CodeDelegator: Mitigating Context Pollution via Role Separation in Code-as-Action Agents

Tianxiang Fei, **Cheng Chen**, **Yue Pan**, **Mao Zheng** and **Mingyang Song**
Large Language Model Department, Tencent
{alvinfei, cianchen, nobitapan, moonzheng, nickmysong}@tencent.com

Abstract

Recent advances in large language models (LLMs) allow agents to represent actions as executable code, offering greater expressivity than traditional tool-calling. However, real-world tasks often demand both strategic planning and detailed implementation. Using a single agent for both leads to context pollution from debugging traces and intermediate failures, impairing long-horizon performance. We propose **CODEDELEGATOR**, a multi-agent framework that separates planning from implementation via role specialization. A persistent *Delegator* maintains strategic oversight by decomposing tasks, writing specifications, and monitoring progress without executing code. For each sub-task, a new *Coder* agent is instantiated with a clean context containing only its specification, shielding it from prior failures. To coordinate between agents, we introduce **Ephemeral-Persistent State Separation (EPSS)**, which isolates each Coder's execution state while preserving global coherence, preventing debugging traces from polluting the Delegator's context. Experiments on various benchmarks demonstrate the effectiveness of CODEDELEGATOR across diverse scenarios. Code is available at https://github.com/txfei/code-delegator.

1 Introduction

The integration of external tools into Large Language Models (LLMs) marks a fundamental advance, enabling complex problem-solving through real-world interaction [Qu *et al.*, 2025; Liu *et al.*, 2025b]. ReAct [Yao *et al.*, 2023] exemplifies this capability through an iterative *Thought-Action-Observation* loop. However, ReAct-based frameworks typically represent actions as JSON or structured text, which constrains their expressiveness to predefined schemas and leads to linearly increasing context length, thereby reducing execution efficiency (Fig. 1(a)). The "code-as-action" paradigm (e.g., CodeAct [Wang *et al.*, 2024b]) addresses this by adopting executable Python as a unified action representation, naturally supporting control flow, variable management, and multi-tool composition within a single step.

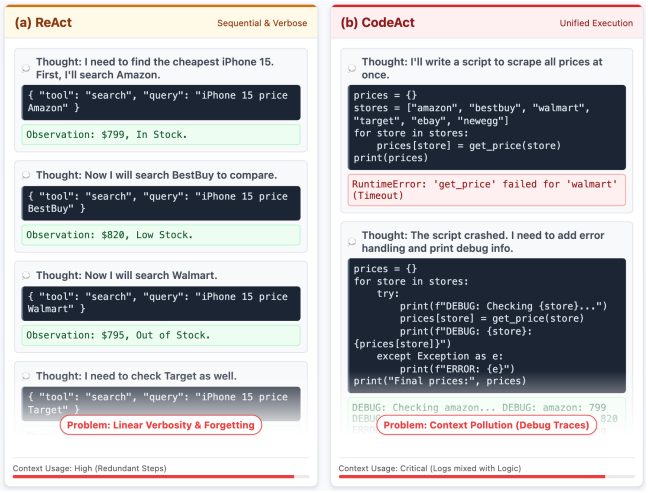

Figure 1: Comparison of action representations: ReAct uses structured text/JSON, while CodeAct uses executable Python code, enabling more complex logic within a single action.

However, when handling real-world, long-horizon tasks that interact with complex environments and humans, code-as-action agents encounter a fundamental tension. Consider a task "analyze competitor iPhone 15 pricing data and generate a market report": the agent must strategically decompose this into sub-tasks (data collection, data cleaning, analysis report generation) and reason about dependencies. For each sub-task, it must generate syntactically correct code with proper API calls and error handling. These objectives compete for attention within a fixed context window, where later tokens progressively overshadow earlier content [Liu *et al.*, 2024]. Moreover, planning requires abstract reasoning, while implementation demands fine-grained attention to detail. When interleaved, planning steps and debugging traces accumulate in shared context. We refer to this as *context pollution*: the progressive dilution of task-relevant information that degrades both planning and implementation capabilities (Fig. 1(b)).

To address this tension, we propose **CODEDELEGATOR**, which decouples strategic planning from code implementation through role separation (Fig. 2). A Delegator maintains strategic focus: decomposing complex tasks, analyzing dependencies, monitoring progress, and dynamically dis-

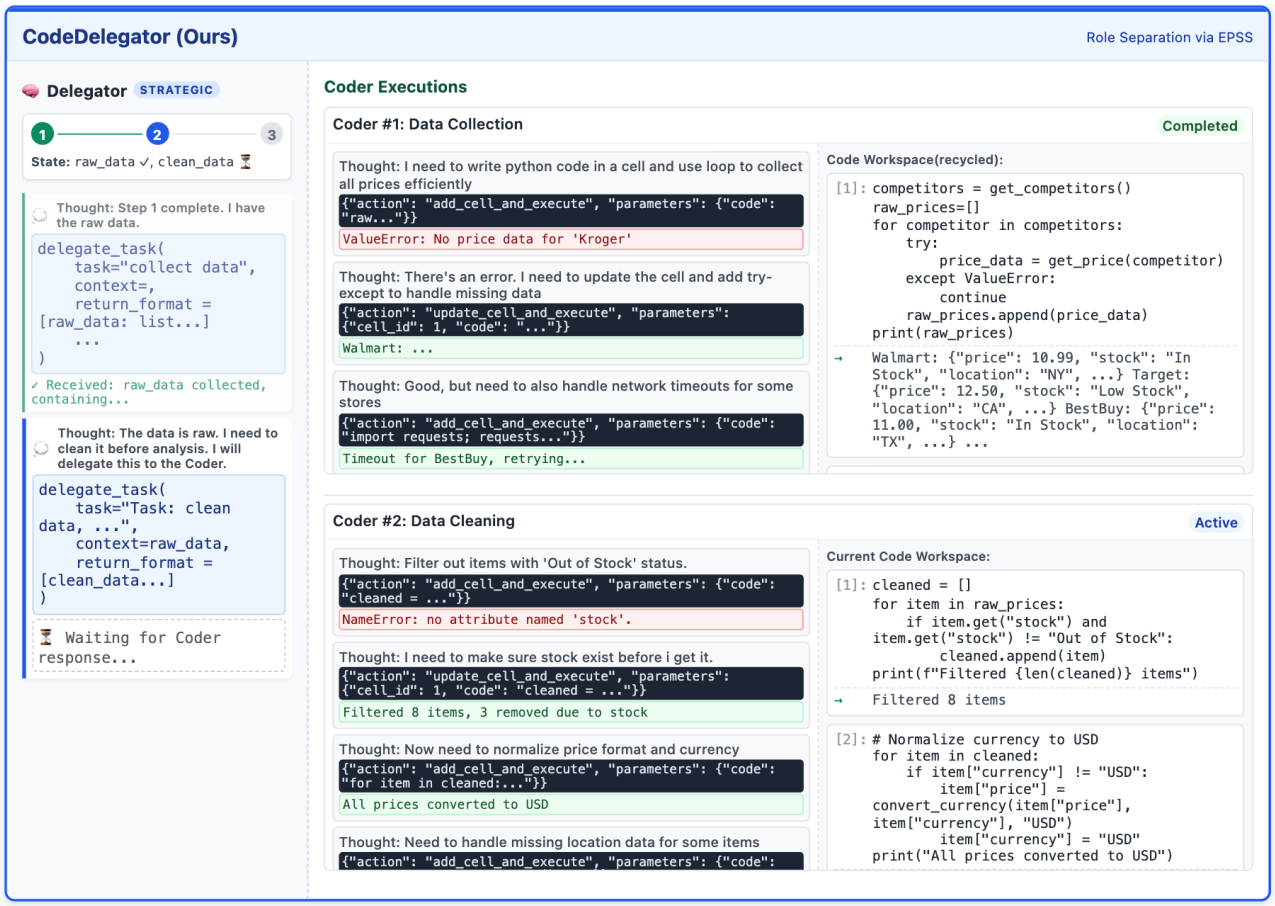

Figure 2: CodeDelegator in action: The persistent Delegator orchestrates a pricing analysis task by delegating sub-tasks to ephemeral Coders. Coder #1 (Data Collection) has completed and been discarded; Coder #2 (Data Cleaning) is actively debugging in an isolated context. Each Coder's execution traces remain confined and never propagate to the Delegator's planning context, preventing context pollution.

patching work, without ever writing implementation code. Ephemeral Coders execute assigned sub-tasks through an interactive loop: generating code, observing execution results, diagnosing errors, and iteratively refining until completion. Each Coder is spawned by the Delegator with a fresh context containing only task-relevant information, preventing accumulation of irrelevant history from other sub-tasks. A key challenge is enabling effective coordination without reintroducing context pollution. We address this through **Ephemeral-Persistent State Separation (EPSS)**, a dual-layer workspace providing code-aware isolation beyond mere conversation separation. The Delegator governs a persistent Orchestration Layer containing global state, task plans, and committed results. Each Coder operates in an ephemeral execution sandbox where debugging traces and local variables remain confined and are discarded upon completion. Structured schemas govern all inter-agent communication, replacing lossy natural language summaries with typed specifications. Our contributions are as follows:

- We identify and empirically characterize the *context pollution* problem in code-as-action agents (§ 3).
- We propose CODEDELEGATOR (§ 4), a targeted isolation mechanism for code-as-action agents comprising: (a) ephemeral workers that prevent cross-subtask pollution, (b) asymmetric schema-constrained communication that filters execution noise, and (c) EPSS for code-aware dual-layer state separation; distinct from generic planner-executor orchestration in general multi-agent frameworks.
- Experiments (§ 5) on τ^2-bench and MCPMark demonstrate that CODEDELEGATOR achieves substantial improvement over baselines.

2 Related Work

The remarkable reasoning capabilities of Large Language Models have spurred research into autonomous agents that use tools to solve complex tasks [Wei *et al.*, 2022; Huang *et al.*, 2022; Yao *et al.*, 2023; Qu *et al.*, 2025]. The *plan-execute-observe* loop introduced by ReAct [Yao *et al.*, 2023] has become foundational for many agent systems [Yang *et al.*, 2023; Wu *et al.*, 2024; Zhang *et al.*, 2024; Hu *et al.*, 2025].

2.1 Code-as-Action LLM Agents

CodeAct [Wang *et al.*, 2024b] demonstrates that executable Python code naturally supports control flow, variable management, and multi-tool composition, outperforming ReAct on complex tasks. This paradigm has been adopted by OpenHands [Wang *et al.*, 2025], SWE-agent [Yang *et al.*, 2024], and Voyager [Wang *et al.*, 2024a]. Recent work explores hierarchical code generation to enhance planning ability in code-as-action agents. Liu *et al.* generates abstract code skeletons for progressive refinement. Yu *et al.* introduces recursive decomposition of placeholder functions. These approaches address *what* to implement versus *how*. However, planning and implementation still share a single context, so debugging traces from implementing step i accumulate when planning step $i{+}1$. CODEDELEGATOR addresses an orthogonal *isolation* concern: the Delegator and Coders operate in separate contexts, ensuring that no implementation detail, regardless of abstraction level, ever reaches the planning process.

2.2 Multi-Agent Collaboration

Multi-agent collaboration leverages specialized roles for complex tasks, as exemplified by AutoGen [Wu *et al.*, 2024], MetaGPT [Hong *et al.*, 2023], and CAMEL [Li *et al.*, 2023]. However, Cemri *et al.* identified recurring coordination failures including specification drift, inter-agent misalignment, and verification gaps. CODEDELEGATOR differs from these frameworks by targeting context pollution specifically: agents are ephemeral rather than persistent, communication is structurally asymmetric via typed schemas, and isolation operates at the runtime level (namespace, interpreter, artifacts), not merely at the conversation-history level.

3 Understanding Context Pollution in Code-as-Action Agents

We define *context pollution* in code-as-action agents as the progressive dilution of task-relevant information in shared context: debugging traces from sub-task s_i persist during s_j, competing for attention when they are irrelevant. To validate that pollution, not task difficulty alone, limits code-as-action agents, we conduct a pilot study on CodeAct.

Setup. We evaluate CodeAct on 50 tasks from the airline domain of τ^2-Bench [Barres *et al.*, 2025], with both the agent and the simulated user powered by DeepSeekV3.2 [DeepSeek-AI *et al.*, 2025]. We record task success, final context length, and cumulative code execution errors, and stratify tasks by intrinsic complexity to disentangle pollution from task difficulty.

Task Complexity. We compute the average success rate of four official τ^2-bench (airline) baselines—Claude-3.7-Sonnet [Anthropic, 2025], GPT-4.1 [OpenAI, 2025a], GPT-4.1-mini [OpenAI, 2025a], and o4-mini [OpenAI, 2025c]. Each model is evaluated over 4 independent runs, yielding 16 trials per task. We categorize tasks into Low (SR $\geq 75\%$, 17 tasks), Medium ($25\%<$SR$<75\%$, 20 tasks), and High (SR $\leq 25\%$, 13 tasks).

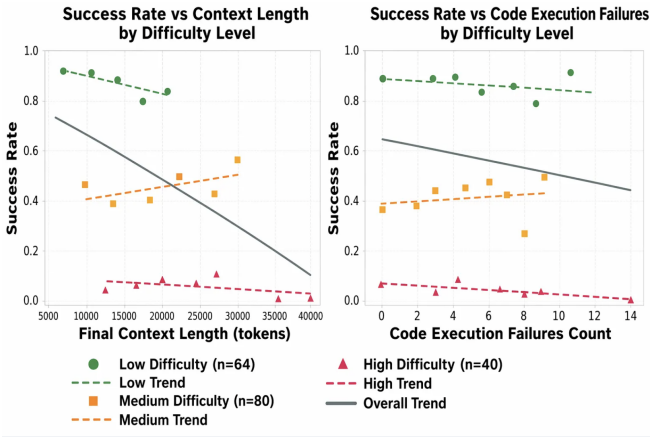

Figure 3: Illustrative examples of agent trajectories and failure modes observed in the pilot study.

Difficulty	ReAct		CodeAct		Diff.
	SR	CL	SR	CL	
Low (n=17)	88.2	7.8K	82.4	12.4K	-5.8
Medium (n=20)	62.5	10.5K	60.0	16.3K	-2.5
High (n=13)	19.2	10.7K	26.9	21.1K	+7.7
Overall	60.0	9.4K	59.0	15.7K	-1.0

Table 1: Performance comparison of ReAct vs. CodeAct by difficulty level on τ^2-bench airline domain. SR: success rate (%); CL: average context length (tokens); Diff.: CodeAct SR − ReAct SR.

Results. Table 1 shows that CodeAct incurs more context length across all difficulty levels compared to ReAct. However, this overhead yields inconsistent returns: CodeAct underperforms ReAct on Low (-5.8%) and Medium (-2.5%) difficulty tasks, while outperforming on High difficulty tasks ($+7.7\%$). Fig. 3 provides further insight: within the CodeAct runs, context length negatively correlates with success on both Low and High difficulty tasks.

Analysis. This pattern reveals where code-as-action provides value and where it incurs cost: the additional context is pure overhead on simpler tasks, but pays off on complex ones requiring multi-step logic. Yet even on High tasks, the absolute success rate remains low (26.9%) and context length is substantial (21.1K tokens). The negative correlation between context length and success within High-difficulty tasks suggests that *cross-sub-task trace accumulation* still degrades performance, even when code flexibility helps. This motivates CODEDELEGATOR: retain the expressiveness benefits of code-as-action for complex sub-tasks while isolating each sub-task's execution to prevent traces from polluting subsequent planning.

4 Methodology

CODEDELEGATOR separates strategic planning from code implementation via role specialization: a persistent *Delegator* orchestrates task decomposition, while ephemeral *Coders* execute atomic sub-tasks in isolated contexts, coordinated

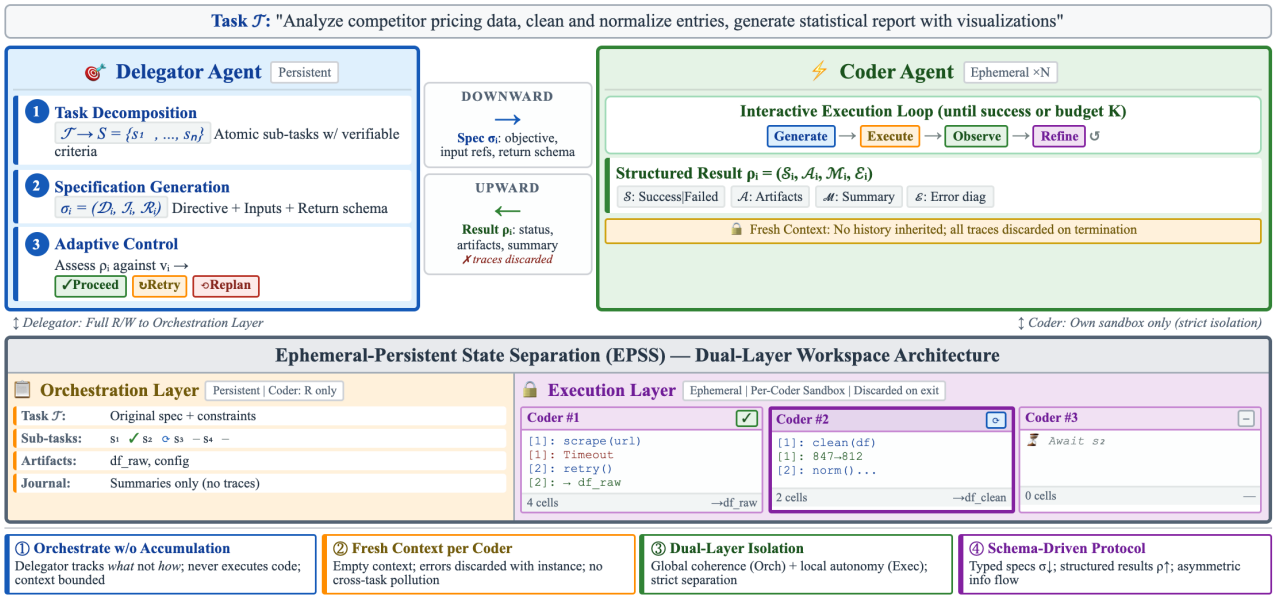

Figure 4: Overview of the CodeDelegator framework with two core mechanisms. **Role Separation** (top): A persistent Delegator decomposes tasks and performs adaptive control, while ephemeral Coders execute sub-tasks through interactive refinement with fresh contexts. **EPSS** (bottom): The dual-layer workspace architecture isolates execution traces in per-Coder sandboxes (Execution Layer) while maintaining global coherence in a persistent Orchestration Layer, enabling four key properties: orchestration without accumulation, fresh context per Coder, dual-layer isolation, and schema-driven communication.

through **Ephemeral-Persistent State Separation (EPSS)**. Fig. 4 gives an overview. Our design follows four principles derived from the pilot study (§3). **(P1) Orchestrate without accumulation:** the Delegator tracks *what* is accomplished via workspace state, never *how*, keeping its context bounded to $O(n)$ task-structure summaries. **(P2) Fresh context per sub-task:** each Coder starts empty, receives only its specification plus designated inputs, and is discarded with all traces upon returning a structured result. **(P3) Context-runtime decoupling:** the reasoning context, sandboxed runtime, and persistent artifact store are separated, with inter-module communication using typed object references rather than printed strings to preserve type fidelity and avoid namespace collisions. **(P4) Schema-driven communication:** specifications (σ_i) and results (ρ_i) use typed schemas—downward: directive, inputs, return schema; upward: status, artifacts, summary, diagnostics—enforcing asymmetry between complete spec and filtered result. The rest of this section instantiates these principles through the agent architecture (§4.1), the EPSS workspace (§4.2), and the execution protocol (§4.3).

4.1 Agent Architecture

CODEDELEGATOR employs two agent types with asymmetric responsibilities, lifespans, and context access patterns.

Delegator Agent. The Delegator is a long-lived agent that maintains strategic oversight throughout the task lifecycle. It performs three core functions:

- **Task Decomposition.** Given a complex task $\mathcal{T}$, the Delegator decomposes it into a sequence of sub-tasks $S = \{s_1, \ldots, s_n\}$. It is instructed to identify logical sub-units that can be implemented independently, specify their execution order, and ensure each sub-task has a clear completion criterion. The decomposition satisfies *atomicity* and *verifiability*: each sub-task is small enough to complete within a single Coder session and has a clear expected outcome that the Delegator can assess upon completion. If a sub-task later proves too complex, the REPLAN mechanism allows dynamic re-decomposition.
- **Sub-task Delegation.** For each sub-task s_i, the Delegator constructs a structured specification $\sigma_i = (\mathcal{D}_i, \mathcal{I}_i, \mathcal{R}_i)$, where $\mathcal{D}_i$ is a *directive* describing the goal and constraints in natural language; $\mathcal{I}_i$ denotes *input bindings* as references to Orchestration Layer objects, annotated with type, shape, and sample values; and $\mathcal{R}_i$ is a *return schema* specifying expected output names, types, and validation conditions. The Delegator then dispatches σ_i to a freshly initialized Coder.
- **Adaptive Progress Control.** After each Coder terminates, the Delegator assesses the structured result ρ_i (§4.1) to determine whether the sub-task has been satisfactorily completed, based on the reported status, output summary $\mathcal{M}_i$, and error diagnostics $\mathcal{E}_i$. It then selects one of three actions: (i) PROCEED—commit the successful Coder's outputs to the Orchestration Layer and advance to the next sub-task; (ii) RETRY—spawn a new Coder with a refined specification when the failure is judged recoverable (typically specification ambiguity or missing context rather than fundamental decomposition flaws); (iii) REPLAN—revise the overall task decomposition when failure is attributed to structural issues or the retry limit is reached.

Coder Agent. Each Coder is a short-lived specialist implementing exactly one atomic sub-task. Its defining characteristic is context isolation: it inherits no history from the Delegator's planning process or sibling Coders' implementations.

- **Interactive Execution.** A fresh Coder receives the specification σ_i, runtime access to designated input variables, and an isolated execution environment. It operates in an interactive loop analogous to computational notebooks: generate code, execute, observe outputs or errors, and refine based on feedback—using runtime errors as actionable signals for iterative refinement. The loop terminates when the Coder produces all outputs specified in $\mathcal{R}_i$, encounters an unrecoverable error, or exhausts its iteration budget K.
- **Structured Result.** Upon completion, each Coder returns a structured result $\rho_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{M}_i, \mathcal{E}_i)$: $\mathcal{S}_i \in \{\text{SUCCESS}, \text{FAIL}\}$ is the completion status; $\mathcal{A}_i$ contains output artifacts as references to Python objects satisfying $\mathcal{R}_i$ (populated only on SUCCESS); $\mathcal{M}_i$ is a concise summary of the outcome; and $\mathcal{E}_i$ provides brief error diagnostics with root-cause explanation (only on FAIL). Critically, full execution histories, intermediate code cells, debugging traces and failed attempts are *not* returned—they are discarded with the Coder instance, enforcing asymmetric information flow that prevents context pollution. The structured diagnostics $\mathcal{E}_i$ summarize root causes and failed operations, preserving what the Delegator needs to decide between RETRY and REPLAN while filtering implementation noise.

4.2 Ephemeral-Persistent State Separation

A naive approach—simple context separation—gives each agent an independent conversation history. However, code-as-action agents require deeper isolation: they exchange actual Python objects that lose fidelity when described as text, operate in interpreter namespaces where variable collisions cause subtle bugs, and generate verbose traces that must be filtered rather than merely truncated. We propose **Ephemeral-Persistent State Separation (EPSS)**, a code-aware dual-layer architecture isolating conversation context, execution namespace, and runtime artifacts.

Orchestration Layer (Persistent). As illustrated in Fig. 4, the Delegator maintains a session-wide workspace providing task tracking, typed artifact storage, and event logging. It contains: (i) the task specification $\mathcal{T}$ and decomposition S with per-sub-task status tracking; (ii) *committed artifacts*, i.e., Python objects from successful Coders, indexed by name with type annotations; and (iii) a progress journal recording outcome summaries. Artifacts are stored as actual objects, not textual descriptions. When sub-task 2 produces `df_clean`, subsequent Coders receive the DataFrame itself, preserving type, shape, and content without lossy natural language encoding.

Execution Layer (Ephemeral). Each Coder operates in an *isolated Python runtime* that provides: (i) *namespace isolation*—variables do not collide across Coders or persist across retries; (ii) *trace confinement*—stack traces, print outputs, and intermediate results remain local; and (iii) *clean state*—a deterministic environment without pollution from prior executions. Upon termination, the runtime is discarded entirely; only the structured result ρ_i, containing artifact *references*, returns to the Delegator.

Sub-task 2: Clean and normalize pricing data
Directive: Remove duplicate rows by `product_id`. Forward-fill missing `price` values; drop rows where `product_id` is null. Convert all prices to USD using provided exchange rates.
Inputs: `df_raw` (`pd.DataFrame`, shape 847×12); `exchange_rates` (`Dict[str, float]`)
Return: `df_clean` (`pd.DataFrame`, no nulls in `price` or `product_id`, prices in USD)

⇓ *Coder executes and returns*

Result ρ_2: Clean and normalize pricing data
Status: SUCCESS
Artifacts: `df_clean` (`pd.DataFrame`, shape 821×12, no nulls in `price` or `product_id`)
Summary: Deduplicated, forward-filled missing prices, converted to USD.
Error: `None`

Figure 5: Asymmetric communication example. Top: specification σ_i sent to Coder. Bottom: structured result ρ_i returned to Delegator. Debugging traces remain confined to the Execution Layer.

Typed, Asymmetric Communication. Unlike systems relying on natural language handoffs, EPSS uses schema-driven messaging, as shown in Fig. 5. Communication is structurally asymmetric: *downward*, the Delegator sends a specification σ_i with typed input bindings (object references plus annotations); *upward*, the Coder returns a result ρ_i with artifact references, summary, and diagnostics, but no raw execution traces. This enables Coders to manipulate large objects without printing them into context, while the Delegator maintains a compact $O(n)$ planning state. Table 2 summarizes how the Orchestration and Execution Layers differ in lifecycle, state management, information flow, and access control.

Category	Orch. Layer	Exec. Layer
Lifecycle		
Lifespan	Session-wide	Sub-task scoped
Cardinality	Singleton (shared)	Per-Coder (isolated)
Owner	Delegator	Ephemeral Coder
State Management		
Contents	Sub-tasks, artifacts, journal	Code cells, local vars, traces
On exit	Absorbs ρ_i selectively	Entirely discarded
Information Flow		
Sends (↓)	Specs σ_i, input refs	—
Receives (↑)	ρ_i filtered results	—
Propagates (↑)	—	Status, artifacts, summary
Discarded	—	Debug traces, failed attempts
Access Control		
Delegator	Full R/W	None
Coder	Input refs (R-only)	Full R/W

Table 2: Workspace architecture under EPSS.

Algorithm 1 CODEDELEGATOR Execution Protocol

Require: Task $\mathcal{T}$, retry budget R, iteration budget K
Ensure: Orchestration Layer $\mathcal{W}$ or FAILURE

$S \leftarrow \text{DECOMPOSE}(\mathcal{T})$
$\mathcal{W} \leftarrow \emptyset$
while S has pending sub-tasks **do**
  $s_i \leftarrow$ next pending sub-task in S
  resolved $\leftarrow$ FALSE
  for $r \leftarrow 1$ to R **do**
    $\sigma_i \leftarrow \text{GENSPEC}(s_i, \mathcal{W})$
    $\rho_i \leftarrow \text{EXECCODER}(\sigma_i, K)$
    if $\rho_i.\mathcal{S} = \text{SUCCESS}$ **then**
      $\mathcal{W} \leftarrow \mathcal{W} \cup \rho_i.\mathcal{A}$
      resolved $\leftarrow$ TRUE
      break
    else if $\text{ISRECOVERABLE}(\rho_i.\mathcal{E})$ **then**
      continue ▷ Retry with refined spec
    else
      $S \leftarrow \text{REPLAN}(S, s_i, \rho_i.\mathcal{E})$
      resolved $\leftarrow$ TRUE ▷ Handled via replan
      break
    end if
  end for
  if $\neg$*resolved* **then**
    return $\text{FAILURE}(s_i, \rho_i.\mathcal{E})$
  end if
end while
return $\mathcal{W}$

4.3 Execution Protocol

Algorithm 1 summarizes the end-to-end execution protocol. The Delegator first decomposes task $\mathcal{T}$ into sub-tasks $S = \{s_1, \ldots, s_n\}$. For each sub-task whose dependencies are satisfied, it generates a specification σ_i and spawns a fresh Coder, which tackles the sub-task through iterative coding with tool invocations until either success or the iteration budget K is exhausted. Based on the returned status and error diagnostics $\mathcal{E}_i$, the Delegator evaluates completion and decides the next action: on success with evaluation criteria met, it commits the artifacts to the Orchestration Layer and proceeds to the next sub-task; on failure or unmet criteria, it analyzes the diagnostics—if the issue is resolvable through specification refinement, it retries with a fresh Coder instance; otherwise, it invokes REPLAN to split the problematic sub-task, add missing dependencies, or reformulate requirements. Execution terminates successfully when all sub-tasks complete, or fails when retry or replan budgets are exhausted.

5 Experiments

5.1 Experiment Setup

Environments. We evaluate CODEDELEGATOR under two interaction paradigms. (1) **Human-Agent-Environment interaction (HAE)** (§5.2): the agent converses with users while invoking tools to interact with the environment. We adopt τ^2-bench [Barres *et al.*, 2025], which simulates customer service scenarios involving multi-turn dialogues and backend operations (e.g., order lookup, ticket modification), covering retail (115 tasks) and airline (50 tasks) domains. (2) **Agent-Environment-only interaction (AE)** (§5.3): the user specifies a task upfront, after which the agent interacts with the environment autonomously without further dialogue. We use MCPMark [Wu *et al.*, 2025], which provides 127 expert-curated tasks requiring extensive CRUD operations with 5 MCP services: Filesystem, GitHub, Notion, Playwright and PostgreSQL. Together, these benchmarks assess agent capabilities across diverse interaction patterns and complexity levels.

Baselines. We compare against two representative action-representation paradigms spanning the expressiveness spectrum. (1) *ReAct* [Yao *et al.*, 2023]: interleaves reasoning and action with structured text, enabling explicit deliberation but limited to one tool invocation per step. (2) *CodeAct* [Wang *et al.*, 2024b]: employs executable Python as the action space, supporting control flow and multi-tool composition within single actions. To ensure compatibility with contemporary tool-use frameworks and model completion APIs, we adapt CodeAct's original free-form code format to a JSON-based function call interface where code is passed as a string argument—preserving its expressiveness while enabling standardized evaluation. Both baselines operate as single-agent systems with shared context across the entire task, isolating the effect of our proposed hierarchical delegation and EPSS mechanisms.

Evaluation. For τ^2-bench, we report pass^k ($k \in \{1, 2, 3, 4\}$) (%) [Yao *et al.*, 2024] across its two domains: retail and airline. A task is considered passed only if all user requirements are satisfied and all backend state changes are correct. For MCPMark, we report pass@1 as the primary metric, where success requires correct tool invocations, proper output parsing, and ground-truth-matching results verified by programmatic scripts.

Implementation Details. For τ^2-bench, we employ DeepSeekV3.2 [DeepSeek-AI *et al.*, 2025] in fast-thinking mode and GPT-5 [OpenAI, 2025b] at low reasoning level as backbone models to ensure fair comparison, and following the τ^2-bench framework requirements which require an LLM-based user simulator for multi-turn dialogue, we use DeepSeekV3.2 as the simulated user. For MCPMark, we adopt DeepSeekV3.2 as the sole backbone model due to cost considerations. The temperature is set to 0 across all experiments to ensure reproducibility. For ReAct and CodeAct baselines, we set the maximum number of interaction steps to 200 and the maximum tolerated errors to 10. For CODEDELEGATOR, the Delegator component is limited to 100 dispatch rounds while each Coder instance is constrained to 20 iterations, where the Delegator and Coders share the same base model but operate within separate contexts.

5.2 Results on HAE Scenario

Table 3 presents results on τ^2-bench. With DeepSeekV3.2, CODEDELEGATOR achieves the highest pass^1 accuracy across both domains, reaching 82.0% on Retail and 63.5% on Airline. CodeAct consistently underperforms ReAct despite its greater expressiveness, confirming that context pollution negates the benefits of code-based actions in long-horizon tasks. The advantage of CODEDELEGATOR becomes more

Model	Domain	Method	pass^1	pass^2	pass^3	pass^4
DeepseekV3.2	Retail	ReAct	79.6	69.9	63.4	58.8
		CodeAct	70.2	59.0	50.0	47.0
		CODEDELEGATOR	**82.0**	**71.2**	**63.4**	57.0
	Airline	ReAct	58.5	47.0	40.5	36.0
		CodeAct	59.0	45.3	37.0	32.0
		CODEDELEGATOR	**63.5**	**53.7**	**48.5**	**46.0**
GPT5.0	Retail	ReAct	78.7	68.9	62.9	58.8
		CodeAct	75.0	65.3	59.3	57.1
		CODEDELEGATOR	**80.9**	**70.3**	**64.5**	**60.5**
	Airline	ReAct	57.0	49.3	45.0	42.0
		CodeAct	56.5	47.3	41.5	40.0
		CODEDELEGATOR	**62.0**	**50.7**	44.5	**42.0**

Table 3: Results on τ^2-bench. We report pass^k accuracy (%) for $k \in \{1, 2, 3, 4\}$.

Domain	ReAct	CodeAct	CODEDELEGATOR
Filesystem	35.0	39.0	**49.2**
GitHub	18.5	27.9	**38.0**
Notion	13.4	19.4	**34.8**
Playwright	13.0	9.7	**27.0**
PostgreSQL	52.4	36.2	41.7
Total	25.8	26.4	**38.4**

Table 4: Results on MCPMark (DeepSeekV3.2). We report pass@1 (%).

pronounced under stricter consistency requirements: on Airline, the pass^1 $\rightarrow$ pass^4 degradation is 17.5% for CODEDELEGATOR versus 27.0% for CodeAct, with CODEDELEGATOR outperforming CodeAct by 14.0% absolute at pass^4. The Airline domain, involving longer action sequences and more complex API operations, better reveals the benefits of role separation by isolating execution traces from subsequent planning steps. Results with GPT-5 show similar trends, validating that context pollution is a model-agnostic limitation.

5.3 Results on AE Scenario

Table 4 presents results on MCPMark across five MCP tool domains. CODEDELEGATOR achieves the highest overall success rate at 38.4%, outperforming both ReAct (25.8%) and CodeAct (26.4%) by approximately 12% absolute. Despite its greater expressiveness, CodeAct performs nearly identically to ReAct, consistent with our findings that context pollution offsets the flexibility of code-based actions. The advantage of CODEDELEGATOR is particularly pronounced in domains requiring complex multi-step interactions: GitHub (+10.1% over CodeAct), Notion (+15.4%), and Playwright (+17.3%), where role separation and EPSS provide the greatest benefit for tracking intermediate state across API calls. The exception is PostgreSQL, where ReAct achieves the highest accuracy at 52.4%. We attribute this to transactional semantics: SQL statements wrapped in code loops can cause partial failures that leave inconsistent states, whereas ReAct's single-call-per-step pattern naturally aligns with atomic transaction boundaries.

Variant	Total
CODEDELEGATOR (full)	38.4
w/o EPSS	-4.7
w/o Role Separation	-10.5

Table 5: Ablation study on MCPMark.

6 Analysis

Ablation Study. We conduct an ablation study on MCPMark to validate our components (Table 5).

- **w/o Role Separation.** Reverts to a single persistent agent, removing ephemeral Coders entirely. The substantial 10.5% drop confirms that spawning fresh agents per sub-task, rather than accumulating execution history in a shared context, is critical to performance.
- **w/o EPSS.** Retains the two-agent architecture but replaces EPSS's structured schemas with natural-language message passing. The 4.7% drop confirms that typed communication protocols independently reduce coordination noise beyond what role separation alone provides.

Analysis on Complexity Scaling. Fig. 6 shows performance by difficulty on τ^2-bench airline (DeepSeekV3.2). On Low-complexity tasks, both methods perform comparably; however, CODEDELEGATOR's advantage emerges with increasing complexity: it outperforms CodeAct by 11.2% (18.7% relatively) on Medium and 3.9% (14.5% relatively) on High tasks. CodeAct degrades sharply with complexity while CODEDELEGATOR exhibits more graceful degradation, validating that role separation becomes increasingly valuable as task complexity grows.

Quantitative Analysis of Context Pollution Reduction. We analyze τ^2-bench (retail, DeepSeekV3.2) logs along three dimensions.

- **Context Growth Rate.** Fig. 7 plots the average prompt tokens per LLM call at each turn. ReAct and CodeAct exhibit monotonically increasing context, reaching 8.4K and 15.0K tokens respectively, while the Delegator's context grows more slowly (peak 10.3K), confirming EPSS shields the planner from trace accumulation.

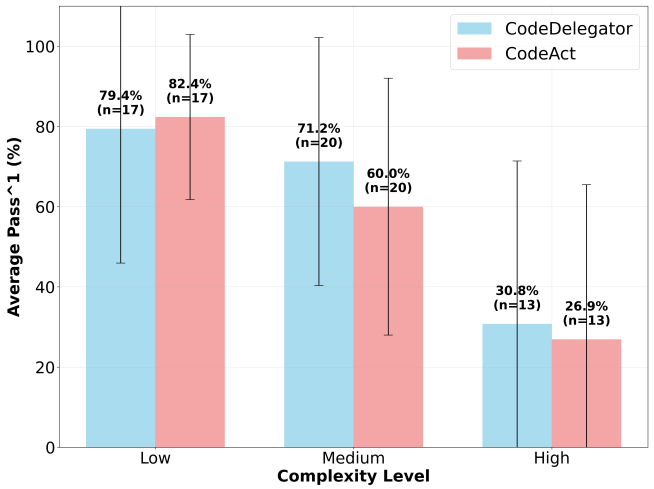

Figure 6: pass^1 accuracy by task complexity.

Method	pass^1 (%)
ReAct	73.2
CodeAct	76.7
AutoGen	59.6
CODEDELEGATOR	**82.0**

Table 6: Comparison with AutoGen on τ^2-bench retail (DeepSeekV3.2, 114 tasks). AutoGen's general multi-agent orchestration without targeted context isolation yields lower success rate than even single-agent baselines, confirming that context pollution requires mechanism-specific solutions.

Metric	ReAct	CodeAct	CODEDELEGATOR
Success Rate (%)	73.2	76.7	**82.0**
Avg Max Prompt Tok	8,436	14,994	10,266
Avg Total Prompt Tok	83,606	90,176	**79,364**
Coder Fresh Context	—	—	6,777

Table 7: Efficiency comparison on τ^2-bench retail (DeepSeekV3.2). "Coder Fresh Context" is the average prompt tokens at Coder initialization.

- **Tool Execution Trace Ratio.** Fig. 8 shows tool traces constitute 25.5% of CodeAct's context and 21.7% of ReAct's, but only **9.6%** of the Delegator's—a 55% relative reduction. Traces are confined to ephemeral Coders (11.1%), validating EPSS's asymmetric filtering.
- **Per-Coder Context and Efficiency.** Each Coder starts with a median context of 6,616 tokens, well below single-agent baselines. Despite running multiple agents, total prompt tokens per task (79.4K, Table 7) are lower than ReAct (83.6K) and CodeAct (90.2K), yielding the best cost-per-solved-task with the highest success rate (82.0%).

7 Conclusion

We presented CODEDELEGATOR, a multi-agent framework that mitigates context pollution in code-as-action agents via role separation: a persistent Delegator plans while ephemeral Coders execute sub-tasks in isolated contexts. Ephemeral-Persistent State Separation (EPSS) ensures only validated results propagate upward. Experiments on τ^2-bench and MCP-Mark show consistent improvements over CodeAct, especially on hard tasks, and confirm that EPSS reduces tool-trace contamination by over 55% relative to shared-context baselines while lowering total token consumption.

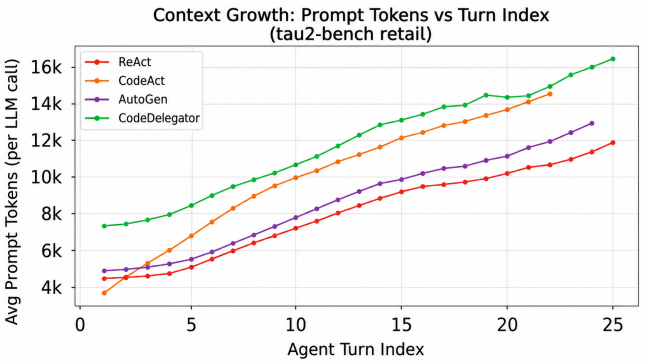

Figure 7: Average prompt tokens per LLM call at each turn index (τ^2-bench retail). CODEDELEGATOR's planner context grows more slowly than shared-context baselines.

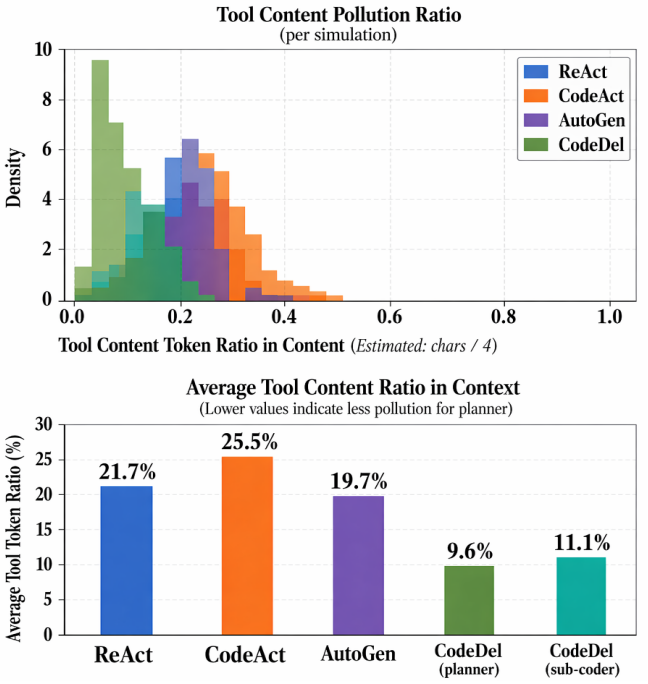

Figure 8: Average tool-execution trace ratio in agent context (estimated). CodeDel denotes CODEDELEGATOR. Its planner context contains only 9.6% tool traces vs. 25.5% for CodeAct, demonstrating effective trace isolation via EPSS.

Contribution Statement

Tianxiang Fei, Cheng Chen, and Yue Pan contributed equally to this work.

References

[Anthropic, 2025] Anthropic. Claude 3.7 sonnet. https://www.anthropic.com/news/claude-3-7-sonnet, 2025.

[Barres *et al.*, 2025] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025.

[Cemri *et al.*, 2025] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

[DeepSeek-AI *et al.*, 2025] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models, 2025.

[Hong *et al.*, 2023] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2023.

[Hu *et al.*, 2025] Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Bowei Xia, Tao Sun, Ziyu Ye, Zhaoxuan Jin, Yingru Li, Qiguang Chen, et al. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. *arXiv preprint arXiv:2505.23885*, 2025.

[Huang *et al.*, 2022] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[Li *et al.*, 2023] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

[Liu *et al.*, 2024] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[Liu *et al.*, 2025a] Anthony Zhe Liu, Xinhe Wang, Jacob Sansom, Yao Fu, Jongwook Choi, Sungryull Sohn, Jaekyeom Kim, and Honglak Lee. Interactive and expressive code-augmented planning with large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 20330–20354, 2025.

[Liu *et al.*, 2025b] Bang Liu, Xinfeng Li, Jiayi Zhang, Jinlin Wang, Tanjin He, Sirui Hong, Hongzhang Liu, Shaokun Zhang, Kaitao Song, Kunlun Zhu, et al. Advances and

challenges in foundation agents: From brain-inspired intelligence to evolutionary, collaborative, and safe systems. *arXiv preprint arXiv:2504.01990*, 2025.

[OpenAI, 2025a] OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, 2025.

[OpenAI, 2025b] OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, 2025.

[OpenAI, 2025c] OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2025.

[Qu *et al.*, 2025] Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343, 2025.

[Wang *et al.*, 2024a] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

[Wang *et al.*, 2024b] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.

[Wang *et al.*, 2025] Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The openhands software agent sdk: A composable and extensible foundation for production agents, 2025.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[Wu *et al.*, 2024] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

[Wu *et al.*, 2025] Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, Yaoqi Ye, Jiawei Wang, et al. Mcpmark: A benchmark for stress-testing realistic and comprehensive mcp use. *arXiv preprint arXiv:2509.24002*, 2025.

[Yang *et al.*, 2023] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.

[Yang *et al.*, 2024] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[Yao *et al.*, 2023] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *11th International Conference on Learning Representations, ICLR 2023*, 2023.

[Yao *et al.*, 2024] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024.

[Yu *et al.*, 2025] Zhaoyang Yu, Jiayi Zhang, Huixue Su, Yufan Zhao, Yifan Wu, Mingyi Deng, Jinyu Xiang, Yizhang Lin, Lingxiao Tang, Yingchao Li, et al. Recode: Unify plan and action for universal granularity control. *arXiv preprint arXiv:2510.23564*, 2025.

[Zhang *et al.*, 2024] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024.