

# CaveAgent: Transforming LLMs into Stateful Runtime Operators

Maohao Ran<sup>1,9\*</sup>, Zhenglin Wan<sup>2\*</sup>, Cooper Lin<sup>3</sup>, Yanting Zhang<sup>1</sup>, Hongyu Xin<sup>1</sup>, Hongwei Fan<sup>6</sup>, Yibo Xu<sup>1</sup>, Beier Luo<sup>3</sup>, Yixin Zhou<sup>4</sup>, Wangbo Zhao<sup>2</sup>, Lijie Yang<sup>7</sup>, Lang Feng<sup>5</sup>, Fuchao Yang<sup>5</sup>, Jingxuan Wu<sup>8</sup>, Yiqiao Huang<sup>10</sup>, Chendong Ma<sup>9</sup>, Dailing Jiang<sup>9</sup>, Jianbo Deng<sup>1</sup>, Sihui Han<sup>1</sup>, Bo An<sup>5</sup>, Yike Guo<sup>1</sup>, Jun Song<sup>1,9†</sup>

<sup>1</sup>HKUST <sup>2</sup>NUS <sup>3</sup>HKU <sup>4</sup>CMU <sup>5</sup>NTU, Singapore

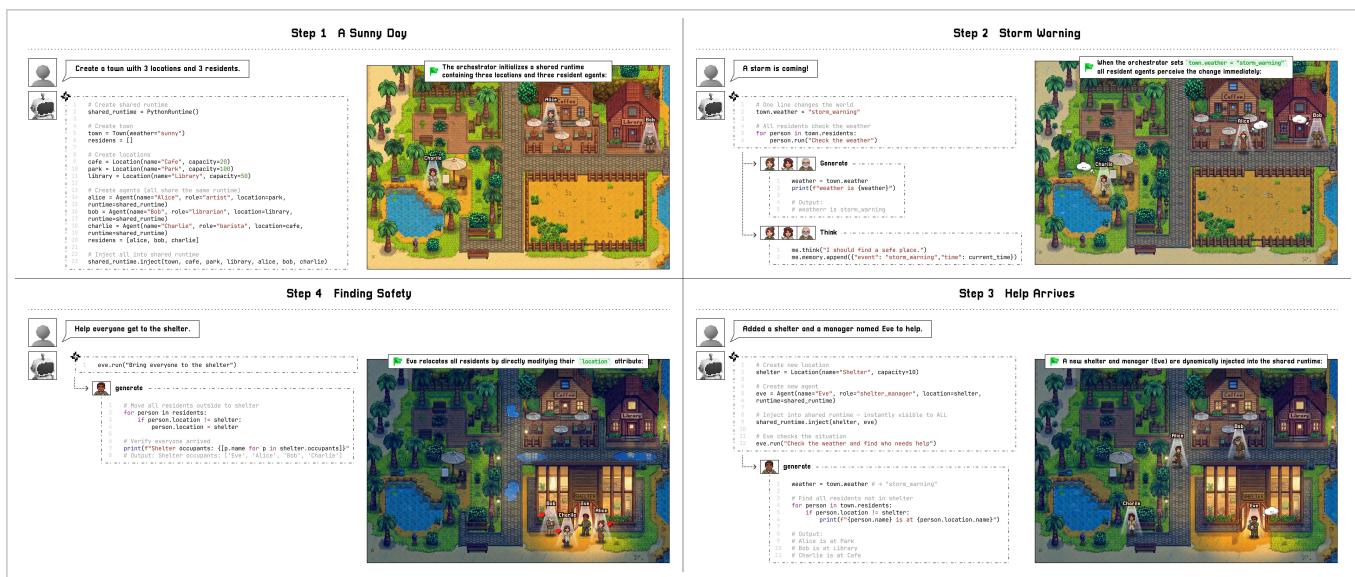
<sup>6</sup>Imperial College London <sup>7</sup>Princeton <sup>8</sup>UNC, Chapel hill <sup>9</sup>HKBU <sup>10</sup>Harvard

**Abstract:** LLM-based agents are increasingly capable of complex task execution, yet current agentic systems remain constrained by text-centric paradigms. Traditional approaches rely on procedural JSON-based function calling, which often struggles with long-horizon tasks due to fragile multi-turn dependencies and context drift. In this paper, we present CaveAgent, a framework that transforms the paradigm from "LLM-as-Text-Generator" to "LLM-as-Runtime-Operator." We introduce a Dual-stream Context Architecture that decouples state management into a lightweight semantic stream for reasoning and a persistent, deterministic Python Runtime stream for execution. In addition to leveraging code generation to efficiently resolve interdependent sub-tasks (e.g., loops, conditionals) in a single step, we introduce *Stateful Runtime Management* in CaveAgent. Distinct from existing code-based approaches that remain text-bound and lack the support for external object injection and retrieval, CaveAgent injects, manipulates, and retrieves complex Python objects (e.g., DataFrames, database connections) that persist across turns. This persistence mechanism acts as a high-fidelity external memory to eliminate context drift, avoid catastrophic forgetting, while ensuring that processed data flows losslessly to downstream applications. Comprehensive evaluations on Tau<sup>2</sup>-bench, BFCL and various case studies across representative SOTA LLMs demonstrate CaveAgent's superiority. Specifically, our framework achieves a 10.5% success rate improvement on retail tasks and reduces total token consumption by 28.4% in multi-turn scenarios. On data-intensive tasks, direct variable storage and retrieval reduces token consumption by 59%, allowing CaveAgent to handle large-scale data that causes context overflow failures in both JSON-based and CodeAct-style agents. Furthermore, the accessible runtime state provides programmatically verifiable feedback, establishing a rigorous foundation for future research in Reinforcement Learning (RL).

**Date:** Jan 5, 2026

**Code:** <https://github.com/acodercat/cave-agent>

**Main Contact:** Zhenglin Wan ([vanzl@u.nus.edu](mailto:vanzl@u.nus.edu)), Jun Song ([junsong@hkbu.edu](mailto:junsong@hkbu.edu))



**Figure 1:** Town Simulation: a toy example for Stateful Runtime-Mediated Multi-Agent Collaboration.

\* Joint first author & Equal Contribution.

† Corresponding to [junsong@hkbu.edu.hk](mailto:junsong@hkbu.edu.hk)

## 1. Introduction

Large Language Models (LLMs) have demonstrated remarkable general knowledge acquisition and human-like reasoning capabilities, exhibiting exceptional performance across diverse natural language processing tasks. Building upon these foundational capabilities, tool-integrated reasoning (TIR) has enhanced LLM agent through reasoning to interact with external tools or application programming interfaces (APIs) in a multi-turn manner<sup>1</sup> (Lu et al., 2023, Shen et al., 2023, Patil et al., 2024, Qu et al., 2025), thereby substantially expanding their information access and solution space. This largely amplifies the landscape of LLM agents to variety of domains, such as scientific discovery (Boiko et al., 2023, Bran et al., 2023), mathematical problem-solving (Gao et al., 2023, Chen et al., 2022), Web GUI navigation (Zhou et al., 2023, Yao et al., 2022a) and Robotics (Driess et al., 2023, Zitkovich et al., 2023).

Despite the promising landscape, the conventional protocol for tool use requires LLMs to conform to predefined JSON schemas and generate structured JSON objects containing precise tool names and arguments (Qin et al., 2023, Achiam et al., 2023). For example, to retrieve stock data, the model must strictly synthesize a JSON string like `{"tool": "get_stock", "params": {"ticker": "AAPL", "date": "today"}}`, requiring exact adherence to syntax and field constraints. However, this approach exhibits significant limitations: **1) Flexibility:** Agents are typically constrained by a rigid, iterative loop: executing a single tool call (or a parallel batch), serializing the output, and feeding the result back into the context for the subsequent generation. This introduces significant latency and context redundancy, resulting in suboptimal performance when addressing complex tasks that demand the sophisticated orchestration of sequential tool interactions (Wu et al., 2024, Shen et al., 2023). **2) Hallucination:** Achieving reliable tool-calling capabilities necessitates that the LLM outputs tool-related tokens with zero-shot precision (Patil et al., 2024). However, in practice, relying on in-context learning to guide tool generation often suffers from severe hallucinations, such as inventing non-existent parameters or violating type constraints (Patil et al., 2024, Qin et al., 2023, Li et al., 2023b). Crucially, errors in early turns propagate through the conversation, leading to cascading failures in multi-turn tasks (Kim et al., 2023). Moreover, adhering to JSON-Schema needs post-training LLMs which requires significant time and computational resources, and may even result in an LLM with lower intelligence levels than before post-training (He, 2024).

While recent works attempt to address this issue by empowering LLMs with code-based tool-use (Wang et al., 2024, Yang et al., 2024), they predominantly adopt a *process-oriented* paradigm where the runtime state remains **internalized and text-bound**. The interaction forces a "textualization bottleneck": variables are accessible to external systems only through text output, requiring serializing into text strings (e.g., printing a DataFrame) to communicate with the user (Wang et al., 2024, Yao et al., 2022b). This limitation fundamentally prohibits the direct input and output of structured, manipulatable objects, making it inefficient or impossible to handle complex non-textual data (e.g., large datasets, videos) (Qiao et al., 2023) and interact with down-stream tasks. To address these limitations:

*We aim to build a system that utilizes Python's "everything is an object" philosophy to enable full Object-Oriented function calling and interaction, delegating context engineering to a persistent runtime and allowing the direct injection and retrieval of high-fidelity objects without serialization loss, thereby fully leveraging the strong code-generation capability of LLMs.*

We present **CaveAgent**, an open-source platform that pioneers the concept of **Stateful Runtime Management** in LLM agents. This marks a shift of code-based function calling paradigm from "process-oriented function-calling" to persistent "object-oriented state manipulation". CaveAgent operates on a **dual-stream architecture** that enhances interaction between LLM agents and environments through two distinct streams: a **semantic stream** for reasoning and a **runtime stream** for state management and code execution. In this framework, the semantic stream remains lightweight, receiving only abstract descriptions of functions' API and variables. It leverages the LLM's inherent coding capabilities to generate code that manipulates the **runtime stream**—the primary locus of our stateful management. By injecting complex data structures (such as graph, dataframe, etc.) directly into the runtime as persistent objects, we achieve another form of **context engineering**: the agent manipulates high-fidelity data via concise variable references, decoupling storage from the limited context window to the persistent runtime. Specifically, any intermediate result (e.g., DataFrames, planning trees, or key metadata) can be stored in newly injected stateful variables and the agent will actively retrieve *relevant* variables for later use or downstream applications. This avoids catastrophic forgetting (Ouyang et al., 2022, Luo et al., 2025), enables efficient context compression and error-free recall for long-term memory where the runtime serves as an "external memory dictionary". Besides, this persistent environment further enables **few-steps solution of complex logical dependencies** by directly using code to interact with multiple logically inter-dependent tools, allowing the agent to compose intricate workflows (e.g., data filtering followed by analysis) in a few turns, thus avoiding the potential error and instability caused by multi-round function calling (Wang et al., 2023a, Qin et al., 2023). Furthermore, the runtime's transparency makes agent behavior fully verifiable, supporting checks on both intermediate programmatic states and final output objects of any data type. This capability creates

<sup>1</sup>In this paper, we refer to tool-use and function-calling interchangeably.

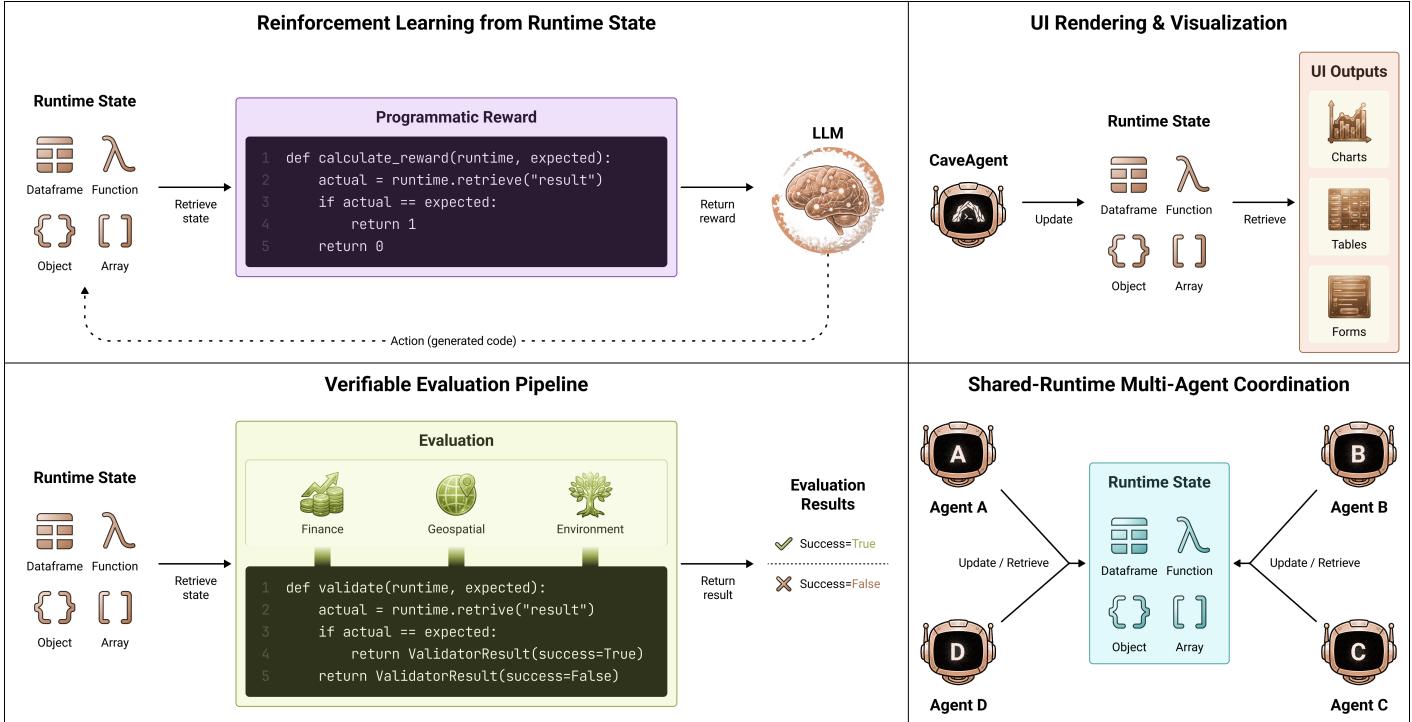


Figure 2: Key Advantages of CaveAgent

a rigorous framework for Reinforcement Learning by enabling the generation of verifiable, fine-grained reward signals. Finally, CaveAgent supports **lossless artifact handoff** by returning **native Python objects** rather than text representations, and the extraction of manipulated Python objects for direct use in down-stream tasks such as UI rendering, visualization and structured validation. The runtime can be easily serialized and reloaded, providing a simple yet powerful mechanism for preserving the agent's complete state across sessions and enabling true long-term memory and task continuity. This transforms the LLM from isolated text generator into an interoperable computational entity, seamlessly embedding within complex software ecosystems and automated decision-making frameworks.

In addition to these insights, we found that the function calling paradigm in CaveAgent could potentially extends beyond single-agent capabilities to pioneer **Runtime-Mediated Multi-Agent Coordination**, as shown in Figure 1 and the right-bottom sub-figure of Figure 2. Unlike conventional frameworks where agents coordinate via lossy, high-latency text message passing (Li et al., 2023a, Park et al., 2023), CaveAgent enables agents to interact through direct state manipulation. In this paradigm, a supervisor agent can programmatically inject variables into a sub-agent's runtime to dynamically alter its environment or task context, effectively controlling behavior without ambiguous natural language instructions. Furthermore, multiple agents can operate on a unified shared runtime, achieving implicit synchronization: when one agent modifies a shared object (e.g., updating a global 'weather' entity in a town simulation), the change is instantly perceivable by all peers through direct reference. This transforms multi-agent collaboration from a complex web of serialized dialogue into a precise, verifiable state flow, ensuring that large-scale coordination remains coherent and grounded (the details of Runtime-Mediated Multi-Agent Coordination can be found in Appendix E). We summarize our contribution as follows:

- We introduce **CaveAgent**, a new function-calling paradigm that pioneers the concept of **Stateful Runtime Management**. This architecture marks a paradigm shift from process-oriented function calling to persistent, object-oriented state management. CaveAgent achieves a form of **context compression** and **context-grounded memory recall** via delegating context engineering to **persistent** runtime, eliminating the token overhead and precision loss inherent in textual serialization while enabling the few step solution of complex, logically interdependent tasks.
- The framework's programmatic inspectability provides deterministic feedback on intermediate states, establishing a rigorous foundation for future research in **Reinforcement Learning with Verifiable Rewards (RLVR)** on this paradigm without the need for subjective human annotation.
- We conduct evaluations demonstrating CaveAgent's tool use ability on standard benchmarks (e.g., Tau<sup>2</sup>Bench) and provides comprehensive case study across various domains to showcase the unique advantages of CaveAgent. Additionally, we identified the potential to extend the paradigm to enable **Stateful Runtime-Mediated Multi-Agent Coordination** and provided qualitative results, opening the opportunity for future research on this direction.

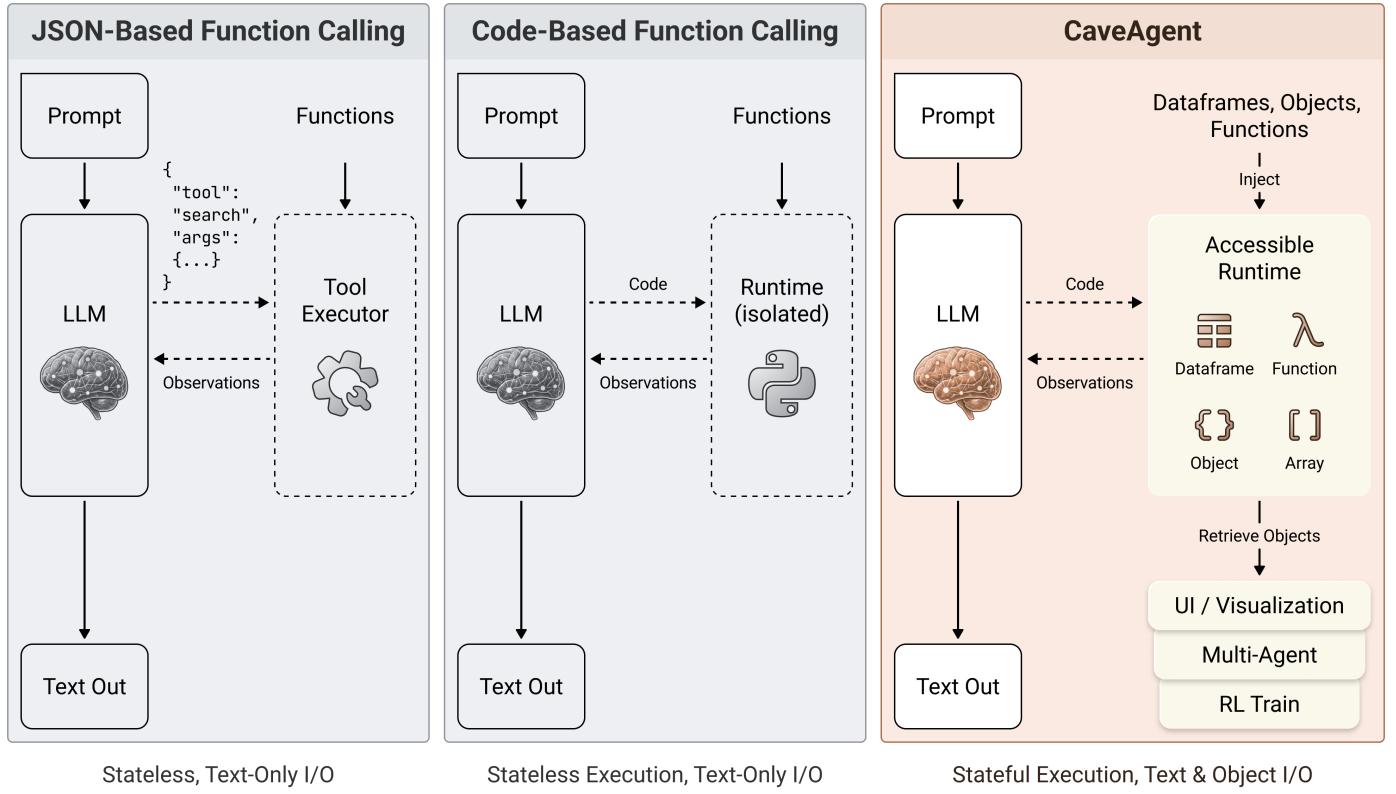


Figure 3: Evolution of Agentic Tool Use

## 2. Background

In this section, we formally formulate the function-calling (tool-use) in LLMs. We consider an LLM agent  $\mathcal{M}_\theta$  parameterized by  $\theta$ , tasked with solving a user query  $q$ . The agent is equipped with a tool library  $\mathcal{T} = \{t_1, t_2, \dots, t_N\}$ . Each tool  $t_i$  is defined by a tuple  $(n_i, d_i, \rho_i)$ , representing the tool name, description, and parameter space, respectively. The problem is modeled as a multi-step decision process. At time step  $k$ , given the context history  $H_k$ , the agent generates a response. The history is defined as a sequence of interactions:

$$H_k = \{q, (r_1, a_1, o_1), \dots, (r_{k-1}, a_{k-1}, o_{k-1})\} \quad (1)$$

where  $r_k$  denotes the internal reasoning (thought),  $a_k$  denotes the tool action, and  $o_k$  denotes the execution observation.

**ReAct Paradigm** In the traditional ReAct paradigm (Yao et al., 2022b), both reasoning  $r_k$  and action  $a_k$  are generated as contiguous natural language sequences from the model's vocabulary  $\mathcal{V}$ . The generation probability is formalized as:  $P(r_k, a_k | H_k) = \prod_{j=1}^L P_\theta(w_j | H_k, w_{<j})$ , where  $w$  represents tokens in the sequence  $(r_k, a_k)$ . Crucially, the action  $a_k$  in ReAct is a raw text string (e.g., "Action: Search[query]") that requires a heuristic parser  $\mathcal{P}$  to extract the executable function and parameters. Let  $\hat{a}_k$  be the raw text output. The effective action is obtained via:  $a_k = \mathcal{P}(\hat{a}_k)$ , s.t.  $\hat{a}_k \in \mathcal{V}^*$ . This formulation suffers from *hallucination* and *format errors*, as the support of  $P_\theta$  covers the entire vocabulary  $\mathcal{V}$ , meaning  $\sum_{w \notin \text{valid\_syntax}} P_\theta(w | \dots) > 0$  (Patil et al., 2024, Liu et al., 2023).

**JSON-Schema Function Calling** To address the ambiguity of unstructured text generation, modern agents adopt *JSON-Schema Function Calling* (Patil et al., 2024, Qin et al., 2023). Here, the toolset  $\mathcal{T}$  is augmented with a set of structured schemas  $\mathcal{S} = \{S_1, \dots, S_N\}$ , typically defined in JSON Schema format. The reasoning process remains in natural language, but the action generation is transformed into a constrained decoding process. The model is conditioned explicitly on  $\mathcal{S}$ , and the action  $a_k$  is no longer treated as free text, but as a structured object  $j_k$  (a JSON object). To help the model output the JSON object correctly, we could utilize In-Context Learning to internalize schema structures (Schick et al., 2023). Crucially, special tokens (e.g., <tool>) are introduced to explicitly demarcate the reasoning phase from the action phase. While

recent development in Agentic RL adopts Reinforcement Learning using composite reward signal to incentive the model to output correct JSON format and function calling parameter (Le et al., 2022), this paper mainly focus on inference-time rather than training-time techniques.

Essentially, JSON-Schema function calling operates as a **text-based serialization loop** (Yao et al., 2022b, Mialon et al., 2023). The process consists of three phases: (1) **Context Serialization**: The structured schema  $\mathcal{S}$  is flattened into a textual description and injected into the system prompt via context engineering; (2) **String Generation**: The LLM acts as a neural generator, predicting a string JSON payload based on textual instructions; and (3) **Execution**: An external middleware parses this string, executes the actual code, and serializes the execution result back into text to update the context window. This paradigm does not fundamentally deviate from the traditional context engineering framework of LLMs, suffering from inherent limitations such as context explosion, hallucination, and error propagation (Packer et al., 2023, Wang et al., 2023b).

**Code-based Function Calling** To address these limitations, recent works such as *codeact* (Wang et al., 2024) utilize executable code as the media of function calling. However, current code agents suffer from architectural limitations. *CodeAct* essentially does not expose explicit APIs for external object injection and retrieval. Interaction is strictly mediated by the LLM via a "textualization" bottleneck, where intermediate states must be serialized into standard output (e.g., `print`) to be perceived by the user. For instance, as Wang et al. (2024) stated, when the agent requires external data for analysis, *Codeact* typically downloads the dataset via Python (e.g., `pd.read_csv(url)`). This approach is inherently inflexible: its interface boundary relies on text serialization for data ingestion, making it difficult to directly inject pre-existing Python objects such as in-memory DataFrames, trained models, or custom class instances without custom workarounds. Moreover, this reliance on text makes it challenging to handle of non-textual or high-dimensionality artifacts (e.g., raw video streams, large-scale databases), and exposes the risk of context explosion, distraction of LLM and hallucination (Liu et al., 2024, Packer et al., 2023).

In the subsequent section, we demonstrate how CaveAgent pioneers the **Object-Oriented** paradigm based on Python's "everything is an object" philosophy by maintaining two paralleled context stream and delegating context management to a **persistent** Python runtime stream. Figure 3 shows the evolution path of paradigm shift in Agentic Tool Use.

### 3. CaveAgent: Stateful Runtime Management

#### 3.1. Core Methodologies

In this section, we present the design philosophy of CaveAgent. As illustrated in Figure 4, CaveAgent adopts a **dual-stream architecture**, maintaining two synchronized streams throughout the interaction lifecycle: a *Semantic Stream* for light-weight reasoning, and a *Runtime Stream* for stateful execution, observation and context engineering. This design fundamentally redefines the agentic interaction loop, shifting from stateless text-based serialization to a persistent, state-aware model.

We model the agent's task as a sequential decision process over a horizon  $T$ . At each turn  $t \in [1, T]$ , the agent receives a query or observation  $x_t$  and must produce a response  $y_t$ . Unlike traditional formulations where the entire state is re-serialized into  $x_t$ , we introduce a latent runtime state  $\mathcal{S}_t$  (we call it "in-runtime context"). The system evolution is thus defined by:

$$h_t = \text{LLM}(x_t, h_{t-1}) \quad (\text{Semantic Stream: Context History}) \quad (2)$$

$$\mathcal{S}_t = \text{Exec}(c_t, \mathcal{S}_{t-1}) \quad (\text{Runtime Stream: Persistent Environment}) \quad (3)$$

where  $h_t$  represents the semantic history (we call it "in-prompt context") and  $c_t$  is the executable code generated by the agent. The critical innovation lies in the decoupling of  $h_t$  and  $\mathcal{S}_t$ : the semantic stream tracks *intent* and light-weight *reasoning* for code generation, while the runtime stream maintains all crucial *data* and *execution state* via the code generated by semantic stream.

**The Runtime Stream:** The core engine of the runtime stream is a persistent Python kernel (specifically, an IPython interactive shell). We conceptualize each interaction turn  $t$  not as an isolated API call, but as a **cell execution** in a virtual Jupyter notebook.

- **Persistent Namespace:** The state  $\mathcal{S}_t$  comprises the global namespace  $\mathcal{N}_t$ , containing all variables, functions, and imported modules. When the agent executes code  $c_t$  (e.g., `x = 5`), the modification to  $\mathcal{N}_t$  persists to  $\mathcal{N}_{t+1}$ . This allows subsequent turns to reference `x` directly without requiring the LLM to memorize or re-output its value.

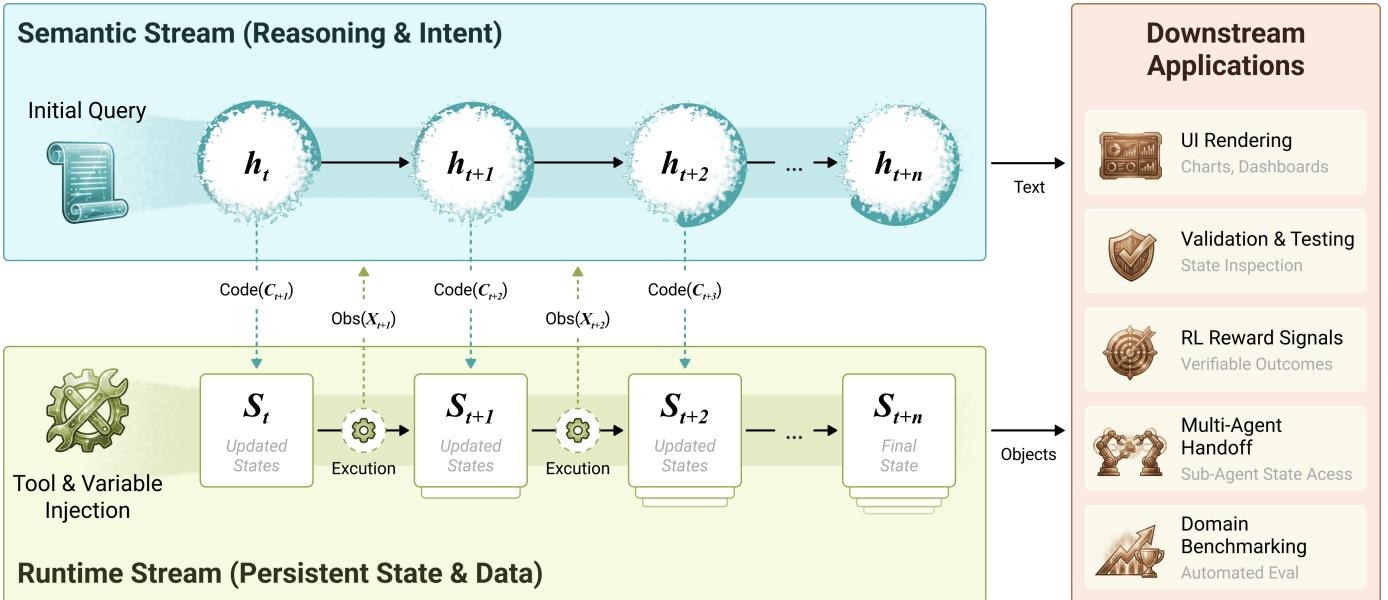


Figure 4: Framework Overview

- **Stateful Injection:** Tools are not only described in text; they are *injected* into  $\mathcal{N}_0$  as live Python objects. This allows the agent to interact with stateful objects via calling tools that modify the object’s internal state across turns.

Notably, the runtime stream can assign values to new variables during the interaction process and inject them into the Persistent Namespace (in-runtime context). This enables heavy context in complex tasks, such as large DataFrames, graphs, or other intricate data structures, to be managed entirely by the Python runtime stream as stateful variables. Their values are thus preserved natively in persistent runtime memory without requiring repeated serialization into text, effectively eliminating the risk of hallucination that arises from lossy textual representations. Besides, the agent can inject and store crucial information (such as key reasoning chains and intermediate data analysis results) via new persistent variables into in-runtime context, retaining only a lightweight description and reference in its in-prompt context. Consequently, the runtime functions as an external memory dictionary, allowing the agent to actively retrieve this memory as native, lossless Python objects, thus achieving a form of context compression and avoiding catastrophic forgetting. This property is crucial, as it addresses persistent challenges in agentic tool use—specifically memory, dynamic decision-making, and long-horizon reasoning (Patil et al.). Meanwhile, this system also makes the manipulation of data objects and propagating them between multiple-turns much easier, no matter how complex the data structures are.

It is also notable that the programmatic state retrieval enables the extraction of manipulated Python objects for direct use in downstream applications. Unlike conventional agents that produce terminal text outputs requiring parsing and reconstruction, CaveAgent exposes native objects (DataFrames, class instances, arrays) with full type fidelity and structural integrity. This enables diverse integration pathways. For example:

- UI components can bind directly to retrieved objects for exact data visualization, enabling real-time dashboards reflect to exact agent state.
- RL pipelines can compute precise reward signals through programmatic state inspection rather than noisy text-based heuristics, automate the process of success/failure detection for trajectory labeling, and conduct credit assignment based on state analysis.
- Validation frameworks can apply unit test assertions and schema verification against returned structures, enabling domain-specific benchmarking with programmatic correctness.
- Multi-agent systems can pass objects directly between agents without serialization loss, share state synchronization across agent swarm and resolve dependency based on object availability (as discussed in detail at Appendix E).

The agent thus transforms LLMs from an isolated text generator into the operator of a stateful, interoperable computational component whose outputs integrate natively into broader software ecosystems and automated decision-making pipelines.

**The Semantic Stream:** Parallel to the runtime stream, the semantic stream utilizes LLM as a brain to generate code to manipulate runtime. Besides, it is also responsible for:

- **Prompt Construction:** Dynamically generating system instructions that describe the *signatures* of available tools in  $\mathcal{N}_t$ , without dumping their full state (which may be large) into the in-prompt context window.
- **Observation Shaping:** Captures execution outputs and enforces a length constraint  $\tau(\cdot)$  to prevent context explosion. This feedback mechanism actively teaches the agent to interact with the persistent state efficiently, prioritizing concise and crucial information over verbose raw dumps in the in-prompt context  $h_{t+1}$ .

This dual-stream design solves the "Context Explosion" problem: massive data remains in the Runtime Stream ( $\mathcal{S}_t$ ), while only the high-level reasoning and necessary summaries flow through the Semantic Stream ( $h_t$ ). The LLM effectively operates a remote control (code) to manipulate a complex runtime without needing to hold the runtime's entire state in its working memory. Compared to traditional JSON-based function calling where larger models tend to parallelize tool calls for efficiency but fall short when there are inter-dependencies between tools (Lu et al., 2025), CaveAgent enables dependency-aware parallelism, allowing agents to dispatch complex, interdependent tool chains in a few turns via executable code to guarantee both efficiency and correctness. Compared to traditional code-based function calling that adopts internalized runtime, CaveAgent opens the runtime as a bidirectional interface, allowing developers to **inject** arbitrary variables directly and **retrieve** structured, manipulatable objects of any type at any time and achieves true stateful interoperability. Algorithm 1 in Appendix A showcases the iteration loop of CaveAgent framework. Then, we demonstrate three core designs of CaveAgent beyond traditional JSON-based and code-based tool use.

### 3.1.1. Variable and Function Injection

To bridge the gap between large language models and executable environments, we introduce a unified abstraction for **Variable** and **Function** injection. CaveAgent treats Python objects and functions as first-class citizens within the runtime environment to ensure object-oriented interactions. This mechanism consists of two key components: metadata extraction for the semantic stream's context and direct object injection into the runtime namespace.

**Descriptive Abstraction** Each injectable entity is wrapped in a container that automatically extracts its metadata. For functions, this includes the signature, type hints, and docstrings; for variables, it includes the name, type, and an optional description. Formally, a function  $f$  is represented as a tuple  $(n_f, \sigma_f, d_f)$ , where  $n_f$  is the function name,  $\sigma_f$  is the signature derived from inspection, and  $d_f$  is the documentation. Similarly, a variable  $v$  is represented as  $(n_v, \tau_v, d_v)$ , where  $\tau_v$  is the type. This metadata is aggregated and injected into the system prompt, providing the model with a clear "API reference" of available capabilities without exposing implementation details or raw values. For example, an injected data processing object might be presented to the model as:

```
name: processor
type: DataProcessor
description: A utility for sorting and filtering data collections.
methods: process(data: list) -> list
```

**Namespace Injection** Critically, injection goes beyond mere description. Upon initialization, the runtime maps these entities directly into the namespace of the underlying execution engine (e.g., IPython). This means that if a function add or an object `processor` is injected, they become immediately available as global symbols in the execution environment. This design enables *Object-Oriented Interaction* and *Stateful management*. Instead of stateless function calls (e.g., `tool: "sort", args: {data: ...}`), the model can invoke methods on stateful objects directly (e.g., `processor.process(data)`). This significantly enhances composability, as the model can chain method calls and manipulate object attributes naturally, mirroring standard programming practices rather than rigid API request-response cycles.

After variable and function injection, CaveAgent interacts with the environment via executable Python programs leveraging native Python syntax for robust parsing and utilizing control flow (loops, conditionals) with stateful data passing to handle multi-step logic. Unlike text-based paradigms, CaveAgent allows for lossless manipulation of complex data structures throughout the interaction. Consequently, the agent delivers the final output not as a textual approximation, but as a valid, native Python object guaranteed to match the expected type, enabling seamless integration with downstream applications.

### 3.1.2. Dynamic Context Synchronization

While the dual-stream architecture decouples reasoning from state storage, effective collaboration requires a regulated information flow between the Semantic Stream and the Runtime Stream. We implement a dynamic synchronization mechanism to ensure the agent remains aware of the runtime state without overwhelming its context window.

In our framework, the Semantic Stream is "blind" to the Runtime Stream by default. Visibility is achieved explicitly via execution outputs. To inspect the state  $\mathcal{S}_t$  (e.g., the content of a variable), the agent must generate code to print a summary (e.g., `print(df.head())`). This design enforces an **Active Attention Mechanism**: the agent consciously selects which part of the massive runtime state is relevant to the current reasoning step, pulling only that slice from the runtime, the external memory, into the token context.

To prevent "Context Explosion" caused by accidental verbose outputs (e.g., printing a million-row list), we introduce an **Observation Shaping** layer. The runtime captures standard output and subjects it to a length constraint function  $\tau(\cdot)$ .

$$o_t = \tau(s_t) = \begin{cases} \text{stdout}_t & \text{if } |\text{stdout}_t| \leq L_{\max} \\ \text{Error}_{\text{size}}(\text{len}) & \text{if } |\text{stdout}_t| > L_{\max} \end{cases} \quad (4)$$

When the output exceeds  $L_{\max}$ , instead of truncating silently, the system injects a specific meta-instruction prompting the agent to revise its code (e.g., "Output exceeded limit, please use summary methods"). This feedback loop teaches the agent to interact with the persistent state efficiently, favoring information which is concise and most-relevant, over verbose raw data dumps.

### 3.1.3. Security Check via Static Analysis

CaveAgent mitigates code execution risks via Abstract Syntax Tree (AST)-based static analysis, enforcing security policies without compromising flexibility. We parse code  $c_t$  into a tree  $\mathcal{T}_{c_t}$  and validate it against a policy  $\Pi = \{r_1, \dots, r_k\}$ , where  $\text{Valid}(c_t) \iff \forall r \in \Pi, r(\mathcal{T}_{c_t}) = \emptyset$ . The modular rule set includes (example):

- **ImportRule:** Blocks unauthorized modules (e.g., `os`, `subprocess`).
- **FunctionRule:** Prohibits dangerous calls (e.g., `eval()`, `exec()`).
- **AttributeRule:** Prevents sandbox bypass via internals (e.g., `__builtins__`).

**Structured Error Feedback.** Violations trigger structured observations rather than system crashes. For instance, a `SecurityError` is returned to the semantic stream, enabling the agent to self-correct (e.g., replacing `eval()` with safe tools) and ensuring interaction continuity.

## 4. Experiments

In this section, validate CaveAgent by answering four questions:

- [Q1.] Can CaveAgent perform on par with or surpass standard function-calling paradigms on widely-used benchmarks involving **basic** function-calling tasks? This is to showcase the basic function calling capabilities of CaveAgent.
- [Q2.] Can CaveAgent successfully perform state management across multi-turns correctly and efficiently?
- [Q3.] How token-efficient is CaveAgent compared to traditional JSON-based and Codeact style function calling?
- [Q4.] How does CaveAgent adapt to complex scenarios that require manipulating complex data objects? This is to showcase CaveAgent's unique advantages.

### 4.1. [Q1] Standard Function Calling Benchmarks

To verify the CaveAgent's basic function-calling capabilities on standard function-calling tasks, we employ two widely-adopted benchmarks in Agentic tool use: Tau<sup>2</sup>-bench (Yao et al., 2024) and the Berkeley Function Calling Leaderboard (BFCL) (Patil et al.).

**Models.** We evaluate a wide spectrum of State-of-the-Art (SOTA) LLMs to benchmark our performance, ensuring a comprehensive coverage of different architectures (e.g., dense vs. MoE) and model scales. The model suite includes:

- **DeepSeek-V3.2:** The latest iteration of the DeepSeek series, featuring a Mixture-of-Experts (MoE) architecture with 685B parameters (37B active). *Setting: Temperature set to 0.2 to ensure stable code generation.*
- **Qwen3 Coder 30B:** A specialized code-centric model built on the Qwen3 architecture. It utilizes a highly efficient MoE design with 30B parameters (3B active). *Setting: Configured with a temperature of 0.2 for stable output generation.*
- **Kimi K2 0905:** A large-scale MoE model with 1000B parameters (32B active), designed for long-context interactions. *Setting: We adopt the official recommended temperature of 0.6.*
- **Claude Sonnet 4.5:** The SOTA model of Claude-series from Anthropic. *Setting: Temperature is set to 0.2 for stable code generation.*
- **GPT-5.1:** An evolution of the GPT-series. *Setting: We utilize the default temperature of 1.0, as this is the only supported value for the current snapshot.*
- **Gemini 3 Pro:** Known for its massive context window and native multimodal reasoning. *Setting: Configured with "Low thinking" reasoning mode and a temperature of 1.0, adhering to official recommendations.*

For each backbone model, we conduct a comparative analysis between its native function-calling mechanism and our proposed CaveAgent framework. Crucially, within the CaveAgent workflow, the LLM is repurposed solely as a text generation engine, referred to as the *semantic stream* in our framework, bypassing its internal function-calling modules. We run each model using the standard API offered by the model provider.

#### 4.1.1. Results on Tau<sup>2</sup>-bench

Tau<sup>2</sup>-bench (Yao et al., 2024) is a comprehensive benchmark designed to evaluate the dynamic tool-use capabilities of LLM-based agents in realistic, multi-turn conversational scenarios. Unlike static evaluation sets that focus on single-turn intent detection, Tau<sup>2</sup>-bench necessitates that the agent interacts with a simulated user to achieve complex goals (e.g., modifying a flight reservation or processing a retail refund) while maintaining consistency across multiple turns. Following the original Tau<sup>2</sup>-bench paper, we focus on two primary domains within the benchmark: *Airline* and *Retail*. These domains are challenging since they require the agent to accurately track user constraints, database states, and policy regulations throughout the dialogue history.

**Experimental Setup.** To ensure a rigorous comparison, we strictly follow the evaluation protocols of Tau<sup>2</sup>-bench. Specifically, we utilize **DeepSeek V3** as the user simulator for all experiments to generate diverse and coherent user responses. To mitigate variance in generation, each model is tested three times for each domain. The reported results represent the respective scores and average scores of these three independent runs.

**Evaluation of CaveAgent.** Since CaveAgent executes Python code rather than JSON tool calls, we employ runtime instrumentation to capture function invocations. Wrapper functions intercept each function call, recording function names and arguments before delegating to the underlying implementation. The captured invocation sequence is compared against ground-truth actions using identical evaluation criteria applied to JSON-based agents, ensuring a fair cross-paradigm comparison based on *which* functions were called with *what* arguments.

**Performance Analysis.** The quantitative results on Tau<sup>2</sup>-bench are summarized in Table 1. The key insights include: (1). CaveAgent consistently outperforms the standard JSON-based function calling paradigm across 11 out of 12 experimental settings, covering both open-source and proprietary models ranging from 30B to over 1000B parameters. Significant improvements are observed in most SOTA models like **DeepSeek-V3.2** and **Gemini 3 Pro** (averaging +5.3% and +6.1% respectively), demonstrating that our framework breaks the performance ceiling of even the most capable semantic reasoners by offloading state management to a deterministic and error-free code runtime. (2). CaveAgent shows superiority in state-intensive scenarios. We observed that the performance advantage is markedly amplified in the *Retail* domain compared to *Airline*. Retail tasks in Tau<sup>2</sup>-bench typically involve complex transaction modifications and policy checks, which require maintaining high-fidelity state consistency across turns. The standard paradigm suffers from serialization overhead here, leading to hallucinations. In contrast, CaveAgent achieves double-digit gains in Retail for models like Qwen3 and Kimi K2. This validates our hypothesis that **Stateful Runtime Management** effectively eliminates errors caused by the repetitive text-based serialization of complex data objects (e.g., shopping carts or refund policies). We provided a detailed agent trajectory analysis about the reason of CaveAgent’s outstanding performance in Retail tasks in Appendix F.1. (3). CaveAgent unlocks the potential in code-centric models. Most notably, the smaller, code-specialized **Qwen3-Coder (30B)** exhibits the largest relative improvement (+13.5% in Retail), enabling it to rival the performance of significantly larger generic models. This confirms that CaveAgent **effectively leverages the inherent coding proficiency of LLMs**. By decoupling the *semantic*

**Table 1:** Performance comparison on the Tau<sup>2</sup> Benchmark across different models and domains (3 Runs). **Bold** indicates CaveAgent outperforms Function Calling. The number inside each cell represents the success rate of each task (%).

Model	Domain	Function Calling				CaveAgent			
		Run 1	Run 2	Run 3	Avg.↑	Run 1	Run 2	Run 3	Avg.↑
<b>Open Source</b>									
DeepSeek-V3.2 (685B)	Airline	56.0	56.0	54.0	55.3	62.0	60.0	58.0	<b>60.0 (+4.7)</b>
	Retail	79.8	77.2	74.6	77.2	85.1	82.5	78.1	<b>81.9 (+4.7)</b>
Qwen3-Coder (30B)	Airline	36.0	40.0	38.0	38.0	36.0	42.0	44.0	<b>40.7 (+2.7)</b>
	Retail	41.2	43.0	39.5	41.2	51.8	54.4	57.9	<b>54.7 (+13.5)</b>
Kimi-K2-0905 (1000B)	Airline	52.0	56.0	54.0	54.0	58.0	54.0	54.0	<b>55.3 (+1.3)</b>
	Retail	62.3	60.5	59.6	60.8	69.3	72.8	71.9	<b>71.3 (+10.5)</b>
<b>Closed Source</b>									
Claude Sonnet 4.5	Airline	56.0	54.0	62.0	57.3	56.0	52.0	62.0	56.7 (-0.6)
	Retail	68.4	67.5	81.6	72.5	73.7	75.4	80.7	<b>76.6 (+4.1)</b>
GPT-5.1	Airline	50.0	58.0	50.0	52.7	58.0	56.0	54.0	<b>56.0 (+3.3)</b>
	Retail	64.0	66.7	66.7	65.8	65.8	69.3	73.6	<b>69.6 (+3.8)</b>
Gemini 3 Pro	Airline	64.0	62.0	58.0	61.3	68.0	68.0	68.0	<b>68.0 (+6.7)</b>
	Retail	72.8	72.8	66.7	70.8	77.2	76.3	75.4	<b>76.3 (+5.5)</b>

stream from the *runtime stream*, our approach allows code-centric models to focus on logic generation rather than struggle with verbose context tracking, thereby maximizing the utility of limited parameters.

#### 4.1.2. Results on BFCL

While Tau<sup>2</sup>-bench evaluates the capability of maintaining long-term state, it is equally critical to assess the agent’s precision in atomic function executions. To this end, we employ the **Berkeley Function Calling Leaderboard (BFCL)** (Patil et al.), a widely recognized benchmark for quantifying the accuracy of LLM tool invocation.

**Benchmark Overview.** BFCL constructs a rigorous evaluation environment consisting of approximately 2,000 question-function-answer pairs derived from real-world use cases. The dataset is designed to test models across varying levels of complexity. Key evaluation categories include:

- **Simple Function:** Represents the fundamental evaluation scenario where the model is presented with a single function definition and must generate a unique invocation with correct arguments and results.
- **Multiple Function:** Assesses the model’s *selection capability*. The model is provided with a candidate set of 2 to 4 function definitions and must identify and execute the single most appropriate function that addresses the user’s query, filtering out irrelevant tools.
- **Parallel Function:** Evaluates the ability to execute concurrent/parallel actions within a single turn. The model must decompose a complex user query (spanning one or multiple sentences) into multiple distinct function calls, invoking them simultaneously to optimize efficiency.
- **Parallel Multiple Function:** The most challenging category, combining tool selection with parallel execution. The model is confronted with a larger pool of function definitions and must determine both the correct subset of tools to use and the frequency of their invocation (zero or more times) to fully resolve the request.

Notably, we use **Executable Evaluation (Functional Correctness)**, rather than Abstract Syntax Tree (AST) evaluation since AST is not directly applicable in our CaveAgent framework due to the lack of explicit JSON schema. We execute the generated code in a controlled environment and compares the execution result against the ground truth. This validation ensures that the function call triggers the correct behavior in real-world applications.

**Table 2:** Detailed Performance comparison on BFCL Benchmark (3 Runs). Data is presented in **score/total** format. The Avg. columns indicate the average overall percentage across 3 runs. **Bold** indicates CaveAgent outperforms Function Calling. Simp. means Simple Function, Mult. means Multiple Function, Para. means Parallel Function, P-M. means Parallel Multiple Function, Ov. means the overall performance across four function categories, and Avg. means the average performance across three runs.

Model	Run	Function Calling						CaveAgent					
		Simp.	Mult.	Para.	P-M.	Ov.	Avg.(%)	Simp.	Mult.	Para.	P-M.	Ov.	Avg.(%)
<b>Open Source</b>													
DeepSeek-V3.2 (685B)	R1	354/400	183/200	175/200	159/200	871/1000	86.9	382/400	192/200	185/200	178/200	937/1000	<b>94.0</b> (+7.1)
	R2	353/400	185/200	167/200	159/200	864/1000		386/400	193/200	184/200	178/200	941/1000	
	R3	360/400	185/200	173/200	154/200	872/1000		384/400	192/200	186/200	180/200	942/1000	
DeepSeek-V3.2 (w/o prompt)	R1	312/400	162/200	33/200	26/200	533/1000	53.1	382/400	192/200	185/200	178/200	937/1000	<b>94.0</b> (+40.9)
	R2	316/400	162/200	29/200	23/200	530/1000		386/400	193/200	184/200	178/200	941/1000	
	R3	314/400	161/200	35/200	21/200	531/1000		384/400	192/200	186/200	180/200	942/1000	
Qwen3-Coder (30B)	R1	381/400	185/200	166/200	167/200	899/1000	89.8	386/400	191/200	187/200	180/200	944/1000	<b>94.4</b> (+4.6)
	R2	381/400	185/200	166/200	167/200	899/1000		387/400	189/200	189/200	181/200	946/1000	
	R3	381/400	185/200	164/200	167/200	897/1000		386/400	190/200	189/200	178/200	943/1000	
Kimi-K2-0905 (1000B)	R1	372/400	183/200	170/200	168/200	893/1000	89.2	387/400	191/200	186/200	187/200	951/1000	<b>94.7</b> (+5.5)
	R2	368/400	181/200	167/200	171/200	887/1000		381/400	189/200	188/200	186/200	944/1000	
	R3	373/400	185/200	173/200	165/200	896/1000		379/400	191/200	188/200	187/200	945/1000	
<b>Closed Source</b>													
Claude Sonnet 4.5	R1	387/400	189/200	184/200	183/200	943/1000	94.4	382/400	189/200	185/200	187/200	943/1000	<b>94.4</b> (0.0)
	R2	388/400	190/200	183/200	182/200	943/1000		384/400	189/200	185/200	186/200	944/1000	
	R3	387/400	190/200	184/200	184/200	945/1000		385/400	189/200	184/200	186/200	944/1000	
GPT-5.1	R1	366/400	183/200	174/200	173/200	896/1000	89.6	367/400	186/200	172/200	176/200	901/1000	88.9 (-0.7)
	R2	367/400	186/200	173/200	169/200	895/1000		354/400	184/200	174/200	174/200	886/1000	
	R3	367/400	185/200	174/200	172/200	898/1000		356/400	180/200	170/200	175/200	881/1000	
Gemini 3 Pro	R1	380/400	190/200	187/200	185/200	942/1000	94.3	382/400	191/200	184/200	186/200	943/1000	<b>94.3</b> (0.0)
	R2	380/400	192/200	188/200	183/200	943/1000		378/400	194/200	187/200	185/200	944/1000	
	R3	384/400	190/200	188/200	182/200	944/1000		380/400	194/200	184/200	185/200	943/1000	

**Performance Analysis.** The results in Table 2 highlight the *atomic* precision of CaveAgent in single-turn scenarios. Key observations include: (1). The results for DeepSeek-V3.2 (w/o prompt) reveal a critical insight. We hypothesize that due to its training emphasis on reasoning regarding tool dependencies (Liu et al., 2025), DeepSeek-V3.2 exhibits a strong inductive bias toward sequential execution, causing it to fail in parallel-calling scenarios under the standard JSON paradigm (53.1% accuracy). To ensure a fair comparison, we added explicit prompting to the system prompt of DeepSeek V3.2 to “force” parallel execution. In stark contrast, CaveAgent achieves SOTA performance (**94.0%**) without any prompt intervention. This demonstrates a unique advantage of our paradigm: by utilizing Python code, CaveAgent naturally supports parallel execution (e.g., via independent lines of code) while simultaneously preserving the capacity to reason about inter-tool dependencies, resolving the conflict between reasoning depth and execution parallelism that standard JSON approaches struggle. (2). The 30B parameter **Qwen3-Coder**, when equipped with our framework, achieves a 94.4% average score, outperforming the much larger proprietary **GPT-5.1** (89.6%) and matching **Claude Sonnet 4.5**. We attribute this to that CaveAgent unlocks more potential of small LLMs via effectively leveraging the inherent coding proficiency of LLMs.

For most SOTA models like Claude Sonnet 4.5 and Gemini 3 Pro, CaveAgent performs on par with the standard baseline (94.3%), with negligible variance. We attribute this plateau to **benchmark saturation**. Current SOTA models have likely reached the upper limit of the BFCL dataset, where remaining errors stem from ambiguous natural language queries or ground-truth noise rather than model incapacity. Since BFCL focuses strictly on single-turn intent detection without the complexity of state maintenance, the “ceiling” is hit relatively quickly. It is important to emphasize that Tau<sup>2</sup>-bench and BFCL serve primarily to validate the **basic** function-calling capabilities of CaveAgent. However, the true superiority of our proposed

**Table 3:** Stateful Management Benchmark Results (success rate %) across three evaluation dimensions. (1) For Type Proficiency, we designed 36, 36, and 42 cases for Simple, Object, and Scientific types, respectively. (2) For Multi-variable Stateful Management, we established 15 evaluation points for each variable-count tier. (3) For Multi-turn Stateful Management, we developed two scenarios, each consisting of 40 turns distributed across two conversations.

Model	Type Proficiency (%)				Multi-Variable (%)					Multi-Turn (%)			
	Simple (36)	Object (36)	Sci. (42)	Avg	5V (15)	10V (15)	15V (15)	20V (15)	25V (15)	Avg	Home (40)	Fin. (40)	Avg
DeepSeek-V3.2	100	100	100	100	100	100	100	100	100	100	100	100	100
Qwen3 Coder	100	94.4	95.2	96.5	94.4	100	80.0	80.0	100	90.9	77.5	85.0	81.3
Kimi K2 0905 Preview	100	100	100	100	100	100	100	100	100	100	90.0	100	95.0
Gemini 3 Pro Preview	100	100	100	100	100	100	100	100	100	100	97.5	100	98.7

paradigm lies in tasks demanding more advanced tool-use—specifically, the manipulation of **complex data objects** over long-horizon tasks. Consequently, existing benchmarks are insufficient to fully capture the stateful management capability of CaveAgent. In the following section, we utilize our hand-crafted cases to provide a deeper and more rigorous assessment of CaveAgent’s capabilities in long-horizon stateful management.

#### 4.2. [Q2] Case Study: Stateful Management

To evaluate CaveAgent’s stateful runtime management capabilities, we design a benchmark targeting multiple complementary dimensions of state manipulation that existing function-calling benchmarks fail to address. The benchmark tests an agent’s ability to read, modify, and persist variables across multiple conversational turns. We divide the measurement of stateful management into three categories: Python type proficiency, capability of multi-variable manipulation, and robustness in multi-turn, long-horizon interaction. A unifying design principle is programmatic validation: rather than parsing text outputs or relying on heuristic matching, we directly inspect runtime state after execution, verifying exact values, object attributes, and data structure contents against ground-truth expectations. This enables precise, unambiguous evaluation and demonstrates a key advantage of CaveAgent’s architecture: agent behavior becomes programmatically verifiable, opening pathways for automated evaluation and reinforcement learning with accurate reward signals.

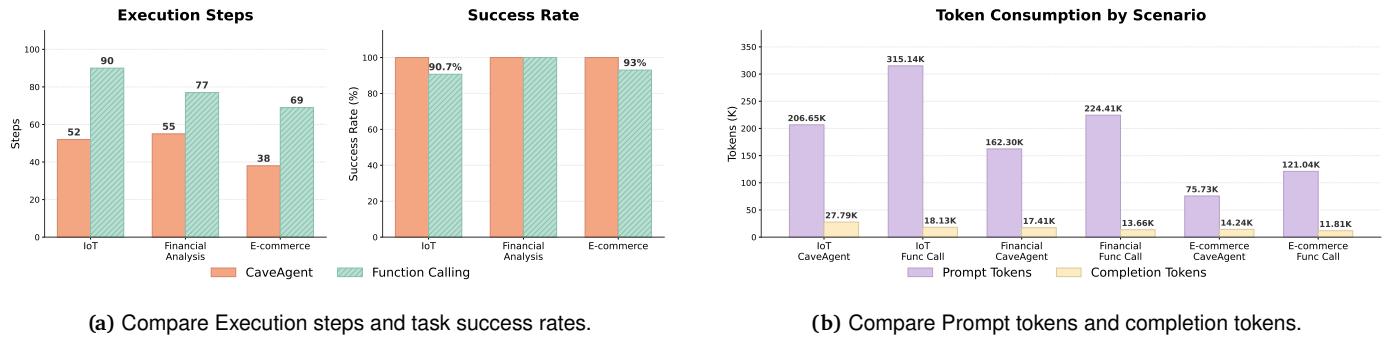
For each dimension, we manually curate multiple test cases, each consisting of multiple natural language queries and an initial variable state, where the queries are linearly dependent. The agent sequentially manipulates the variable according to the queries, after which we retrieve the resulting variable for validation. A query is considered successful if the value of the output variable aligns with our expectations (see Appendix D for the details about the test cases). To isolate core state management capabilities, we craft queries with unambiguous requirements and explicit expected outcomes, ensuring that failures reflect genuine limitations in state tracking rather than instruction misinterpretation. Multiple queries per case further measure long-horizon state persistence and numerical precision across multi-step operations. For each dimension, we select four models to conduct this experiment: QWen3 Coder, Kimi K2 0905, Deepseek V3.2 and Gemini3 Pro. We report the success rate computed by the number of successful queries/total number of queries, as shown in Table 3.

##### 4.2.1. Type Proficiency

**Type Proficiency** aims to evaluate an agent’s ability to manipulate variables across a spectrum of Python types:

- *Simple Types*: Fundamental operations on Python primitive types including integers, floating-point numbers, strings, booleans, lists, and dictionaries.
- *Object Types*: Interaction with user-defined class instances, including attribute access and modification, method invocation, and state tracking across object lifecycles.
- *Scientific Types*: Proficiency with data science primitives commonly used in computational workflows: pandas DataFrames, pandas Series, and NumPy ndarrays. Operations include column creation, filtering, sorting, element-wise transformations, aggregations, and cross-type interactions (e.g., storing array computations as DataFrame columns).

The results yield uniformly high scores (96.5%–100%), validating that code-based manipulation of complex types—including DataFrames, ndarrays, and custom objects—is tractable for current LLMs.



**Figure 5:** Performance comparison between CaveAgent and traditional JSON-based Function Calling across three scenarios, focusing on execution steps, success rates, prompt token and completion token consumptions. Here, the steps means the number of turns needed for task completion, prompt tokens refers to the cumulative input tokens sent to the model across all turns (including system prompts, conversation history, and tool results), and completion tokens refers to the cumulative output tokens generated by the model (including reasoning, function calls, or code generation). The Func Call in right figure represents the abbreviation of Function Calling.

#### 4.2.2. Multi-Variable

The **Multi-Variable** benchmark evaluates how state management accuracy changes when number of variables scale up.

The benchmark comprises five tiers with 5, 10, 15, 20, and 25 concurrent variables, systematically testing the agent’s working memory capacity and ability to perform coordinated state manipulation. Each tier contains 5 multi-turn conversations, and each conversation contains 3 turns, yielding 15 evaluation points per variable tier. The results show no systematic degradation as variable count scales to 25, with top models maintaining 100% accuracy throughout, demonstrating that concurrent state management scales effectively within CaveAgent’s architecture.

#### 4.2.3. Multi-Turn

The **Multi-Turn** benchmark assesses an agent’s ability to read, modify, and persist variables across extended interactions—a critical capability for real-world deployments requiring the tracking of cumulative state changes. The benchmark comprises two domain-specific scenarios, each spanning 40 turns across two conversations:

- **Smart Home:** Simulates a home automation environment where the agent manages devices (e.g., lighting, thermostats) via natural language. This scenario tests the agent’s ability to interpret intent and maintain device state consistency as commands accumulate.
- **Financial Account:** Simulates banking operations such as transfers and inquiries. This scenario specifically targets **numerical precision**—ensuring calculation accuracy over multi-step operations—and **stateful reasoning** within a growing transaction history.

Collectively, these scenarios evaluate **long-horizon state persistence**, measuring whether the agent can reliably modify and track program state without drift as the conversation length increases. The results reveal the most meaningful differentiation between models. Long-horizon state persistence across 40 turns proves challenging: while DeepSeek-V3.2 maintains perfect accuracy, other models exhibit degradation, particularly on Smart Home scenarios requiring object state consistency. This suggests that accumulated state tracking over extended interactions remains the frontier capability for stateful agents. The consistently high accuracy across top models validates our central thesis: when LLMs interact through code with persistent runtime state, **reliable** and **verifiable** agent behavior becomes achievable. Notably, we restrict this evaluation to CaveAgent, as the fine-grained programmatic verification of Python objects is fundamentally incompatible with the text-based outputs of the JSON function-calling paradigm. Nevertheless, the near-perfect performance exhibited by CaveAgent independently substantiates the robustness of our stateful management.

### 4.3. [Q3] Token Efficiency Study

As a complementary experiment, we evaluate CaveAgent’s advantages in context engineering and token efficiency against traditional JSON-based function calling. We benchmark CaveAgent across three domains: IoT device control, financial portfolio analysis, and e-commerce operations. This benchmark specifically targets scenarios requiring logically interdependent tool operations, creating “check → decide → act” cycles where multiple tool calls depend on prior results. We evaluated both

methods using DeepSeek V3.2, measuring success rate, execution steps, and token consumption to isolate and quantify the efficiency gains attributable to the architectural shift from iterative JSON dispatching to native code generation.

Figure 5 shows the full results of this study, and Table 4 summarizes the performance improvement by comparing the summed performance metrics across three domains. The results demonstrate that CaveAgent achieves 28.4% lower total token consumption (504K vs. 704K) while improving task success rate from 94.6% to 100%. The efficiency gain stems from reducing interaction turns. Traditional function calling requires separate request-response cycles for each dependent operation, causing prompt tokens to accumulate as conversation history grows with each turn. CaveAgent instead generates Python code that resolves multiple dependencies in a single execution, reducing total steps from 236 to 145 and consequently cutting prompt tokens by 32.7%. More importantly, the Stateful Management property of CaveAgent naturally reduces the token overhead in multi-turn interaction. This is because CaveAgent manipulates persistent objects via variable references rather than repeatedly serializing full data states into text, as required by stateless process-oriented paradigms.

Notably, CaveAgent consumes 36.3% more completion tokens, since Python code with loops and conditionals is more verbose than JSON schemas. However, prompt tokens dominate overall consumption and accumulate across turns, while completion tokens only account for very small proportion of total token consumption.

**Table 4: Summary of Improvements**

Metric	CaveAgent	Function Calling	Improvement
Total Steps	145	236	-38.6%
Prompt Tokens	444,679	660,588	-32.7%
Completion Tokens	59,440	43,600	+36.3%
Total Tokens	504,119	704,188	-28.4%
Avg. Success Rate	100%	94.6%	+5.4 pp

#### 4.4. [Q4] Case Study: Data-intensive Scenario

To assess the practical benefits of stateful runtime management, we evaluate three agent architectures on a data-intensive benchmark comprising 30 tasks across data query, statistical analysis, and visualization. The benchmark uses stock market data from Apple and Google (2020–2025, Yahoo Finance, retrieved from <https://finance.yahoo.com>). To ensure consistency, all three task categories were equipped with identical data retrieval functions. CodeAct Style replicates standard code-execution agent behavior by disabling CaveAgent’s variable injection and retrieval. JSON-based Function Calling operates without code execution, relying solely on tool outputs fed back to the model. The results are shown in Table 5.

**Data Query Tasks** CaveAgent achieved 100% accuracy while consuming only 123K tokens by storing query results directly in runtime variables, thereby bypassing prompt context accumulation. In contrast, CodeAct Style (80%, 250K tokens) and Function Calling (80%, 295K tokens) both required serializing complete datasets into the conversation history, either through printed output or tool responses, resulting in context overflow failures on high-volume queries. Notably, both baseline methods failed on identical tasks involving large result sets that exceeded the model’s context limit.

**Data Analysis Tasks** For tasks requiring statistical computation (e.g., volatility, correlation, Sharpe ratio), CaveAgent and CodeAct Style both achieved 100% accuracy with comparable token consumption (116–119K), demonstrating that code execution is essential for programmatic analysis. However, Function Calling achieved only 10% accuracy while consuming over 1.3M tokens. Without code execution capabilities, the model could only succeed on a trivial counting task; all computational tasks failed as the model attempted to infer statistics from raw data rather than compute them programmatically.

**Visualization Tasks** These tasks required generating ECharts configurations containing both chart specifications and underlying data arrays. CaveAgent maintained 90% accuracy (405K tokens) by retrieving computed chart data from runtime variables without context serialization. CodeAct Style achieved 40% accuracy but consumed approximately 1M tokens, as generated visualizations must be printed to the conversation for output extraction. Function Calling achieved only 30% accuracy (662K tokens)—lower than CodeAct despite fewer total tokens—because earlier context overflow caused complete failures before task completion.

**Table 5:** Performance Comparison of three Function Calling paradigms across three task categories. **CaveAgent** (highlighted in green) achieves superior performance. Improvements relative to the best baseline are marked in parentheses.

Task Category	Method	Success Rate ↑	Prompt Tokens ↓	Compl. Tokens ↓	Total Tokens ↓
Data Query	<b>CaveAgent</b>	<b>100.0% (+20%)</b>	118,901	4,584	123,485 (-51%)
	CodeAct Style	80.0%	232,990	17,219	250,209
	JSON-based FC	80.0%	278,239	16,413	294,652
Data Analysis	<b>CaveAgent</b>	<b>100.0% (Tie)</b>	110,550	5,832	116,382 (-2%)
	CodeAct Style	100.0%	112,990	6,232	119,222
	JSON-based FC	10.0%	1,328,779	8,024	1,336,803
Visualization	<b>CaveAgent</b>	<b>90.0% (+50%)</b>	374,855	30,250	405,105 (-39%)
	CodeAct Style	40.0%	957,447	43,144	1,000,591
	JSON-based FC	30.0%	644,778	17,899	662,677

**Discussion** These findings demonstrate that stateful runtime management provides substantial efficiency gains for data-intensive agent tasks. By decoupling intermediate computational state from the prompt context, CaveAgent avoids the token accumulation that causes context overflow in conventional architectures. This advantage becomes increasingly crucial as task complexity and data volume scale, suggesting that persistent runtime environments represent a promising direction for building robust agentic systems capable of handling real-world data processing workloads.

## 5. Conclusion

We present **CaveAgent**, a novel framework that transforms LLM tool use from stateless JSON function calling to persistent, object-oriented stateful runtime management. CaveAgent enables agents to maintain high-fidelity memory of complex objects and execute sophisticated logic via Python code. Experiments on Tau<sup>2</sup>-bench show that this approach significantly outperforms SOTA baselines in multi-turn success rates (+10.5%) and token efficiency. Crucially, the programmatic verifiability of the runtime state provides a rigorous ground for future advancements in Reinforcement Learning and runtime-mediated multi-agent coordination, marking a critical step towards more reliable and capable autonomous agents. Qualitative case studies are provided in Appendix F.

## 6. Related Work

### 6.1. Tool Learning & Instruction Following (JSON-centric Paradigm)

The foundational approach to equipping Large Language Models (LLMs) with agency has relied on a "Classification-Slot Filling" paradigm, where models interface with external environments via structured data formats, predominantly JSON. Seminal works such as ToolLLM (Qin et al., 2023) and Gorilla (Patil et al., 2024) demonstrated that LLMs could be fine-tuned to navigate massive API indices and mitigate hallucinations by strictly adhering to predefined schemas. This structured interaction model was further formalized by industry standards like GPT-4 Function Calling and adopted by agentic frameworks such as ReAct (Yao et al., 2022b) and the JSON mode of AutoGen (Wu et al., 2024), which orchestrate reasoning through iterative schema population. Nevertheless, the JSON-centric paradigm imposes severe architectural constraints. First, JSON is inherently a static data interchange format lacking native control flow; it cannot represent loops or conditional logic, forcing agents into expensive, multi-turn interactions to execute complex workflows (Wang et al., 2024). Second, the verbose syntax of JSON introduces significant token overhead, resulting in low information density and high latency (Wang et al., 2024). Finally, the rigidity of schema enforcement creates a fragility trade-off, where complex nested structures increase the probability of syntax errors and hallucination (Patil et al., 2024).

### 6.2. Code as Action & Programmatic Reasoning

Recognizing the limitations of static schemas, recent research has explored "Code as Action" paradigm, where executable code (primarily Python) serves as the unified medium for reasoning and tool invocation. Wang et al. (2024) challenged the JSON-Schema convention with *CodeAct*, proposing executable Python code as a unified representation for both reasoning and action. Their work demonstrated that code-based interactions reduce multi-turn overhead by up to 30% compared

to JSON-based methods and improve task success rates by 20% on benchmarks like M3ToolEval. This shift utilizes the Turing-complete nature of code to naturally express complex logic, loops, and variable dependencies that are cumbersome in JSON. This paradigm extends to domain-specific reasoning. Surís et al. (2023) introduced *ViperGPT*, which composes vision modules into executable subroutines to solve visual queries, rendering the reasoning process interpretable. Similarly, Chen et al. (2022) and Gao et al. (2023) proposed *Program of Thoughts (PoT)* and *Program-aided Language Models (PAL)*, respectively. These frameworks decouple computation from reasoning by delegating arithmetic and symbolic logic to a Python interpreter, thereby mitigating the calculation errors common in pure language models. The efficacy of these methods is further amplified by code-optimized models such as *DeepSeek-Coder-V2* (Zhu et al., 2024), which exhibit superior performance in following complex programmatic instructions.

### 6.3. Context Management & Stateful Architectures

The constraints of LLM context windows have necessitated advanced memory management strategies. Packer et al. (2023) introduced *MemGPT*, which implements an OS-inspired virtual context management system, organizing memory into tiers (main vs. external) to handle long-horizon tasks. Similarly, Qiao et al. (2023) proposed *TaskWeaver*, a code-first framework that attempts to maintain stateful execution for data analytics by preserving data structures like DataFrames across turns. However, existing approaches largely rely on *Retrieval-Augmented Generation (RAG)* or textual summarization to manage context. These methods are inherently **lossy**: converting complex runtime objects (e.g., high-dimensional matrices, class instances) into text or vector embeddings strips them of their structural integrity and executable properties. **CaveAgent** addresses this by proposing *Runtime-based Context Compression*. Unlike prior work that externalizes state to vector stores, we utilize *Variable Injection* to treat the Python runtime itself as a high-fidelity external memory. This allows arbitrary variables to be persisted in their native object form, maintaining full manipulability without the overhead of re-tokenization or the information loss associated with serialization.

### 6.4. Multi-Agent Coordination Mechanisms

Research into multi-agent systems has focused on structuring collaboration through natural language communication. Li et al. (2023a) proposed *CAMEL*, a role-playing framework that facilitates autonomous cooperation via communicative agents. Building on this, Qian et al. (2024) introduced *ChatDev*, which organizes agents into a "chat chain" following a waterfall software development model, while Hong et al. (2023) developed *MetaGPT*, which encodes Standardized Operating Procedures (SOPs) into agent prompts to streamline complex workflows. However, these frameworks predominantly rely on **text-based message passing** for coordination. This architecture introduces a critical *serialization bottleneck*: transferring complex state (e.g., a trained machine learning model or a processed dataset) between agents requires converting it into natural language descriptions or intermediate files, leading to high latency and potential ambiguity. **CaveAgent** overcomes this via *Runtime-Mediated State Flow*. By leveraging the shared variable space established in our runtime architecture, agents collaborate by directly injecting and retrieving variables. This shifts the coordination paradigm from "communication by talking" to "communication by shared state," enabling atomic, lossless, and zero-latency information exchange.

## 7. Acknowledgment

We thank Rui Zhou, a professional UI designer affiliated with Hong Kong Generative AI Research & Development Center, HKUST, for his professional contributions to the figure design of this paper.

## References

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Daniil A Boiko, Robert MacKnight, and Gabe Gomes. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332*, 2023.
- Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376*, 2023.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

- Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, et al. Palm-e: An embodied multimodal language model. 2023.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.
- Shengtao He. Achieving tool calling functionality in llms using only prompt engineering without fine-tuning, 2024. URL <https://arxiv.org/abs/2407.04997>.
- 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.
- Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36:39648–39677, 2023.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.
- Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbulin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023a.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023b.
- Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.
- 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.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities, 2025. URL <https://arxiv.org/abs/2408.04682>.
- Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36:43447–43478, 2023.
- Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *IEEE Transactions on Audio, Speech and Language Processing*, 2025.
- Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
- Charles Packer, Vivian Fang, Shishir\_G Patil, Kevin Lin, Sarah Wooders, and Joseph\_E Gonzalez. Memgpt: Towards llms as operating systems. 2023.
- Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

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. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

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.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 11888–11898, 2023.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023a.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023b.

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.

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.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, et al. If llm is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *arXiv preprint arXiv:2401.00812*, 2024.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022b.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR, 2023.

# Appendix

<b>A Pseudo Code</b>	<b>21</b>
<b>B What Happens in Semantic Stream</b>	<b>21</b>
B.1 System Prompt Construction . . . . .	21
B.2 Context Injection Format . . . . .	22
B.3 Runtime Feedback Prompts . . . . .	23
<b>C What Happened in Runtime Stream</b>	<b>24</b>
C.1 Environment Initialization via Injection . . . . .	24
C.2 The Interleaved Execution Paradigm . . . . .	24
C.3 Illustrative Case Study . . . . .	25
<b>D Test Cases in Stateful Management Benchmark</b>	<b>25</b>
D.1 Type Proficiency Cases . . . . .	25
D.2 Multi-variable Cases . . . . .	26
D.3 Multi-turn Cases . . . . .	26
<b>E Stateful Runtime-Mediated Multi-Agent Coordination</b>	<b>27</b>
E.1 Meta-Agent Runtime Control . . . . .	28
E.2 State-Mediated Communication . . . . .	28
E.3 Shared-Runtime Synchronization . . . . .	28
<b>F Features</b>	<b>28</b>
F.1 Case Analysis in Tau <sup>2</sup> -bench . . . . .	28
F.2 Smart Home . . . . .	30

**Algorithm 1** CaveAgent Interaction Loop

---

```

Require: Query  $q$ , Tools  $\mathcal{T}$ , Max Turns  $T_{\max}$ 
1:  $\mathcal{S}_0 \leftarrow \text{INITKERNEL}(); \text{INJECT}(\mathcal{S}_0, \mathcal{T})$                                 ▷ Init Runtime Stream
2:  $D \leftarrow \text{GENSIGS}(\mathcal{T}); H_0 \leftarrow \{\text{Sys}(D), \text{User}(q)\}$                   ▷ Init Semantic Stream
3: for  $t = 1$  to  $T_{\max}$  do
4:   Phase 1 (Reasoning):  $R_t \leftarrow \text{LLM}(H_{t-1})$                                          ▷ Sample thought & code
5:   if  $R_t$  contains code block  $c_t$  then
6:     Phase 2 (Security):  $V \leftarrow \text{ASTCHECK}(c_t, \Pi)$                                      ▷ Pre-exec validation
7:     if  $V \neq \emptyset$  then
8:        $o_t \leftarrow \text{FormatError}(V)$ 
9:     else
10:      Phase 3 (Execution):  $o_t, \mathcal{S}_t \leftarrow \text{RUN}(\mathcal{S}_{t-1}, c_t)$                          ▷ Stateful update
11:      Phase 4 (Shaping):  $o_t \leftarrow \text{SHAPE}(o_t, L_{\max})$                                ▷ Truncate & format
12:    end if
13:     $H_t \leftarrow H_{t-1} \cup \{(R_t, o_t)\}$                                               ▷ Sync observation to history
14:  else
15:    return  $R_t$                                                                ▷ Output final answer
16:  end if
17: end for
18: return "Max steps reached"

```

---

**A. Pseudo Code**

Algorithm 1 shows the general workflow of CaveAgent.

**B. What Happens in Semantic Stream**

The following sections detail the exact prompt templates used to instruct the Semantic Stream in CaveAgent to help readers understand what happens in this stream. The system prompt is dynamically constructed by combining the Agent Identity, Context Information (functions, variables, types), and specific Instructions.

**B.1. System Prompt Construction**

The full system prompt is composed using the following template structure. The placeholders (e.g., {functions}) are populated at runtime with the specific tools and variables available in the current environment.

System Prompt Template
<pre> {agent_identity}  Current time: {current_time}  You have access to:  &lt;functions&gt; {functions} &lt;/functions&gt;  &lt;variables&gt; {variables} &lt;/variables&gt;  &lt;types&gt; {types} &lt;/types&gt; </pre>

```
Instructions:  

{instructions}  

{additional_context}
```

Below are the default values for the key components referenced in the template above.

#### Component: Agent Identity

You are a tool-augmented agent specializing in Python programming that enables function-calling through LLM code generation. You have to leverage your coding capabilities to interact with tools through a Python runtime environment, allowing direct access to execution results and runtime state. The user will give you a task and you should solve it by writing Python code in the Python environment provided.

#### Component: Core Instructions

1. Carefully read and analyze the user's input.
2. If the task requires Python code: - Generate appropriate Python code to address the user's request. - Your code will then be executed in a Python environment, and the execution result will be returned to you as input for the next step.  
- During each intermediate step, you can use 'print()' to save whatever important information you will then need in the following steps. - These print outputs will then be given to you as input for the next step. - Review the result and generate additional code as needed until the task is completed.
3. CRITICAL EXECUTION CONTEXT: You are operating in a persistent Jupyter-like environment where: - Each code block you write is executed in a new cell within the SAME continuous session - ALL variables, functions, and imports persist across cells automatically - You can directly reference any variable created in previous cells without using locals(), globals(), or any special access methods.
4. If the task doesn't require Python code, provide a direct answer based on your knowledge.
5. Always provide your final answer in plain text, not as a code block.
6. You must not perform any calculations or operations yourself, even for simple tasks like sorting or addition.
7. Write your code in a {python\_block\_identifier} code block. In each step, write all your code in only one block.
8. Never predict, simulate, or fabricate code execution results.
9. To solve the task, you must plan forward to proceed in a series of steps, in a cycle of Thought and Code sequences.

## B.2. Context Injection Format

Examples of how context is formatted for the LLM.

#### Example: Function Injection

```
<functions>
- function: buy_stock(symbol: str, quantity: int) -> Transaction
  description: Executes a stock purchase for the current portfolio.
  doc:
    Args:
      symbol: The ticker symbol of the stock (e.g., 'AAPL').
```

quantity: The number of shares to purchase.

Returns:

A Transaction object recording the details of the purchase.

</functions>

### Example: Variable Injection

```
<variables>
- name: portfolio
  type: Portfolio
  description: The user's current investment portfolio object.

- name: market_data
  type: DataFrame
  description: A pandas DataFrame containing historical price data.
</variables>
```

### Example: Type Schema Injection

```
<types>
Portfolio:
  doc: Manages a collection of stock holdings and cash balance.
  methods:
    - get_total_value() -> float
    - get_holdings() -> Dict[str, int]
    - add_cash(amount: float) -> None

Transaction:
  doc: An immutable record of a stock transaction.
  fields:
    - id: str
    - symbol: str
    - quantity: int
    - price_at_execution: float
    - timestamp: datetime
</types>
```

## B.3. Runtime Feedback Prompts

The agent operates in a closed feedback loop. After each code execution step, the runtime environment captures the output (stdout or errors) and constructs a new user message to guide the agent's next action.

### B.3.1. Standard Execution Output

This prompt is used when code executes successfully. It provides the standard output and explicitly reminds the agent that the variable state has been preserved.

#### Execution Output Template

```
<execution_output>
{execution_output}
</execution_output>
```

**IMPORTANT CONTEXT REMINDER:** - Based on this output, should we continue with more operations?

- If the output includes an error, please review the error carefully and modify your code to fix the error if needed.

- If yes, provide the next code block. If no, provide the final answer (not as a code block).
- You are in the SAME Jupyter-like session. All variables from your previous code blocks are still available and can be accessed directly by name.
- You DO NOT need to use locals(), globals(), or any special methods to access them.
- Think of this exactly like working in Jupyter: when you create a variable in cell 1, you can simply use it by name in cell 2, 3, 4, etc.

### B.3.2. Error Handling & Constraints

The system includes specific templates for handling edge cases, such as context window limits and security violations.

**Output Length Exceeded:** Used when the code generates excessive output (e.g., printing a massive DataFrame), prompting the agent to summarize instead.

#### Output Truncation Template

The code execution generated output\_length characters of output, which exceeds the maximum limit of max\_length characters. Please modify your code to:

1. Avoid printing large datasets or lengthy content
2. Use summary statistics instead of full data (e.g., print shape, head(), describe() for dataframes)
3. Print only essential information needed for the task """"

**Security Violation:** Used when the static analysis security checker blocks unsafe code (e.g., os.system).

#### Security Error Template

```
<security_error>
{error}
</security_error>
```

Code blocked for security reasons. Please modify your code to avoid this violation.

## C. What Happened in Runtime Stream

While the Semantic Stream governs the agent's reasoning and planning, the **Runtime Stream** serves as the system's execution engine and persistent memory. This stream operates as a dedicated Python kernel where the actual "work" of the agent (data manipulation, tool invocation, and state transitions) occurs. The interaction between the two streams follows a strict chronological topology, synchronized through an interleaved exchange of code instructions and execution feedback.

### C.1. Environment Initialization via Injection

The runtime lifecycle begins not with an empty state, but with **Context Injection**. Before the reasoning cycle commences, the user (or the system orchestration layer) initializes the runtime environment by injecting native Python objects directly into the global namespace.

- **Function Injection:** Tool definitions are loaded as executable Python callables. Unlike RESTful API wrappers, these are native functions that can be inspected and invoked directly.
- **Variable Injection:** Domain-specific data—such as complex DataFrames, graph structures, or class instances—are instantiated within the "memory" in the runtime stream.

This initialization phase populates the <functions> and <variables> blocks described in Section B.

### C.2. The Interleaved Execution Paradigm

Once initialized, the workflow proceeds as a synchronized dialogue between the Semantic Stream (Reasoning) and the Runtime Stream (Execution). We conceptualize this as a dual-column timeline where actions are interleaved strictly in chronological order:

1. **Semantic Turn (Left Cell):** The LLM analyzes the current task and available context. It generates a *Thought* followed by a discrete *Code Block* (the instruction). This represents the input to the runtime.
2. **Runtime Turn (Right Cell):** The system extracts the code block and executes it within the persistent Python kernel. This execution constitutes the state transition  $S_t \rightarrow S_{t+1}$ . Crucially, this is not a stateless function call; it is a stateful operation where:
  - New variables defined in this cell are persisted in memory.
  - Existing objects (e.g., a list or a database connection) are mutated in place.
  - Side effects (e.g., saving a file) are realized immediately.
3. **Feedback Loop:** Upon completion of the Runtime Turn, the standard output (`stdout`), standard error (`stderr`), or the return value of the last expression is captured. This raw execution result is wrapped in the `<execution_output>` tags and injected back into the Semantic Stream, triggering the next Semantic Turn.

This mechanism ensures that the agent’s reasoning is always grounded in the current, actual state of the runtime environment.

### C.3. Illustrative Case Study

To intuitively demonstrate the temporal synchronization and state dependency between the two streams, we present a concrete walkthrough in Figure 6. This example illustrates a toy data analysis task where the agent must filter a dataset and perform calculations on the result.

The workflow proceeds in a “zig-zag” pattern, alternating between reasoning (Left) and execution (Right):

1. **Initialization ( $T_0$ ):** The user injects a pandas DataFrame named `df`. Note that the Semantic Stream only receives a lightweight pointer (variable name and documentation) instead of the whole data, while the Runtime Stream holds the actual heavy data object in memory.
2. **Step 1 ( $T_1 \rightarrow T_2$ ):** The agent generates code to filter the data. Crucially, the Runtime Stream does not return the full filtered dataset as text. Instead, it creates a new variable `high_vol` in the local scope and returns only a status update. This exemplifies our **Stateful Management**: the “result” of the tool use is a state change in memory, not a text string.
3. **Step 2 ( $T_3 \rightarrow T_4$ ):** The agent references the *previously created* variable `high_vol` to compute a statistic. This demonstrates **Context Compression**: the agent manipulates the data via variable references without ever consuming context tokens to “read” the full dataset.

The analogy to view the runtime-stream as a jupyter-notebook with multiple cells (where each cell corresponds to the execution of Runtime Stream of each time step) could help us understand the mechanism of stateful management, especially how the states remain persistent across each cell.

## D. Test Cases in Stateful Management Benchmark

In this section, we provide the examples of our test cases in Stateful Management Benchmark.

### D.1. Type Proficiency Cases

The **Type Proficiency** category evaluates the agent’s competency in precise, state-aware manipulation of Python runtime elements. Unlike generic code generation, this section rigorously tests the agent’s “working memory” across three structural tiers: *Simple Types* (primitives types such as list, dictionary and string), *Object Types* (custom classes), and *Scientific Types* (high-dimensional complex data). Mastery of these domains serves as the foundational prerequisite for complex reasoning tasks.

#### D.1.1. Simple Types

Figure 7 shows the examples of our test cases of Simple types.

#### D.1.2. Object Types

Figure 8 shows the examples of our test cases of Object types.

Timeline of Interleaved Execution (Semantic vs. Runtime)		
Time	Semantic Stream (Reasoning & Code)	Runtime Stream (State & Execution)
$T_0$	(Waiting for Input)	[Initialization] <ul style="list-style-type: none"> <li>• df = pd.read_csv("market.csv") loaded into memory.</li> <li>• &lt;variables&gt; metadata sent to Left Stream.</li> </ul>
$T_1$	User: "Find stocks with volume > 1M." Agent Thought: I need to filter the dataframe df. I will save the result to a new variable. Code: <code>high_vol = df[df['volume'] &gt; 1e6] print(f"Filtered. Rows: {len(high_vol)}")</code>	(Waiting for Instruction)
$T_2$	(Waiting for Feedback)	[Execution] <ol style="list-style-type: none"> <li>Evaluates filter logic on df.</li> <li>New State: Variable <code>high_vol</code> created in RAM.</li> <li>Output: "Filtered. Rows: 42" sent to Left Stream.</li> </ol>
$T_3$	Input: "Filtered. Rows: 42" Agent Thought: Now I need to calculate the average price of these high-volume stocks. Code: <code>avg_price = high_vol['price'].mean() print(round(avg_price, 2))</code>	(Waiting for Instruction)
$T_4$	(Waiting for Feedback)	[Execution] <ol style="list-style-type: none"> <li>Accesses persistent object <code>high_vol</code>.</li> <li>Computes mean.</li> <li>Output: 154.20 sent to Left Stream.</li> </ol>

**Figure 6:** A visualization of the CaveAgent workflow. The process alternates between the Semantic Stream (generating instructions) and the Runtime Stream (executing and maintaining state). Note how the variable `high_vol` is maintained in the Runtime Stream ( $T_2$ ) and accessed in the subsequent step ( $T_4$ ) without re-loading or serialization, illustrating the efficiency of Stateful Runtime Management.

### D.1.3. Scientific Types

Figure 9 shows the examples of our test cases of Scientific types.

## D.2. Multi-variable Cases

Since there are 5 tiers of variable numbers, we select the variable number = 20 to demonstrate our test case since different variable number shares similar patterns of test cases. Figure 10 shows one example of test case where the agent is required to process 20 variables in 3 turns.

## D.3. Multi-turn Cases

This class of test cases is designed to evaluate the agent's capability to process complex, sequential instructions and maintain state precision over long-horizon scenarios. Unlike single-turn tasks where information is self-contained, these scenarios require the agent to maintain a persistent memory of the system's status, as subsequent queries often depend on the outcome of previous actions. We categorize these multi-turn benchmarks into two distinct domains: **Smart Home Control** and **Financial Account Management**.

### D.3.1. Smart Home

In the **Smart Home** scenario, the agent acts as a central automation controller responsible for managing a suite of simulated IoT devices, including smart lighting, thermostats, motorized blinds, security cameras, and media players.

This benchmark specifically targets two advanced capabilities in stateful management:

- Users frequently issue relative commands rather than absolute ones (e.g., "turn up the music *more*" or "dim the lights *a bit*"). To execute these correctly, the agent must recall the exact discrete level set in previous turns (e.g., incrementing volume from 'medium' to 'high') rather than resetting to a default value.

Turn	User Query (Input)	Immediate Validation (State Assertion)
<b>Case: string_split_join</b>		
T1	“Set text to ‘a,b,c’, split by comma, and rejoin with ‘   ’ as separator...”	<b>validate_str_split</b> • Assert <code>text == "a   b   c"</code> .
T2	“Sort the parts of text alphabetically while keeping the ‘   ’ separator format.”	<b>validate_str_sort</b> • Assert <code>text remains "a   b   c"</code> . • Checks persistence of structure.
T3	“Reverse the order of parts in text but keep the ‘   ’ separator...”	<b>validate_str_reverse</b> • Assert <code>text == "c   b   a"</code> .
<b>Case: dict_nested</b>		
T1	“Change the math score to 90 in data[‘scores’][‘math’].”	<b>validate_dict_nested_update</b> • Assert <code>data[‘scores’][‘math’] == 90</code> .
T2	“The student just took a science test. Add a science score of 88 to data[‘scores’].”	<b>validate_dict_nested_add</b> • Assert key ‘science’ exists with value 88.
T3	“There was a curve on all tests. Add 5 points to every score in the scores dictionary.”	<b>validate_dict_increment</b> • Assert <code>math == 95 (90+5)</code> . • Assert <code>science == 93 (88+5)</code> . • Assert <code>english == 95 (Initial 90+5)</code> .

Figure 7: Illustration of test cases of Simple Types. The results show the agent’s capability to manipulate any object types.

- The agent must dynamically adjust device states based on simulated environmental contexts (e.g., “sunset”, “motion detected”) and complex user-defined conditions (e.g., “if the temperature drops below 10°C, set heating to 22°C”).

As illustrated in Figure 11, the `weekend_party` case spans a simulated 24-hour cycle. The agent must maintain a coherent environment state—transitioning from a quiet morning to a loud party and finally to a secure night mode, without drifting from the user’s cumulative intent.

#### D.3.2. Financial Account

The **Financial Account** benchmark evaluates the agent’s capability to maintain **strict numerical integrity** and execute **state-dependent logic** within a banking ledger system. Unlike the relative adjustments in Smart Home, this domain demands exact integer arithmetic, where the agent must process a continuous stream of transactions—including deposits, interest applications, and loan amortizations—without cumulative drift.

This scenario imposes two critical constraints designed to stress-test the agent’s reasoning stability:

- Operations require strict integer truncation (e.g., calculating 20% of 1105 as 221, not 221.0). Since the output of each turn (e.g., current balance) serves as the immutable basis for subsequent calculations (e.g., compound interest), a single arithmetic error in early turns triggers a cascading failure, rendering the entire subsequent interaction trajectory incorrect.
- The agent must evaluate complex logic gates based on dynamic runtime states rather than static instructions. As demonstrated in the `carol_debt_paydown` case (Figure 12), queries often involve comparative functions (e.g., “pay the *smaller* of 15% of balance or 15% of loan”) or threshold checks (e.g., upgrading to ‘premium’ status only if net worth becomes positive). This requires the agent to retrieve, compare, and act upon multiple variable states simultaneously before executing a transaction.

## E. Stateful Runtime-Mediated Multi-Agent Coordination

The function calling paradigm in CaveAgent introduces three foundational innovations for multi-agent coordination; Figure 1 illustrates an intuitive and straightforward example of these implications. In this paper, we primarily focus on qualitative analysis and provide intuitive case studies to facilitate understanding, leaving rigorous methodological development and quantitative justification for future work. We introduce the high-level idea below.

Object Types		
Turn	User Query (Input)	Immediate Validation (State Assertion)
<b>Case: stack_advanced</b>		
T1	“Push 'A', 'B', 'C', 'D' in order.”	<code>validate_stack_multi_push</code> • Assert <code>stack.size() == 4.</code>
T2	“User wants to go back to first page. Pop until only 1 item remains, store count in <code>result_num</code> .”	<code>validate_stack_pop_until</code> • Assert <code>stack.size() == 1.</code> • Assert <code>result_num == 3</code> (Popped D,C,B).
T3	“Verify we’re at the right page. Peek at stack’s top and store in <code>result_str</code> .”	<code>validate_stack_peek_after</code> • Assert <code>result_str == 'A'.</code> • Assert <code>stack.size() == 1.</code>
<b>Case: cart_quantity</b>		
T1	“Add 3 Apples at \$10.00 each to cart with quantity.”	<code>validate_cart_qty_add</code> • Assert <code>len(cart.items) == 1.</code> • Assert <code>items[0]['quantity'] == 3.</code>
T2	“Also add 2 Oranges at \$5.00 each...”	<code>validate_cart_qty_add2</code> • Assert <code>len(cart.items) == 2.</code>
T3	“Calculate total (price * quantity)... store in <code>result_num</code> .”	<code>validate_cart_qty_total</code> • Assert <code>result_num == 40.0.</code> • Logic: $(3 \times 10) + (2 \times 5)$ .

**Figure 8: Illustration of test cases of Object Types.** The results show the agent’s capability to manipulate custom class instances (Stack, ShoppingCart, Person) and verifying their internal attributes and method side-effects.

### E.1. Meta-Agent Runtime Control

Sub-agents are injected as first-class objects into an Meta-agent’s runtime, enabling the Meta-agent to programmatically access and manipulate child agent states through generated code. Rather than following predefined communication protocols, the Meta-agent dynamically sets variables in sub-agent runtimes, triggers execution, and retrieves results, enabling adaptive pipeline construction, iterative refinement loops, and conditional branching based on intermediate states.

### E.2. State-Mediated Communication

Inter-agent data transfer bypasses message-passing entirely. Agents communicate through direct runtime variable injection: the Meta-agent retrieves objects from one agent’s runtime and injects them into another’s as native Python artifacts (DataFrames, trained models, statistical analyses), preserving type fidelity and method interfaces without serialization loss.

### E.3. Shared-Runtime Synchronization

For peer-to-peer coordination, multiple agents can operate on a unified runtime instance, achieving implicit synchronization without explicit messaging. When one agent modifies a shared object, all peers perceive the change immediately through direct reference. New entities injected into the shared runtime become instantly discoverable, enabling emergent interaction and collaborative manipulation of a unified “world” model with low coordination overhead.

**How town simulation demonstrates this capability** When the Meta-agent modifies the weather state, all resident agents observe the change through direct attribute access; when a new location and manager are injected, existing agents can immediately query and interact with them.

Together, these patterns transform multi-agent systems from lossy text-based message exchange into typed, verifiable state flow, and enable automated validation of inter-agent handoffs and seamless integration with downstream pipelines.

## F. Features

### F.1. Case Analysis in Tau<sup>2</sup>-bench

To empirically validate the architectural advantages of CaveAgent, we analyzed trajectory differences on the tau<sup>2</sup>-bench retail benchmark. CaveAgent achieved a 72.8% success rate (83/114) compared to 62.3% (71/114) for the baseline JSON

Turn	User Query (Input)	Immediate Validation (State Assertion)
<b>Case: dataframe_merge (Relational Logic)</b>		
T1	“Merge df and df2 on product column. Store in result_df.”	<b>validate_df_merge</b> • Assert len(result_df) == 3. • Assert column “supplier” exists.
T2	“Update result_df to keep only rows where supplier is ‘SupA’.”	<b>validate_df_merge_filter</b> • Assert len(result_df) == 2. • Logic: Keeps ‘Phone’ and ‘Shirt’.
T3	“Calculate the sum of prices in result_df. Store in result_value.”	<b>validate_df_merge_sum</b> • Assert result_value == 550.0. • Logic: 500.0 + 50.0.
<b>Case: dataframe_pivot (Structure Reshaping)</b>		
T1	“Create pivot table from df_sales: region=rows, quarter=cols, sales=values.”	<b>validate_df_pivot</b> • Assert result_df.shape == (3, 2). • Checks dimensions (3 regions, 2 quarters).
T2	“Calculate total sum of all sales...”	<b>validate_df_pivot_sum</b> • Assert result_value == 890. • Verifies data integrity post-pivot.
T3	“Find which region has highest total sales (sum of Q1+Q2). Store sum...”	<b>validate_df_pivot_max_region</b> • Assert result_value == 380. • Logic: South (200 + 180).
<b>Case: ndarray_reshape (Tensor Manipulation)</b>		
T1	“Reshape array to shape (2, 4). Store in result_array.”	<b>validate_array_reshape</b> • Assert result_array.shape == (2, 4). • Checks memory layout transformation.
T2	“Sum result_array along axis 1 (row sums).”	<b>validate_array_sum_axis</b> • Assert result equals [70, 48]. • Validates axis-wise reduction.
T3	“Calculate the total sum of result_array...”	<b>validate_array_total</b> • Assert result_value == 118. • Logic: 70 + 48.

**Figure 9: Illustration of Scientific Types test cases.** This benchmark challenges the agent with high-dimensionality operations, including relational merges (dataframe\_merge), structural reshaping (dataframe\_pivot), and tensor axis manipulation (ndarray\_reshape), going beyond simple arithmetic.

agent (Kimi K2 backbone), yielding a 10.5% improvement. We conducted a root cause analysis on the 24 tasks where CaveAgent succeeded but the baseline failed.

#### F.1.1. Failure Taxonomy of the Baseline

Baseline failures were categorized into five distinct patterns (Figure 13). The dominant failure mode (37.5%) was *Missing Critical Action*, where the agent retrieved necessary information but failed to execute the final operation (e.g., return, cancel). This was often coupled with *Incomplete State Exploration* (16.7%), where the agent heuristically queried subsets of data (e.g., checking only one recent order) rather than performing the exhaustive search required by the query.

#### F.1.2. Architectural Advantages: Loops and Conditionals

The analysis reveals that CaveAgent’s superiority stems from its ability to generate programming constructs, specifically loops (used in 92% of winning cases) and conditionals (83%), which resolve the semantic gaps inherent in single-step function calling.

**Exhaustive State Exploration via Loops.** Tasks requiring global search (e.g., “return the order sent to Texas”) baffled the baseline agent, which typically checked only 1–2 arbitrary orders. In contrast, CaveAgent generated for-loops to iterate through all user orders. For instance, in Task 26, the agent iterated through `user.orders`, checked `order.address.state` for “TX”, and correctly identified the target order without hallucination.

Turn	Complex User Query (Full Text)	State Verification (Partial View)
<i>Case: startup_journey (20 Variables)</i>		
T1	"I'm documenting our startup TechStart. We're in the Software industry, led by CEO Alice Johnson, headquartered in San Francisco. We have 50 employees, founded in 2020, 1 office, 2 products. Revenue is \$5M (5000000) with 10% profit margin (0.1), not public yet so no stock price or market cap. We're profitable and hiring but not international yet. Departments: ['Engineering', 'Sales', 'Marketing']. Locations: ['SF']. Financials: funding 10000000, round 'Series A'. Contacts: email 'info@techstart.com', phone '555-0100'."	<b>validate_startup_init</b> <ul style="list-style-type: none"> <li>• <code>employees</code> → 50</li> <li>• <code>revenue</code> → 5,000,000.0</li> <li>• <code>profit_margin</code> → 0.1</li> <li>• <code>public</code> → False</li> <li>• <code>stock_price</code> → 0.0 (<i>Initial</i>)</li> </ul>
T2	"Big growth update! Set employees to 150, offices to 3, products to 5. Set revenue to \$15M (15000000), profit_margin to 0.15. Set international to true. Append 'HR' and 'Finance' to departments. Append 'NYC' and 'London' to locations. Add 'valuation': 10000000 to financials while keeping existing entries. Add 'support': '555-0200' to contacts while keeping existing entries."	<b>validate_startup_growth</b> <ul style="list-style-type: none"> <li>• <code>employees</code> → 150</li> <li>• <code>depts</code> → [..., 'HR', 'Finance']</li> <li>• <code>financials</code> → +{'valuation': 100M}</li> <li>• Assert <code>founded_year == 2020</code> (<i>Unchanged</i>)</li> </ul>
T3	"We're going public! Append ' Inc.' to company_name. Set industry to 'Enterprise Software'. Set employees to 500, offices to 10, products to 10. Set revenue to \$50M (50000000), profit_margin to 0.2, stock_price to 25.0, market_cap to \$500M (500000000). Set public to true. Append 'Legal' and 'IR' to departments. Append 'Tokyo' and 'Berlin' to locations. Add 'ipo': true to financials while keeping existing entries. Add 'ir': 'irtechstart.com' to contacts while keeping existing entries."	<b>validate_startup_ipo</b> <ul style="list-style-type: none"> <li>• <code>public</code> → True</li> <li>• <code>stock_price</code> → 25.0</li> <li>• <code>company_name</code> → "TechStart Inc."</li> <li>• <code>market_cap</code> → 500,000,000.0</li> </ul>

**Figure 10: High-Dimensional State Management (Full Transcript).** We present the raw input queries for the `startup_journey` case. The high information density requires the agent to parse and update over 10 distinct variables (Integers, Floats, Strings, Lists, Dictionaries) in a single turn (e.g., T3) without hallucination or omitting details.

**Listing 1:** Snippet from Task 26 showing exhaustive search.

```
# CaveAgent: Systematic iteration ensures no order is missed
for order_id in user_details.orders:
    order = get_order_details(order_id)
    if "TX" in order.address.state:
        return_delivered_order_items(order_id, ...)
```

**Complex Conditional Logic.** The baseline struggled with tasks involving fallback logic (e.g., "modify item, but if price > \$3000, cancel order"). In Task 90, the JSON agent ignored the price constraint and attempted modification regardless. CaveAgent successfully modeled this decision tree using explicit `if/else` blocks, checking variable states (`variant.price`) before execution.

**Precise Attribute Reasoning.** While JSON agents rely on the LLM's internal attention to compare values (often leading to errors like cancelling the wrong order in Task 59), CaveAgent offloads reasoning to the Python interpreter. By storing intermediate results (e.g., timestamps) in variables and using comparison functions (e.g., `min()`), CaveAgent ensured precise argument selection for actions requiring temporal or numerical comparisons.

## F.2. Smart Home

Figure 14 illustrates the mechanistic advantage of CaveAgent through a toy smart-home example. The architecture separates the *Semantic Stream* (logic generation) from the *Runtime Stream* (state storage). This design enables two critical capabilities absent in standard JSON agents:

- **State Persistence:** Variables (e.g., Thermostat, Door) are initialized once and retain their state across multiple turns, eliminating the need to hallucinate or re-query context.
- **Control Flow Execution:** The agent generates executable Python code with conditionals (e.g., `if not door_lock.is_locked:`), allowing for precise, context-dependent state transitions rather than blind API execution.

Multi-Turn Scenario: Smart Home "Weekend Party" (Selected Turns)		
Time / Turn	User Query (Intent & Context)	State Evolution & Validation
Turn 3 1:00 PM	"Party prep! Guests arriving soon. Adjust thermostat for comfort, set music to medium, open blinds fully, make lights bright."	<code>validate_party_turn_3</code> <ul style="list-style-type: none"><li>• Music: OFF → 40% (Medium)</li><li>• Blinds: Closed → 100% (Full)</li><li>• Light: Dim → 80% (Bright)</li></ul>
Turn 5 4:00 PM	"Party mode! Full swing now. <b>Turn up the music</b> and make lights <b>very bright</b> . Verify camera is recording."	<code>validate_party_turn_5</code> <ul style="list-style-type: none"><li>• Music: 50% → 60% (Party)</li><li>• Light: 80% → 90% (Very Bright)</li><li>• Camera: Assert status == Recording</li></ul>
Turn 7 7:00 PM	"Evening party. <b>Close blinds completely</b> , set mood lighting... turn up music <b>more</b> ."	<code>validate_party_turn_7</code> <ul style="list-style-type: none"><li>• Blinds: Partial → 0% (Closed)</li><li>• Music: 60% → 70% (Up More)</li><li>• Light: 90% → 60% (Mood)</li></ul>
Turn 10 10:00 PM	"Guests leaving. <b>Lower music more</b> , lock door, turn off bedroom light."	<code>validate_party_turn_10</code> <ul style="list-style-type: none"><li>• Music: 80% → &lt; 60% (Lowered)</li><li>• Door: Unlocked → Locked</li><li>• Bed Light: ON → OFF</li></ul>
Turn 17 Sun 10 AM	"Lazy morning... Finally getting up. Turn on bedroom light, open blinds, raise thermostat."	<code>validate_party_turn_17</code> <ul style="list-style-type: none"><li>• Long-horizon consistency check</li><li>• Thermostat: Eco (18) → Comfort (21)</li><li>• Blinds: Closed → 70% (Open)</li></ul>

**Figure 11: State persistence in long-horizon interactions.** We visualize 5 key moments from the 20-turn `weekend_party` scenario. The agent must maintain a coherent environment state (lighting, temperature, security, audio) over a simulated 24-hour period. Crucially, it handles **relative instructions** (e.g., "turn up music", "lower music more") by tracking the exact discrete levels (e.g., Medium=40, Party=60) defined in the environment schema.

Multi-Turn Scenario: Financial Account "Carol's Debt Paydown" (Numerical Precision)		
Turn	Conditional Query (Logic & Math)	State Calculation & Assertions
T1	"Initialize account... Name 'Carol', <b>Balance 500</b> , Status 'standard', Interest 8% (Loan rate), <b>Loan 2000</b> ."	<code>validate_carol_turn_1</code> <ul style="list-style-type: none"><li>• Balance: 500</li><li>• Loan: 2000</li><li>• Status: 'standard'</li></ul>
T2	"Monthly loan interest due. Apply interest rate (8%) to loan balance and add to debt."	<code>validate_carol_turn_2</code> <ul style="list-style-type: none"><li>• Interest = <math>2000 \times 0.08 = 160</math></li><li>• New Loan = <math>2000 + 160 = 2160</math></li></ul>
T4	"Pay the <b>smaller of</b> 15% of balance or 15% of loan_balance. Subtract from both." (Context: T3 Paycheck +800 → Balance 1300)	<code>validate_carol_turn_4</code> <ul style="list-style-type: none"><li>• [Logic] IF min(1300 × .15, 2160 × .15)</li><li>• Calc: min(195, 324) = 195</li><li>• New Loan = <math>2160 - 195 = 1965</math></li></ul>
T8	"Pay the <b>larger of</b> 40% of balance or 500 toward loan." (Context: Balance grew to 1574 after T7)	<code>validate_carol_turn_8</code> <ul style="list-style-type: none"><li>• [Logic] Compare: <math>1574 \times 0.4 = 629</math> vs 500</li><li>• Action: Pay 629</li><li>• Verify exact integer subtraction.</li></ul>
T14	"Check upgrade: <b>IF</b> loan_balance < balance, upgrade status to 'premium'." (Context: Loan reduced to 1172, Balance 1646)	<code>validate_carol_turn_14</code> <ul style="list-style-type: none"><li>• [Logic] Condition: <math>1172 &lt; 1646</math> (True)</li><li>• Status → 'premium'</li><li>• Triggers T15 bonus paycheck.</li></ul>
T16	" <b>IF</b> balance > loan_balance, pay off entire loan. Otherwise pay 75%..."	<code>validate_carol_turn_16</code> <ul style="list-style-type: none"><li>• [Logic] Action: Payoff Condition Met.</li><li>• Loan → 0.0</li><li>• Balance reduced by remaining debt.</li></ul>

**Figure 12: Numerical precision and state-dependent reasoning.** In the `carol_debt_paydown` scenario, the agent must perform exact integer arithmetic while navigating complex logic gates (e.g., Turn 4's "smaller of", Turn 14's "net worth check"). A single miscalculation in early turns (e.g., T2 interest) would cascade, causing failures in subsequent logic checks (e.g., failing the T16 payoff condition), thus rigorously testing long-horizon numerical stability.

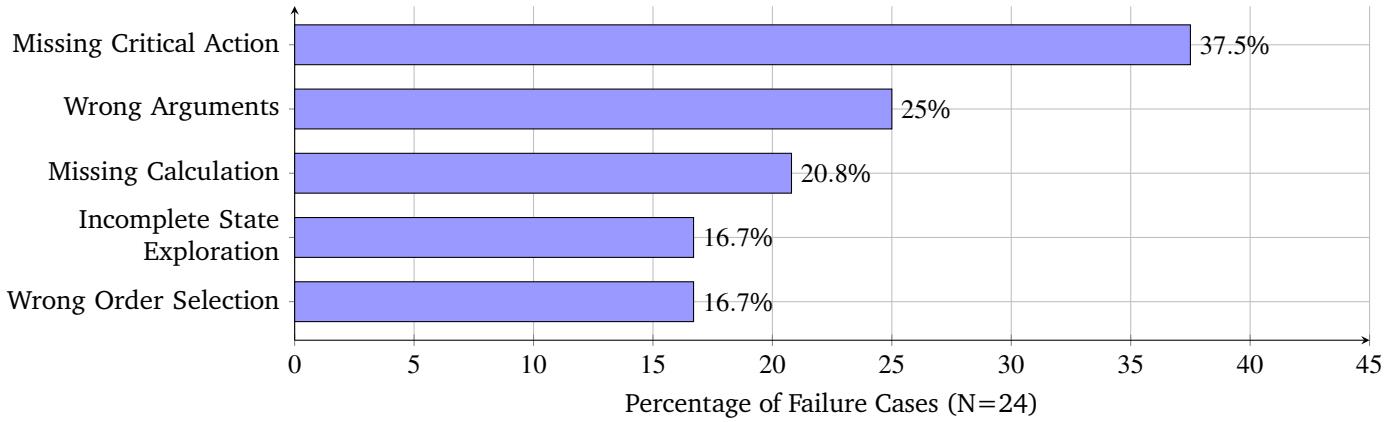


Figure 13: Distribution of failure modes in baseline JSON agent trajectories. Note: Categories are non-exclusive as complex tasks may exhibit multiple failures.

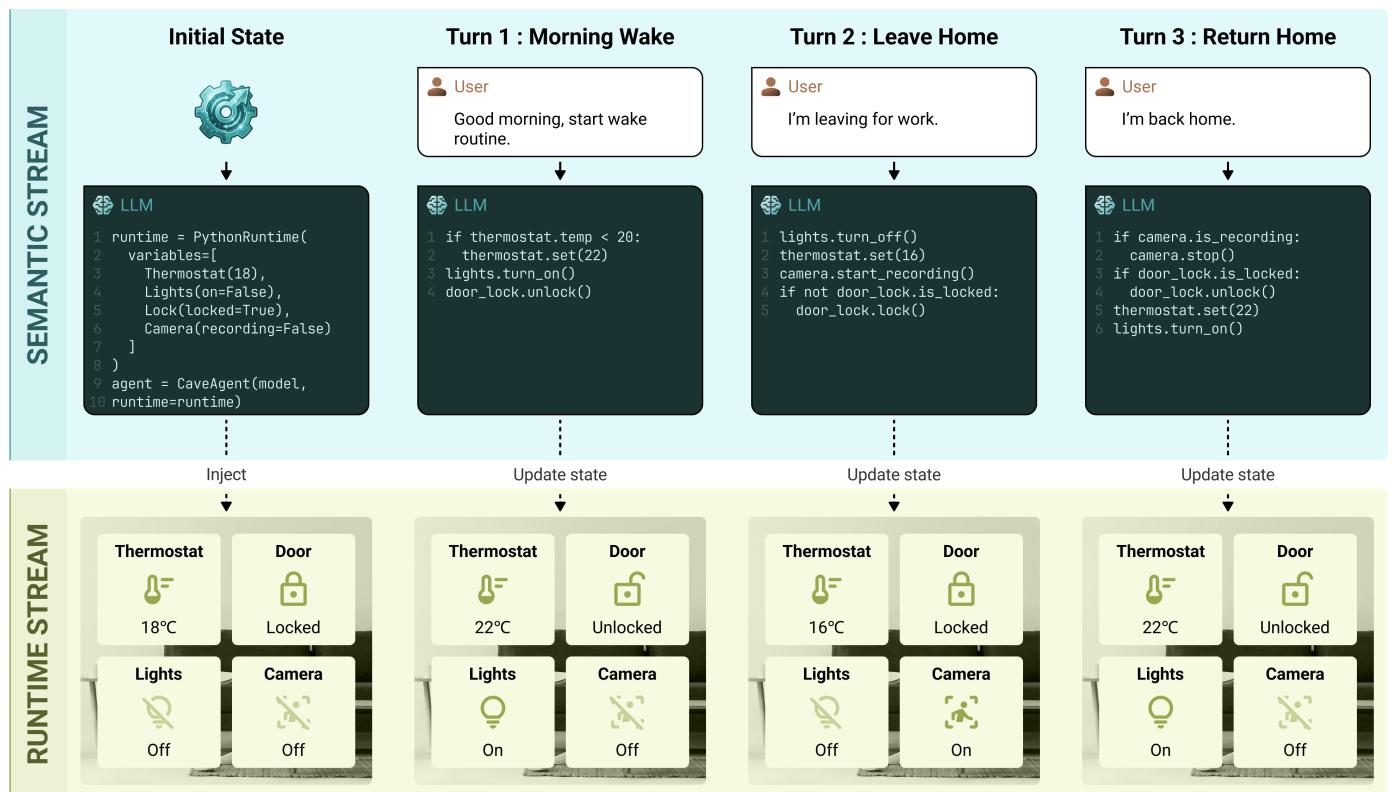


Figure 14: Demonstration of CaveAgent in a toy Smart-home example: the Semantic Stream interact with the Runtime stream via generating code, manipulating the stateful objectives (variables) in the persistent runtime.