu et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In NeurIPS'23.

- Create effect handlers for the ToT operations to solve the Game of 24

```
with (
  LLMHandler(**llm_kwargs),
  Game24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

```
with (
  AsyncHandler(), AsyncLLMHandler(**llm_kwargs),
  AsyncGame24Handler(),
):
  tree_of_thoughts(n_steps=4, n_select=5, n_eval=3)
```

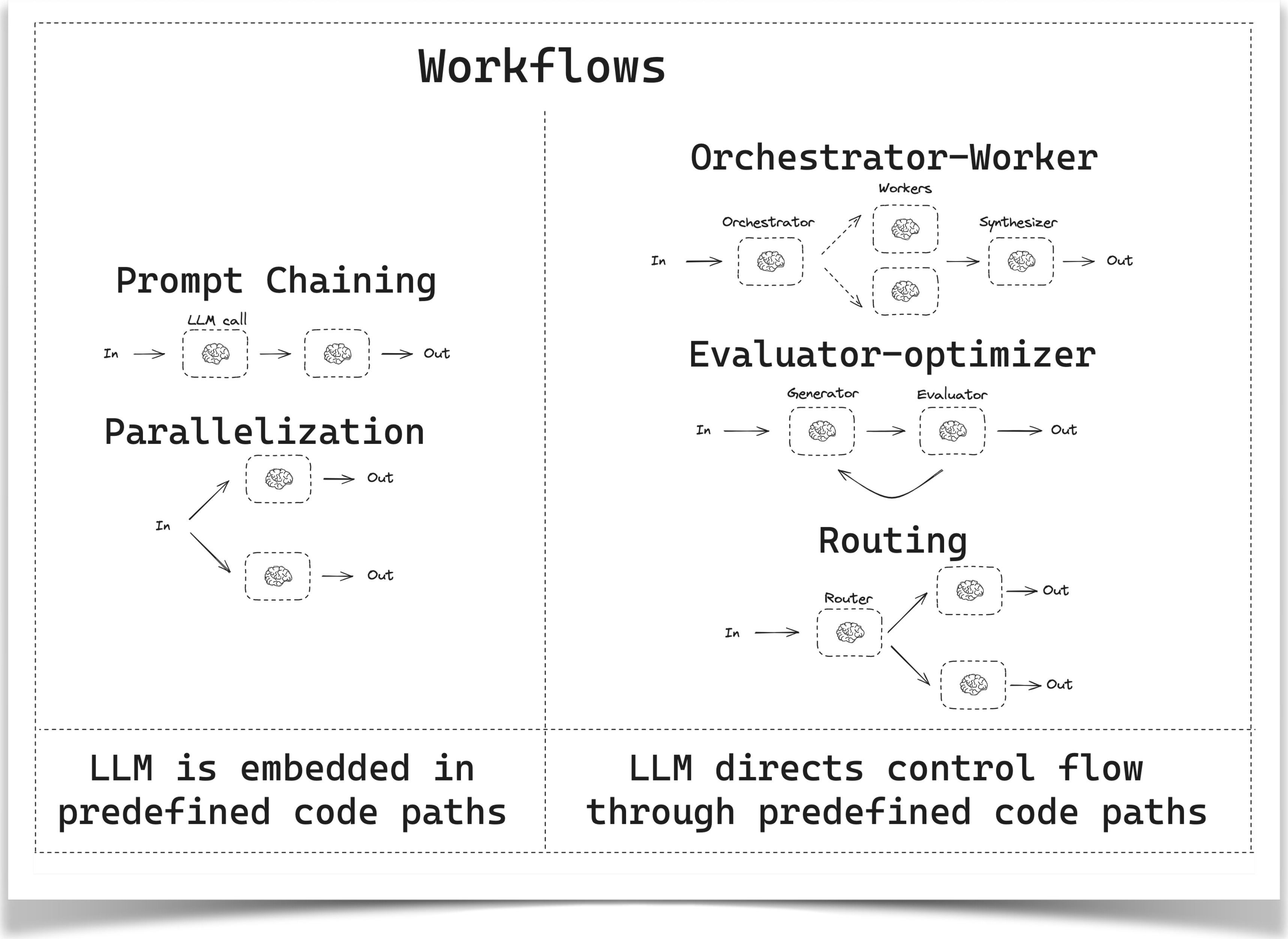
Achieve ~10x speedup without changing the workflow logic

- The official implementation did not exploit parallelization
- Manual optimization would break the code structure of the workflow

Takeaway: One Preliminary Thought

about PL for **humans** writing code

- The **logic** and **workflow** of agents are usually not very complex
- The **effects** make the engineering of agents complex
  - Uncertain behavior
  - Long execution times
  - Distributed execution
- **Effect handler** oriented programming might be suitable for implementing agents




```
with (
  AsyncHandler(),
  AsyncReplayLLMHandler(trace),
  AsyncSeqHandler(),
  AsyncResearchTopicsHandler(),
):
  research_topics()
```

Source: https://langchain-ai.github.io/langgraph/tutorials/workflows/