Monadic Context Engineering

Yifan Zhang[*]   **Yang Yuan**[*]   **Mengdi Wang**   **Andrew Chi-Chih Yao**[†]
[†]IIIS, Tsinghua University

Abstract

The proliferation of Large Language Models (LLMs) has catalyzed a shift towards autonomous agents capable of complex reasoning and tool use. However, current agent architectures are frequently constructed using imperative, ad hoc patterns. This results in brittle systems plagued by difficulties in state management, error handling, and concurrency. This paper introduces **Monadic Context Engineering (MCE)**, a novel architectural paradigm leveraging the algebraic structures of Functors, Applicative Functors, and Monads to provide a formal foundation for agent design. MCE treats agent workflows as computational contexts where cross-cutting concerns, such as state propagation, short-circuiting error handling, and asynchronous execution, are managed intrinsically by the algebraic properties of the abstraction. We demonstrate how Monads enable robust sequential composition, how Applicatives provide a principled structure for parallel execution, and crucially, how **Monad Transformers** allow for the systematic composition of these capabilities. This layered approach enables developers to construct complex, resilient, and efficient AI agents from simple, independently verifiable components. We further extend this framework to describe Meta-Agents, which leverage MCE for generative orchestration, dynamically creating and managing sub-agent workflows through metaprogramming.

1 Introduction

The vanguard of artificial intelligence research increasingly focuses on building autonomous agents: systems that reason, plan, and act to accomplish goals by interacting with environments (Yao et al., 2022; Shinn et al., 2023). While the cognitive capabilities of underlying LLMs are critical, the architectural challenge of orchestrating the operational loop, typically a cycle of *Thought*, *Action*, and *Observation*, presents a formidable barrier to creating robust and scalable systems.

Engineers building these agents confront a recurring set of fundamental challenges. Paramount among these is the maintenance of state integrity, requiring the reliable propagation of memory, beliefs, and history across a sequence of potentially fallible operations. Simultaneously, agents require error resilience to gracefully handle real-world failures, such as API timeouts or malformed model outputs, without obfuscating core logic with defensive boilerplate. Furthermore, developers need logical composability to construct complex behaviors from independent units of logic, facilitating the seamless assembly, reordering, and substitution of steps.

Beyond sequential logic, modern agents demand robust concurrency to orchestrate multiple simultaneous actions without descending into the complexities of manual thread management. Ideally, the architecture should also strictly manage computational effects, separating deterministic logic from non-deterministic interactions with the external world. Finally, as systems scale, we must

arXiv:2512.22431v5 [cs.AI] 22 Jan 2026

address agent orchestration, managing specialized teams of agents that can be formed dynamically to address novel problems without introducing chaotic interactions.

Current mainstream approaches are typically imperative, addressing these issues with improvisational solutions. Consequently, developers face highly coupled codebases with convoluted control flow, rendering the resulting agents difficult to test, debug, and evolve. The lack of a principled structure for managing state, failure, and concurrency leads to inherently brittle systems.

This architectural deficit is becoming more apparent as the community moves towards standardized interaction patterns, such as the Model Context Protocol (MCP) (Anthropic, 2024; Model Context Protocol, 2025), which demand robust and predictable orchestration logic.

We propose that a solution resides in a powerful hierarchy of abstractions from functional programming and category theory: the Functor, the Applicative Functor, and the Monad (Moggi, 1991; Wadler, 1992). These are not merely design patterns but formal algebraic structures that provide a standardized method to compose computations within a *context*. A **Functor** allows one to apply a pure function to a value inside a context (mapping). An **Applicative** extends this by enabling the application of a wrapped function to a wrapped value, a structure essential for executing independent computations concurrently. Finally, a **Monad** allows for the sequencing of dependent operations where the subsequent computation is determined by the result of the previous one (binding).

This progression, culminating in the Monad's `bind` operation, allows us to construct a railway for computation. Each logical step acts as a station, while `bind` lays the track, ensuring the computational process proceeds smoothly on the *success track*. If any step fails, `bind` automatically shunts the entire computation to a *failure track*, bypassing subsequent stations to proceed directly to the destination. This paper details the design, implementation, and application of `AgentMonad`, a structure embodying these principles to bring profound benefits to AI agent engineering.

These abstractions are already practical at scale in adjacent software-engineering settings. For example, the Flag Boot microservice framework builds on Scala functional effect stacks (e.g., Cats/Cats Effect) and monadic effect types to make concurrency, scheduling, and failure propagation explicit in large service systems (Yang et al., 2022; Liu, 2023; InfoQ, 2023). Public community notes documenting these engineering practices also connect monadic effect management (including Monad and IO Monad) to AI-assisted programming and type-safe software construction, and explicitly discuss Flag Boot alongside Cats type classes (Scala Meetup, 2023). In parallel, recent LLM-assisted software engineering pipelines use typed functional translations to automate parts of backend-system verification, where compositional effect management is central to scalability (Xu et al., 2025). MCE focuses these ingredients on the control-flow requirements of interactive, tool-using agents.

2   From Functors to Monads: The AgentMonad Design

To apply Monadic Context Engineering to AI agents, we introduce a specialized structure: the `AgentMonad`. The *context* it manages is a composite structure encapsulating all critical aspects of an agent's execution. We build our understanding by following the classic Functor-Applicative-Monad progression.

2.1 The Anatomy of the `AgentMonad`: A Monad Transformer Stack

The core architectural challenge in agent design is managing multiple, overlapping concerns simultaneously. A single-agent operation might need to interact with an external API, handle possible failures, and update the internal memory or world model of the agent. Attempting to manage these concerns with naive nesting, for example, a type like `Task<Either<State<...»>`, is unworkable. It forces developers to manually unwrap each layer of the context, reintroducing the deeply nested, callback-style code that monads are intended to eliminate.

The principled solution is the Monad Transformer, a concept from functional programming that allows for the systematic composition of monadic capabilities Liang et al. (1995). A monad transformer, `T`, is a type constructor that takes an existing monad `M` and produces a new, more powerful monad, `T(M)`, that combines the behaviors of both. Crucially, transformers provide a **lift** operation (`lift : M A → T M A`) that allows any computation in an inner monad to be seamlessly used within the context of the combined outer monad. This enables the creation of a layered stack of capabilities that share a single, unified interface.

The `AgentMonad` utilizes this technique to create a stack designed specifically for agentic workflows (Figure 1). At the base lies the `IO` or `Task` Monad, which manages interactions with the external world. This separates the description of an action from its execution, making behavior observable. We then apply the `EitherT` Transformer, which introduces short-circuiting error handling. This directly models the requirements of specifications like the Model Context Protocol (MCP) Model Context Protocol (2025), where tool results must explicitly indicate success or failure. Finally, we wrap the stack in the `StateT` Transformer.

The resulting type, `StateT S (EitherT E IO)`, represents a computation that is simultaneously stateful, fallible, and capable of side effects. A single `bind` operation on this composite structure correctly threads the state, checks for errors, and sequences external actions. Mathematically, this implies the shape $S \rightarrow IO(Either(E, (A, S)))$, unifying all contexts into a single return type.

This layered construction provides a robust and formal foundation for agent architecture. The resulting `AgentMonad`, with its type signature `StateT S (EitherT E IO) A`, directly maps its algebraic structure to the primary challenges of agent engineering. It ensures interactions are observable, error handling is robust, state management is functional, and workflows are composable.

2.2 Level 1: AgentMonad as a Functor

The most fundamental operation involves applying a pure function to the value inside our context without altering the context itself. This is the role of the Functor and its `map` operation.

The **map** function (or `fmap`) accepts a function $f : A \rightarrow B$ and an `AgentMonad[S, A]`, returning an `AgentMonad[S, B]`. It applies f to the wrapped value while preserving the state and success status. Crucially, if the flow has already failed, `map` performs no operation.

2.3 Level 2: AgentMonad as an Applicative Functor

Applicatives extend Functors to handle a more complex scenario: what if the function we want to apply is *also* wrapped in our context? This is particularly useful for combining the results of independent computations.

The **apply** operation (or `<*>`) takes an `AgentMonad` containing a function $(A \rightarrow B)$ and an `AgentMonad` containing a value (A), returning a new context containing the result (B). This

The `AgentMonad` is a concrete implementation of this final, layered monad, combining all capabilities.
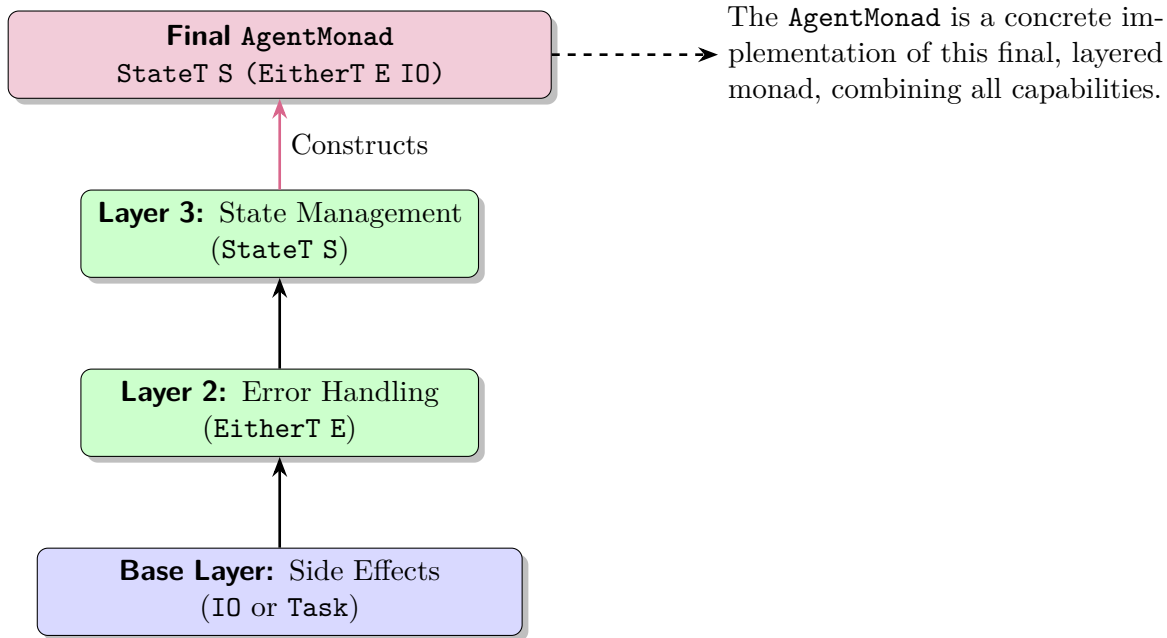
Figure 1 Constructing the `AgentMonad` by stacking monad transformers. Each layer adds a new capability (context) to the monad below it, culminating in a single, unified structure that handles state, errors, and side effects.

mechanism extracts the function and value from their respective contexts and applies them, ensuring state is propagated and failures are bypassed.

2.4 Level 3: AgentMonad as a Monad

The final and most powerful abstraction is the Monad. It addresses the core challenge of agent orchestration: sequencing operations where each step's logic depends on the result of the previous one.

The **bind** operation (often called `flatMap` or `then`) facilitates this chaining. It accepts an `AgentMonad[S, A]` and a function $f : A \rightarrow \text{AgentMonad}[S, B]$. The operation unwraps the value and state from the first context and passes them to f, which returns a new `AgentMonad`. This allows each step to alter the state or fail independently, with the state S passed implicitly by the structure. The logic for `bind` is formalized in Algorithm 1 and visualized in Figure 2.

This structure abstracts away the repetitive and error prone boilerplate of state passing and error checking. The developer can focus entirely on defining the logic of each individual step.
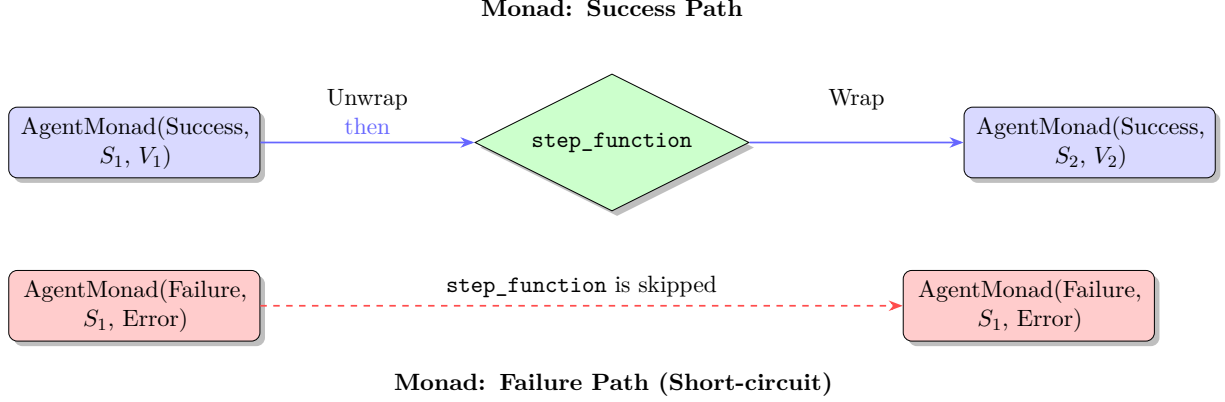
Monad: Success Path

Monad: Failure Path (Short-circuit)

Figure 2 Visualization of the Monad's `bind` operation. In the success path, the function is executed, transforming the entire context. In the failure path, the function is skipped, and the failure is propagated.

Algorithm 1 The `bind` (then) Operation Logic for AgentMonad

1: **procedure** THEN(`current_flow`, `step_function`)
2:          ▷ `current_flow` is an AgentMonad of ($status, state, value$)
3:          ▷ `step_function` is a function: ($state, value$) → AgentMonad$'$
4:      **if** `current_flow`.$status$ is **FAILURE then**
5:          **return** `current_flow`   ▷ Short-circuit: propagate the failure without execution.
6:      **end if**
7:          ▷ If successful, unwrap the container to get the current state and value.
8:      $s \leftarrow$ `current_flow`.$state$
9:      $v \leftarrow$ `current_flow`.$value$
10:          ▷ Execute the next step with the unwrapped values.
11:      **try**
12:      `next_flow` $\leftarrow$ `step_function`(s, v)
13:      **catch** Exception e
14:      `next_flow` $\leftarrow$ AgentMonad(**FAILURE**, s, e)   ▷ Capture runtime exceptions as failures.
15:      **end try**
16:      **return** `next_flow`
17: **end procedure**

---

3 Case Study: A Simple Research Agent

To demonstrate MCE in practice, we present a simple agent that answers the question: *What is a Monad?*. We model the interaction of the agent with its tools using the structure of the Model Context Protocol (MCP) (Anthropic, 2024; Model Context Protocol, 2025), where the agent must process formal tool requests.

The agent logic is decomposed into four composable steps. First, `plan_action` uses an LLM to formulate a plan which resolves into a structured MCP tool call. Second, `execute_tool` consumes the request; if the tool exists, it is executed, and the monadic return value determines the structure of the resulting tool result block. Third, `synthesize_answer` generates a final answer from the tool

output. Finally, `format_output` formats the final answer. Using `AgentMonad`, we chain these steps into a single, declarative workflow.

```
1   task = "What is a Monad?"
2   initial_state = AgentState(task=task)
3
4   # The agent logic is defined as a single, declarative, and robust
        chain.
5   final_flow = (
6       AgentMonad.start(initial_state)
7       .then(lambda s, _: plan_action(s, task))
8       .then(lambda s, call: execute_tool(s, call))
9       .then(synthesize_answer)
10      .then(format_output)
11  )
```

Listing 1 Chaining agent steps using Monadic Context Engineering.

3.1 Robust Failure Handling

The utility of MCE is most apparent during failure states, clarifying its synergy with protocols like MCP. Consider a scenario where the `plan_action` step generates a request for a non-existent tool, such as 'guess'.

In this sequence, the `plan_action` step succeeds, returning an `AgentMonad` in a `Success` state. The monadic chain then passes this object to `execute_tool`. The internal logic of the function attempts to dispatch the tool call, finds no tool named 'guess', and returns an `AgentMonad` in a `Failure` state. This failure corresponds directly to creating an MCP tool result with an error flag.

Crucially, when the subsequent `.then(synthesize_answer)` is called, the `bind` logic immediately detects the failure status and bypasses the rest of the chain. The `synthesize_answer` function is never executed. The failure is propagated to the end of the chain, preserving the error message and the state at the point of failure. The final flow object cleanly reports failure without a single top-level conditional or exception block in the main orchestration logic, demonstrating the inherent resilience of the framework.

4 Extending MCE for Concurrent and Parallel Orchestration

Modern AI agents often interact with multiple external services, such as querying several APIs for data or running different tools to gather diverse information. A purely sequential monadic chain, while robust, creates a performance bottleneck. To address this, MCE must handle asynchronous computations to provide a principled structure for *concurrency* that enables effective I/O *parallelism*.

While our core `AgentMonad` design, built on the transformer stack, can accommodate any base monad, specializing it for asynchronous operations unlocks significant performance gains. By instantiating our stack with a base `Task` or `Future` monad, common in modern programming languages for managing non-blocking I/O, we derive the `AsyncAgentMonad`, a structure purpose-built for high-performance agent orchestration.

4.1 `AsyncAgentMonad`: A Monad Transformer in Practice

The `AsyncAgentMonad` is the concrete implementation of our transformer stack. It provides a single, unified interface for chaining operations that are asynchronous, stateful, and fallible. An `AsyncAgentMonad[S, V]` does not hold a value directly; instead, it holds a *promise* to eventually produce an `AgentMonad[S, V]`. The `bind` (`then`) operation chains asynchronous functions, allowing developers to write non-blocking I/O code that looks clean and sequential.

```
task = "What is a Monad?"
initial_state = AgentState(task=task)

# Each step is now an async function that returns an AgentMonad.
async_flow = (
    AsyncAgentMonad.start(initial_state)
    .then(lambda s, _: async_plan_action(s, task))
    .then(async_execute_tool)
    .then(async_synthesize_answer)
    .then(async_format_output)
)

# The result is itself a promise, which must be awaited to run
    the flow.
final_result_flow = await async_flow.run()
```

Listing 2 Chaining asynchronous steps with `AsyncAgentMonad`.

4.2 Unlocking True Parallelism via the Applicative Interface

The most significant advantage of this extension emerges from the **Applicative** interface. While the Monad's `then` operation is inherently sequential (where step $N + 1$ depends on step N), the Applicative's power lies in combining computations that are *independent* of one another.

When the monadic context involves asynchronicity (like our `AsyncAgentMonad`), this distinction becomes critical. An Applicative combinator, which we will call `gather`, can take a list of independent `AsyncAgentMonad` instances and execute their underlying asynchronous operations *concurrently*. On platforms supporting concurrent I/O or distributed execution, these tasks are effectively parallelized. The `gather` operation initiates all tasks simultaneously and waits for completion. It then collects their results into a single list within a new `AsyncAgentMonad`, correctly propagating state and aborting the entire group if any one of the tasks fails.

For example, consider an agent tasked with creating a daily briefing. It needs to fetch information from several independent sources: a news API, a weather service, and a stock market tracker. These tasks do not depend on each other and can be run in parallel to minimize latency.

```
async def create_daily_briefing(state: AgentState, user_query:
    str) -> AgentMonad:
    # 1. Define independent, asynchronous tasks
    news_task = AsyncAgentMonad.start(state, user_query).then(
    async_fetch_news)
```

```
4       weather_task = AsyncAgentMonad.start(state, user_query).then(
        async_fetch_weather)
5       stocks_task = AsyncAgentMonad.start(state, user_query).then(
        async_fetch_stocks)
6
7       # 2. Execute concurrently via Applicative 'gather'
8       # The result is an AsyncAgentMonad that will resolve to a
        list of results
9       gathered_data_flow = AsyncAgentMonad.gather([news_task,
        weather_task, stocks_task])
10
11      # 3. Synthesize the collected results
12      synthesis_step = await gathered_data_flow.then(
        async_synthesize_briefing).run()
13
14      return synthesis_step
```

Listing 3 Parallel data gathering using an Applicative `gather` operation.
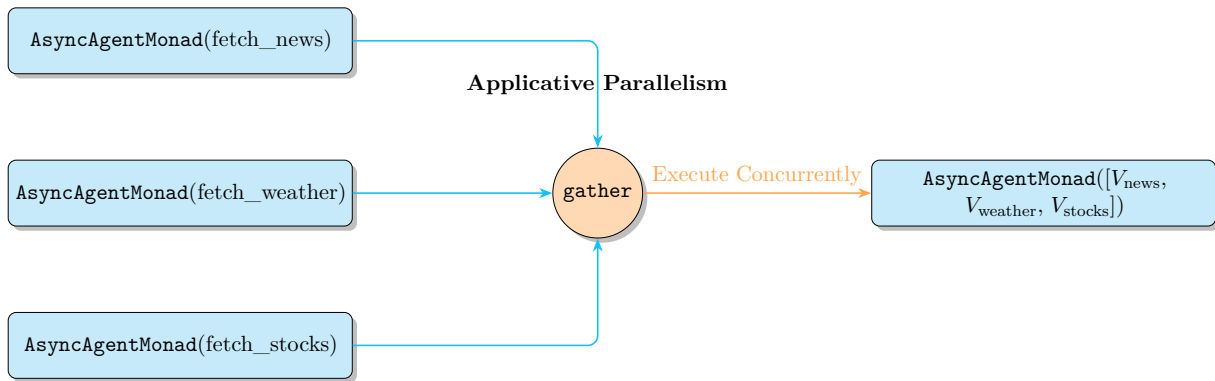

Figure 3 Parallel execution via an Applicative `gather` operation. Three independent asynchronous flows are initiated concurrently. The `gather` combinator waits for all to complete, then merges their results into a single flow. If any task fails, the entire gathered flow fails.

This pattern, visualized in Figure 3, is impossible to express elegantly with a purely monadic chain. A crucial consideration in this parallel model is the handling of state. Since each parallel flow could potentially modify the state, a merge strategy is required. A simple approach is to propagate the state from one predetermined flow, while more complex strategies could involve a custom merge function provided by the developer. Our framework defaults to the former for simplicity but acknowledges the need for more sophisticated state reconciliation in advanced use cases.

The combination of the Monad for sequencing dependent computations and the Applicative for executing independent computations in parallel provides a complete, robust, and high performance toolkit for orchestrating complex agent behaviors.

5 MCE for Meta-Agents: Generative Orchestration

The robustness of an architectural pattern is demonstrated by its ability to scale in complexity and abstraction. We now elevate the MCE paradigm from orchestrating a single agent's workflow to orchestrating a *team* of agents. We introduce a **Meta-Agent**: a higher-level agent whose primary function is not to solve the domain problem directly, but to dynamically create, configure, and supervise a team of specialized sub-agents. This approach is critical for tackling complex, multi-faceted problems that exceed the capabilities of a single monolithic agent.

5.1 The Meta-Agent as a Metaprogrammer

In this model, the Meta-Agent acts as a *metaprogrammer*. Its operations do not manipulate domain data, but rather *computational structures*, specifically, the monadic flows of its sub-agents. The 'AgentMonad' of the Meta-Agent operates at a higher level of abstraction, where the state encompasses the entire system configuration, including active sub-agents and the overall plan. The values produced by steps in a Meta-Agent's flow are often fully formed 'AgentMonad' workflows ready for execution.

The 'bind' operation for a Meta-Agent becomes an act of generative orchestration. A step might take the overall goal, determine that a search capability is required, and output a new 'AsyncAgentMonad' chain pre-configured for a search agent. This dynamically generated workflow is then executed, and its final result is fed back into the Meta-Agent's monadic context for the next step of supervision.

5.2 Meta-Prompting for Dynamic Configuration

A key mechanism for this dynamic configuration is meta-prompting (Zhang et al., 2023; Suzgun and Kalai, 2024). The Meta-Agent uses an LLM not to answer a question, but to generate the prompts and configurations for its sub-agents. For example, given a complex task, the Meta-Agent's first step might be a call to an LLM with a meta-prompt that requests a decomposition of the task into specialized roles. The result of this meta-prompt is then used by the Meta-Agent to programmatically construct and dispatch multiple sub-agents, each with a tailored monadic workflow (Figure 4).
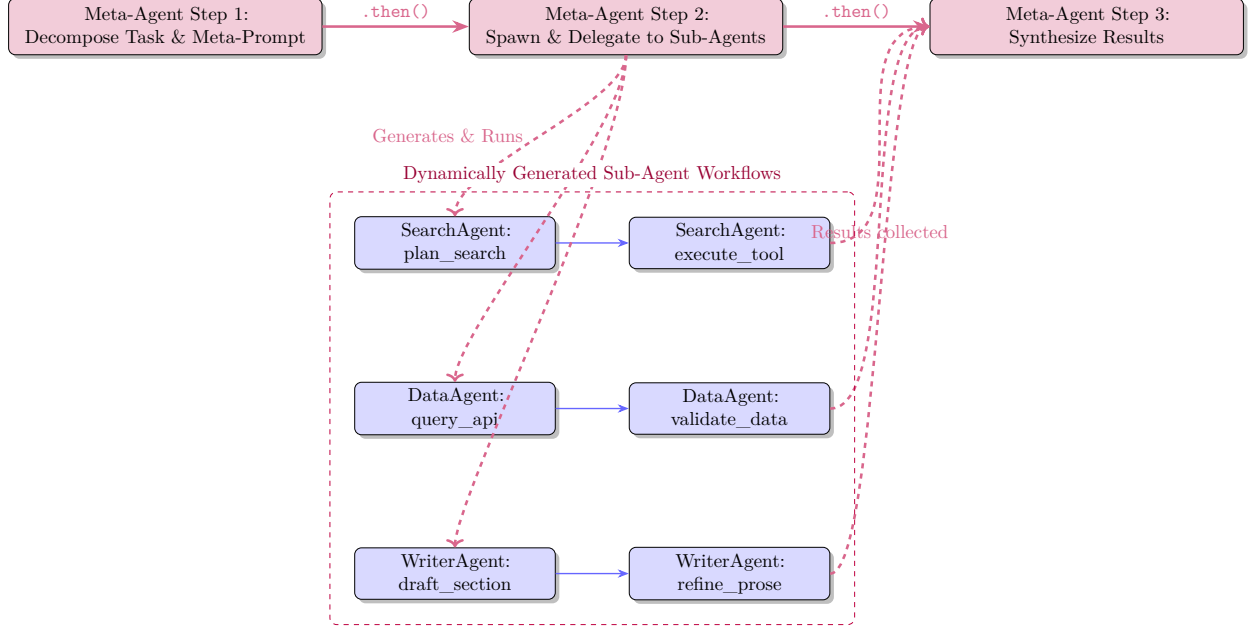
Figure 4 A Meta-Agent's monadic flow. Each step of the Meta-Agent's `then` chain orchestrates the creation and execution of entire monadic workflows for specialized sub-agents. The results are then gathered back into the Meta-Agent's context for synthesis.

6 Related Work

The challenge of orchestrating agentic workflows is not new, and MCE builds upon a rich history of research in both AI and software engineering.

Agent Frameworks. Modern agent toolkits like LangChain (LangChain, 2022) and LlamaIndex (LlamaIndex, 2023) have introduced expression languages to chain components. Their `Runnable` protocol provides a degree of composability, often resembling a Functor or a limited Monad. However, state and error management are frequently handled as side channels rather than being intrinsic to the core abstraction. MCE offers a more formally grounded approach by unifying state, value, and error status into a single, cohesive monadic context, drawing from decades of established practice in functional programming for building robust systems (Hudak et al., 2007).

Category-Theoretic Software Frameworks. Category theory and monadic effect systems have also been used to build large-scale, type-safe software frameworks. Flag Boot is a Scala microservice framework built around functional effects (Cats/Cats Effect) and monadic effect types, emphasizing explicit control flow, predictable scheduling, and high concurrency (Yang et al., 2022; Liu, 2023; InfoQ, 2023). Alongside such system implementations, practitioner-oriented expositions and teaching materials have helped disseminate monadic abstractions for software engineering, including a developer-facing introduction to monads (Yuan, 2022) and documented teaching of type-safe full-stack systems practice (Tsinghua University, 2023).

Multi-Agent Systems. The paradigm of using multiple, collaborating agents to solve complex tasks has gained significant traction, exemplified by systems like AutoGen (Microsoft, 2023) and ChatDev (Qian et al., 2023). These frameworks typically rely on conversational managers or predefined topologies to orchestrate agent interactions. While powerful, their orchestration logic is

often imperative and event-driven, which can make the overall system behavior difficult to predict and verify. MCE offers a complementary formal layer to these systems. A *Meta-Agent* can use a monadic chain to formally define the process of agent creation, task delegation, and result synthesis, bringing the benefits of predictable state and error management to the multi-agent domain.

Reasoning Paradigms. High-level reasoning paradigms like ReAct (Yao et al., 2022), Reflexion (Shinn et al., 2023), and the patterns in AutoGPT (Gravitas, 2023) define the agent's cognitive cycle. MCE is not a replacement for these paradigms; rather, it is a superior low-level implementation framework. An entire ReAct loop can be modeled as a single `AgentMonad` step, which can then be composed with other steps with the guarantee that state and errors are managed robustly throughout.

LLM-Assisted Software Generation and Verification. Beyond interactive agent loops, typed functional abstractions have recently been leveraged in LLM-powered software automation. Xu et al. (Xu et al., 2025) propose a pipeline translating Scala backend code into formal Lean representations and automatically verifying the automatically generated theorems with LLM-based provers. Closely related ideas, using topos-theoretic structure to scaffold large-scale software-assisted generation, have been presented in invited talks at venues such as RLChina 2025, LMG 2025, and FAIC 2025 (RLChina, 2025; Chinese Information Processing Society of China (CIPS), 2025; Institute of Big Data Research, Shanghai University of Finance and Economics, 2025; School of Statistics and Data Science, Shanghai University of Finance and Economics, 2025). MCE addresses the complementary problem of structuring the internal control flow of tool-using agents (planning, tool dispatch, state threading, and failure propagation) rather than verification targets or system-specific code generation.

Model Context Protocol (MCP). Recently, there has been a push to standardize the communication layer between language models and external tools. A prominent example is the Model Context Protocol (MCP) introduced by Anthropic (Anthropic, 2024). MCP proposes a standardized JSON-based format for models to request tool invocations and for the results to be returned to the model. The protocol explicitly includes fields like `tool_id` for tracking requests and an `isError` flag in the result, formalizing the success or failure state of a tool call.

MCE and MCP are highly complementary and operate at different levels of abstraction. MCP standardizes the *data interface*, the format of messages exchanged between the model and the agent's tool execution environment. In contrast, MCE provides a formal structure for the *control flow* within the agent that processes these messages. For instance, an agent built with MCE would receive a `tools_call` request, and the entire process of parsing the request, calling the corresponding tool, handling potential runtime exceptions, and packaging the output or error into an MCP-compliant `tool_result` block can be encapsulated within a single, resilient monadic step. The `EitherT` layer of the `AgentMonad` directly maps to the `isError` flag in the MCP `tool_result`, demonstrating a natural synergy between the two approaches. MCE provides the robust internal engine required to reliably implement the external contract defined by MCP.

Concurrent and Distributed Systems. From a software engineering perspective, MCE is philosophically related to the Actor Model (Hewitt and Baker Jr, 1977), which underpins systems like Erlang/OTP and Akka. Actors are independent agents that manage their own state and communicate via asynchronous messages. While the Actor Model excels at managing highly concurrent, distributed systems, MCE is specifically tailored for the goal-oriented, often sequential but parallelizable workflows of a single logical agent, providing a simpler and more direct abstraction for this common use case, with natural extensions towards parallelism via Applicatives.

7 Conclusion

Monadic Context Engineering provides a paradigm shift for AI agent development, advocating a transition from brittle imperative scripts to a principled, functional architecture. The benefits are immediate and significant. Agent logic becomes a clear, linear sequence of transformations where developers specify *what* to do at each step, while the framework handles *how* state, errors, and asynchronicity are propagated. The error model ensures that failures are handled gracefully and predictably, preventing corrupted states and unexpected crashes. Furthermore, agent behaviors are encapsulated in functions that can be independently tested and composed in novel ways to build increasingly complex agents. Finally, state is managed explicitly through the monadic flow, while the combination of Monad and Applicative interfaces provides a unified model for both sequential and concurrent execution, enabling parallelism.

In essence, MCE is the application of a mature, powerful idea from computer science to address the acute pain points of a new and rapidly evolving domain. By adopting these principled algebraic structures, the AI community can build more reliable, scalable, and understandable agents, laying a solid engineering foundation on the path toward more general and capable artificial intelligence.

References

Anthropic. Introducing the model context protocol. Blog post, November 2024. URL https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-01-21.

Chinese Information Processing Society of China (CIPS). The 4th national conference on large-model intelligent generation (lmg 2025). Conference website, November 2025. URL https://lmg.cipsc.org.cn/conference/lmg2025/index.html. Accessed: 2026-01-21.

Significant Gravitas. Autogpt. https://github.com/Significant-Gravitas/Auto-GPT, 2023.

Carl Hewitt and Henry Baker Jr. Actors and continuous functionals, 1977.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.

InfoQ. Flag boot: A new-generation minimalist open-source microservices framework based on category theory. InfoQ video (Geek Interview), 2023. URL https://www.infoq.cn/video/YoRO74jiv0cOkezLUCxf. Accessed: 2026-01-21.

Institute of Big Data Research, Shanghai University of Finance and Economics. Conference notice for the 2025 faic conference on foundations of artificial intelligence. Conference notice webpage, December 2025. URL https://ibdr.sufe.edu.cn/e0/61/c19466a254049/page.htm. Accessed: 2026-01-21.

LangChain. Langchain. https://github.com/langchain-ai/langchain, 2022.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995.

Yan Liu. Flag boot: A next-generation minimalist open-source microservice framework based on category theory. InfoQ, January 2023. URL https://www.infoq.cn/article/yhajomc93fkrgbegb5fy. Accessed: 2026-01-20.

LlamaIndex. Llamaindex. GitHub repository, 2023. URL https://github.com/run-llama/llama_index. Accessed: 2026-01-21.

Microsoft. AutoGen: A programming framework for agentic AI. https://github.com/microsoft/autogen, 2023. Accessed: July 2025.

Model Context Protocol. Model context protocol specification. Online specification, November 2025. URL https://modelcontextprotocol.io/specification/2025-11-25. Accessed: 2026-01-21.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

RLChina. Rlchina 2025 workshop program. Conference website, September 2025. URL https://rlchina.org/rlchina_2025/Workshop.html. Accessed: 2026-01-20.

Scala Meetup. Activity recap: January 2023 scala meetup. Zhihu Zhuanlan (column post), January 2023. URL https://zhuanlan.zhihu.com/p/599917661. Accessed: 2026-01-21.

School of Statistics and Data Science, Shanghai University of Finance and Economics. Shanghai university of finance and economics successfully hosted the 2025 FAIC conference on AI foundations. News release, December 2025. URL https://ssds.sufe.edu.cn/e1/5b/c1560a254299/page.htm. Accessed: 2026-01-20.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Tsinghua University. Undergraduate training program of tsinghua university: Guiding undergraduate teaching plan for the computer science and technology major (experimental class in computer science), institute for interdisciplinary information sciences. https://www.tsinghua.edu.cn/jxjywj/bkzy2023/zxzy/29-2.pdf, 2023. Includes the course "Type-Safe Frontend and Backend Systems Practice" (accessed 2026-01-20).

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

Kangping Xu, Yifan Luo, Yang Yuan, and Andrew Chi-Chih Yao. Towards automated formal verification of backend systems with llms. *arXiv preprint arXiv:2506.10998*, 2025.

Jingqin Yang, Yang Yuan, Chao Li, and Yu Liu. Flagboot: A fast, modern, light, type-safe microservice architecture for scala. GitHub repository, 2022. URL https://github.com/FlagOpen/FlagBoot. Accessed: 2026-01-20.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yang Yuan. What is a monad useful for ordinary programmers? Zhihu Zhuanlan (column post), October 2022. URL https://zhuanlan.zhihu.com/p/575642401. Accessed: 2026-01-20.

Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. Meta prompting for ai systems. *arXiv preprint arXiv:2311.11482*, 2023.

A   Conceptual Python Implementation

Below are aligned excerpts of the `AgentMonad` and `AsyncAgentMonad` implementations, along with the reference step functions used in this paper.

```python
1  from __future__ import annotations
2
3  from collections.abc import Callable
4  from dataclasses import dataclass
5  from typing import Any, Generic, TypeVar, cast
6
7  from mce.models import AgentState, ToolCall, ToolRegistry,
       default_registry
8
9  S = TypeVar("S")
10 V = TypeVar("V")
11 R = TypeVar("R")
12
13
14 @dataclass(frozen=True)
15 class AgentMonad(Generic[S, V]):
16     """Monadic container for stateful, fallible agent steps."""
17
18     state: S
19     value: V | None
20     is_successful: bool = True
21     error_info: Any = None
22
23     def _require_value(self) -> V:
24         if self.value is None:
25             raise ValueError("AgentMonad has no value.")
26         return self.value
27
28     def then(self, func: Callable[[S, V], AgentMonad[S, R]]) ->
       AgentMonad[S, R]:
29         if not self.is_successful:
30             return AgentMonad.failure(self.state, self.error_info
       )
31         try:
32             value = self._require_value()
33             return func(self.state, value)
34         except Exception as exc:
35             return AgentMonad.failure(self.state, exc)
36
37     def map(self, func: Callable[[V], R]) -> AgentMonad[S, R]:
38         if not self.is_successful:
```

```python
                    return AgentMonad.failure(self.state, self.error_info
    )
        value = self._require_value()
        return AgentMonad.success(self.state, func(value))

    def apply(self, func_flow: AgentMonad[S, Callable[[V], R]])
    -> AgentMonad[S, R]:
        if not self.is_successful or not func_flow.is_successful:
            error = self.error_info if not self.is_successful
    else func_flow.error_info
            return AgentMonad.failure(self.state, error)
        func = func_flow._require_value()
        return self.map(func)

    @staticmethod
    def start(state: S, initial_value: V | None = None) ->
    AgentMonad[S, V]:
        value = initial_value if initial_value is not None else
    cast(V, state)
        return AgentMonad(state, value)

    @staticmethod
    def success(state: S, value: V) -> AgentMonad[S, V]:
        return AgentMonad(state, value, is_successful=True)

    @staticmethod
    def failure(state: S, error_info: Any) -> AgentMonad[S, V]:
        return cast(
            AgentMonad[S, V],
            AgentMonad(state, None, is_successful=False,
    error_info=error_info),
        )


# --- Example: Defining the agent's behavioral steps ---
def plan_action(state: AgentState, task: str) -> AgentMonad[
    AgentState, ToolCall]:
    call = ToolCall(tool_id="tool-1", name="search", arguments={"
    query": task})
    next_state = state.with_history(f"Plan: call {call.name} with
    query='{task}'.")
    return AgentMonad.success(next_state, call)


def execute_tool(
```

```
75        state: AgentState, call: ToolCall, registry: ToolRegistry |
      None = None
76  ) -> AgentMonad[AgentState, str]:
77      registry = registry or default_registry()
78      result = registry.run(state, call)
79      next_state = state.with_history(f"Tool Result ({call.name}):
      {result.content}")
80      if result.is_error:
81          return AgentMonad.failure(next_state, result.content)
82      return AgentMonad.success(next_state, result.content)
83
84
85  def synthesize_answer(state: AgentState, tool_output: str) ->
      AgentMonad[AgentState, str]:
86      answer = (
87          "Monadic Context Engineering structures agent workflows
      as composable steps "
88          "with built-in state threading, error short-circuiting,
      and optional parallelism. "
89          f"Evidence: {tool_output}"
90      )
91      next_state = state.with_history("Synthesized final answer.")
92      return AgentMonad.success(next_state, answer)
93
94
95  def format_output(state: AgentState, answer: str) -> AgentMonad[
      AgentState, str]:
96      formatted = f"Final Report:\n{answer}"
97      next_state = state.with_history("Formatted response for
      delivery.")
98      return AgentMonad.success(next_state, formatted)
```

Listing 4 Conceptual Implementation of the AgentMonad Class.

B   Conceptual AsyncAgentMonad Implementation

```
1  from __future__ import annotations
2
3  import asyncio
4  from collections.abc import Awaitable, Callable, Sequence
5  from typing import Any, Generic, TypeVar, cast
6
7  S = TypeVar("S")
8  V = TypeVar("V")
```

```python
R = TypeVar("R")

AsyncStep = Callable[[S, V], Awaitable[AgentMonad[S, R]]]


class AsyncAgentMonad(Generic[S, V]):
    """Async monadic container for parallel, stateful, fallible
    workflows."""

    def __init__(self, run_func: Callable[[], Awaitable[
    AgentMonad[S, V]]]) -> None:
        self._run = run_func

    async def run(self) -> AgentMonad[S, V]:
        return await self._run()

    def then(self, func: AsyncStep[S, V, R]) -> AsyncAgentMonad[S
    , R]:
        async def new_run() -> AgentMonad[S, R]:
            current_flow = await self.run()
            if not current_flow.is_successful:
                return AgentMonad.failure(current_flow.state,
    current_flow.error_info)
            try:
                value = current_flow._require_value()
                return await func(current_flow.state, value)
            except Exception as exc:
                return AgentMonad.failure(current_flow.state, exc
    )

        return AsyncAgentMonad(new_run)

    @staticmethod
    def start(state: S, initial_value: V | None = None) ->
    AsyncAgentMonad[S, V]:
        async def run_func() -> AgentMonad[S, V]:
            return AgentMonad.start(state, initial_value)

        return AsyncAgentMonad(run_func)

    @staticmethod
    def gather(
        flows: Sequence[AsyncAgentMonad[S, Any]],
        merge_state: Callable[[Sequence[S]], S] | None = None,
    ) -> AsyncAgentMonad[S, list[Any]]:
```

```
48    async def new_run() -> AgentMonad[S, list[Any]]:
49        if not flows:
50            return AgentMonad.failure(cast(S, None), "No
      flows provided")
51
52        results = await asyncio.gather(*(flow.run() for flow
      in flows))
53        errors = [result for result in results if not result.
      is_successful]
54        if errors:
55            failing = errors[0]
56            return AgentMonad.failure(failing.state, failing.
      error_info)
57
58        states = [result.state for result in results]
59        final_state = merge_state(states) if merge_state else
       states[-1]
60        values = [result.value for result in results]
61        return AgentMonad.success(final_state, values)
62
63    return AsyncAgentMonad(new_run)
```

Listing 5 Conceptual Implementation of AsyncAgentMonad.