# CaveAgent: Transforming LLMs into Stateful Runtime Operators

**Anonymous authors**
**Paper under double-blind review**

## Abstract

LLM-based agents are increasingly capable of complex task execution, yet current agentic systems remain constrained by text-centric paradigms that struggle with long-horizon tasks due to fragile multi-turn dependencies and context drift. We present CaveAgent, a framework that shifts LLM tool use from "LLM-as-Text-Generator" to "LLM-as-Runtime-Operator." CaveAgent introduces a dual-stream architecture: a *semantic stream* for lightweight reasoning and a *runtime stream* backed by a persistent Python environment for stateful execution. Rather than treating the LLM's text context as the primary workspace, CaveAgent elevates the persistent runtime as the central locus. Beyond leveraging code generation to resolve interdependent sub-tasks (e.g., loops, conditionals) in a single step, CaveAgent introduces Stateful Runtime Management: it injects, manipulates, and retrieves complex Python objects (e.g., DataFrames, database connections) that persist across turns, unlike existing code-based approaches that remain text-bound. CaveAgent further provides a runtime-integrated skill management system that extends the Agent Skills open standard, enabling ecosystem interoperability through executable skill injections. This persistence mechanism serves as a high-fidelity external memory that reduces context drift in multi-turn interactions and preserves processed data for downstream applications with less information loss. Evaluations on Tau$^2$-bench and the Berkeley Function Calling Leaderboard (BFCL) across six state-of-the-art LLMs demonstrate consistent improvements in 11 out of 12 settings, with gains up to +13.5% success rate on multi-turn retail tasks. On BFCL, the three open-source models we evaluate all reach 94.0–94.7% under CaveAgent, comparable to closed-source Claude Sonnet 4.5 (94.4%) and Gemini 3 Pro (94.3%) and exceeding GPT-5.1 (89.6%) under their native function-calling protocols; the 30B Qwen3-Coder reaching 94.4% suggests the function-calling protocol is a key performance bottleneck alongside model scale. Token efficiency studies show 28.4% reduction in total token consumption and up to 51% token reduction on data-intensive tasks relative to the best baseline. The accessible runtime state further provides programmatically verifiable feedback, enabling automated evaluation and reward signal generation without human annotation and establishing a structural foundation for future research in Reinforcement Learning with Verifiable Rewards (RLVR).

## 1 Introduction

Large Language Models (LLMs) have demonstrated strong knowledge acquisition and reasoning capabilities across diverse natural language processing tasks. Building on these capabilities, tool-integrated reasoning (TIR) enables LLM agents to interact with external tools and APIs in a multi-turn manner[1] (Lu et al., 2023; Shen et al., 2023; Patil et al., 2024; Qu et al., 2025), expanding their information access and solution space. This has extended LLM agents to various domains, including 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 this progress, 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.,

[1]In this paper, we use tool use and function calling interchangeably.

2023; Achiam et al., 2023). For example, to retrieve stock data, the model must synthesize a JSON string like `{"tool": "get_stock", "params": {"ticker": "AAPL", "date": "today"}}`, requiring exact adherence to syntax and field constraints. This imposes three architectural limitations: **1) Rigid Control Flow:** each turn executes a single tool call (or parallel batch) and serializes output back to context, introducing latency for tasks requiring sequential orchestration (Wu et al., 2024; Shen et al., 2023); **2) Statelessness:** each call is an isolated transaction with no persistent state across turns, so intermediate results must be serialized back into text, causing token overhead for complex data structures and cascading error propagation (Qiao et al., 2023; Kim et al., 2023); and **3) Limited Composability:** JSON schemas express flat function signatures and cannot natively represent loops, conditionals, or variable dependencies, forcing multi-step logic into error-prone multi-turn dialogue (Kim et al., 2023; Wang et al., 2024a).

While recent works attempt to address these issues with code-based tool use (Wang et al., 2024a; Yang et al., 2024), they predominantly adopt a *process-oriented* paradigm where the runtime state remains **internalized and text-bound**. This creates a "textualization bottleneck": variables are accessible to external systems only through text output, requiring serialization into text strings (e.g., printing a DataFrame) to communicate with the user (Wang et al., 2024a; Yao et al., 2022b). This prevents 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 downstream 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, and enabling modular distribution of executable tool capabilities through a portable Agent-Skill standard.*

We present **CaveAgent**[2], an open-source framework that introduces the concept of **Stateful Runtime Management** for LLM agents, shifting code-based tool use from "process-oriented function calling" to persistent "object-oriented state manipulation." CaveAgent operates on a **dual-stream architecture** with two distinct streams: a **semantic stream** for reasoning and a **runtime stream** for state management and code execution. This represents an architectural alternative to the conventional design: whereas existing agents treat the LLM's semantic context as the primary workspace with external tools as auxiliary, CaveAgent elevates the persistent runtime as the primary locus of computation and state, with the semantic stream serving as a lightweight orchestrator that generates code to manipulate it.

By injecting complex data structures (e.g., graphs, DataFrames) directly into the runtime as persistent objects, CaveAgent achieves a form of **context engineering**: the agent manipulates high-fidelity data via concise variable references, decoupling storage from the limited context window. Any intermediate result (e.g., DataFrames, planning trees, or key metadata) can be stored in persistent variables that the agent actively retrieves for later use or downstream applications (code as action, state as memory). This reduces progressive context degradation in multi-turn interactions (Ouyang et al., 2022; Luo et al., 2025), enables context compression, and provides error-free recall through the runtime serving as an external memory.

The persistent environment also enables **few-step resolution of complex logical dependencies** by using code to interact with multiple interdependent tools, allowing the agent to compose workflows (e.g., data filtering followed by analysis) in a few turns rather than through error-prone multi-round function calling (Wang et al., 2024b; Qin et al., 2023). The runtime's transparency makes agent behavior fully verifiable, supporting checks on both intermediate programmatic states and final output objects of any data type, thereby enabling fine-grained reward signals for Reinforcement Learning.

Finally, CaveAgent supports **artifact handoff without information loss** by returning native Python objects rather than text representations, enabling direct use in downstream tasks such as UI rendering, visualization, and structured validation. The runtime can be serialized and reloaded, preserving the agent's complete state across sessions. This transforms the LLM from an isolated text generator into a stateful, interoperable computational component within broader software ecosystems.

[2]Code and data are publicly available at `https://anonymous.4open.science/r/cave-agent-826D`.

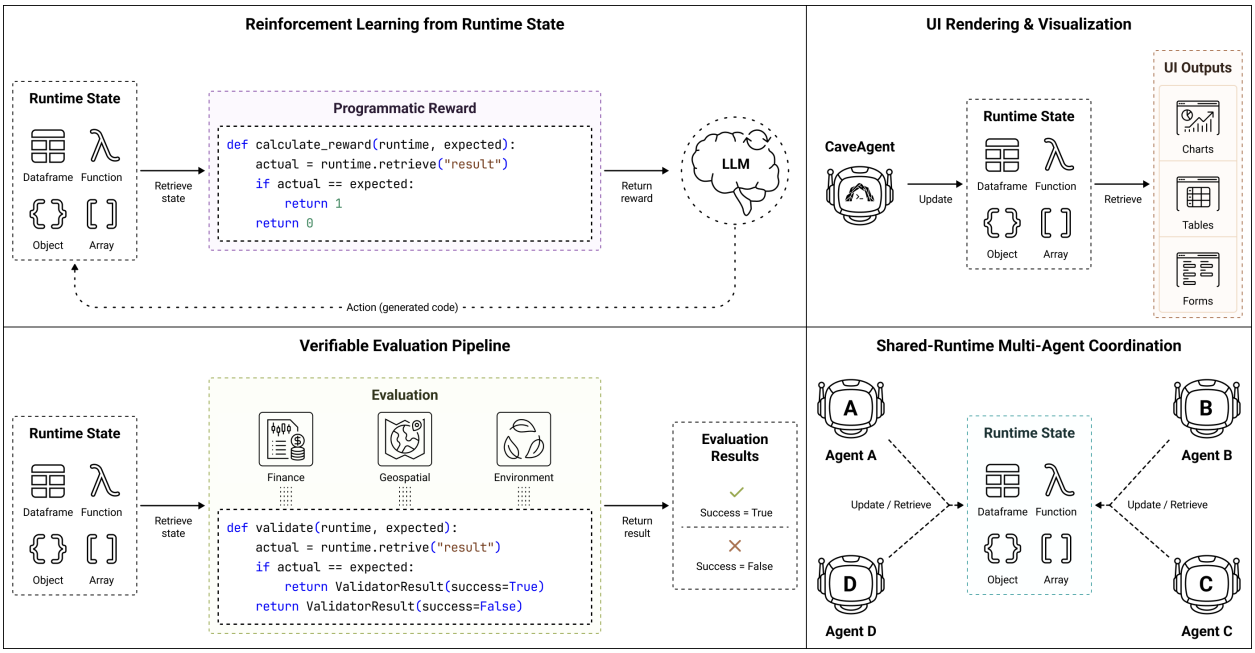



Figure 1: Key Advantages of CaveAgent

The function-calling paradigm in CaveAgent also extends beyond single-agent capabilities to enable **Runtime-Mediated Multi-Agent Coordination**, one of several capabilities summarized in Figure 1. Unlike conventional frameworks where agents coordinate via lossy text message passing (Li et al., 2023; Park et al., 2023), CaveAgent enables agents to interact through direct state manipulation. A supervisor agent can programmatically inject variables into a sub-agent's runtime to alter its environment or task context without ambiguous natural language instructions. Multiple agents can also 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 immediately visible to all peers through direct reference. This enables multi-agent collaboration through state-mediated coordination as an alternative to message passing (qualitative case studies in Appendix E, including a town-simulation walkthrough; rigorous quantitative evaluation is left for future work). We summarize our contributions as follows:

- We introduce **CaveAgent**, a tool-use framework built on **Stateful Runtime Management**. CaveAgent shifts the paradigm from process-oriented function calling to persistent, object-oriented state management. It achieves context compression and context-grounded memory recall by delegating context engineering to a persistent runtime, eliminating the token overhead and precision loss of textual serialization while enabling few-step resolution of logically interdependent tasks. CaveAgent also introduces **runtime-integrated skill management** that extends the Agent Skills open standard: because the architecture treats tools as first-class Python objects, skills can deliver executable artifacts (functions, variables, and type definitions) directly into the runtime upon activation, unifying tool registration and skill distribution into a single mechanism.

- The framework's programmatic inspectability is a *structural* property of the architecture: runtime state is deterministically accessible at any point, providing automated evaluation and fine-grained reward signals for **Reinforcement Learning with Verifiable Rewards (RLVR)** without requiring subjective human annotation.

- We evaluate CaveAgent on standard benchmarks (e.g., Tau$^2$-Bench) and provide case studies across various domains including geospatial analysis (Appendix G.3) and AutoML multi-agent coordination (Appendix E.4). We also provide qualitative case studies suggesting how the same runtime substrate

can support **Stateful Runtime-Mediated Multi-Agent Coordination**, leaving rigorous quantitative evaluation as a direction for future work.

## 2 Background and Related Work

The recent surge of autonomous LLM-driven agents has been comprehensively surveyed in the context of distributed AI for industrial deployment (Piccialli et al., 2025), and concrete domain deployments — such as CDAFlow's stateful clinical decision-making framework (Hou et al., 2026) — underscore a practical demand for agents that can reliably manage state across multi-step workflows. These trends motivate renewed attention to the architectural choices behind state management, memory, and tool orchestration. We review four threads bearing directly on CaveAgent: tool learning, code-based action, context management, and multi-agent coordination.

### 2.1 Tool Learning & Function Calling

The foundational approach to LLM tool use relies on a JSON-centric paradigm. The ReAct framework (Yao et al., 2022b) established the influential thought-action-observation loop, enabling LLMs to interleave reasoning with tool invocation. Building on this, JSON-Schema function calling (Patil et al., 2024; Qin et al., 2023) constrains action generation to structured schemas, formalized by GPT-4 Function Calling and widely adopted by frameworks such as AutoGen (Wu et al., 2024). This schema-based approach has proven remarkably effective in practice, powering the majority of today's deployed agent systems with reliable type checking and standardized interfaces. Constrained decoding methods like xGrammar (Dong et al., 2025) further enforce syntactically valid JSON at low overhead. Despite this success, JSON-based function calling faces inherent architectural trade-offs when applied to complex, multi-step tasks: as a static interchange format, JSON lacks native control flow; its verbose syntax incurs token overhead (Wang et al., 2024a); and each call operates as an isolated transaction, requiring all intermediate state to be serialized back into text (Packer et al., 2023; Wang et al., 2024b).

### 2.2 Code as Action & Programmatic Reasoning

The "Code as Action" paradigm represents a significant advance by using executable Python as a unified medium for reasoning and tool invocation. Wang et al. (2024a) proposed *CodeAct*, which demonstrated that replacing JSON payloads with Python code reduces multi-turn overhead by up to 30% and improves task success rates by 20%. This paradigm leverages the Turing-complete nature of code to express loops, conditionals, and variable dependencies. The paradigm extends to domain-specific reasoning: *ViperGPT* (Surís et al., 2023) composes vision modules into executable subroutines, while *Program of Thoughts* (Chen et al., 2022) and *PAL* (Gao et al., 2023) delegate arithmetic and symbolic logic to a Python interpreter, and Bai et al. (2025) use intelligent agents to enrich the prompts driving LLM code generation. While CodeAct's persistent runtime is a key enabler, its interface remains *text-bound*: intermediate states are communicated to external systems only via `print` output, and external data must be loaded through file I/O (Wang et al., 2024a). This makes it difficult to inject pre-existing Python objects (e.g., in-memory DataFrames, trained models) or retrieve runtime objects for downstream use without serialization loss (Liu et al., 2024; Packer et al., 2023).

### 2.3 Context Management & Stateful Architectures

Packer et al. (2023) introduced *MemGPT*, an OS-inspired virtual context management system with tiered memory for long-horizon tasks. Qiao et al. (2023) proposed *TaskWeaver*, a code-first framework preserving data structures across turns. However, existing approaches rely on RAG or textual summarization, inherently **lossy** methods that strip complex runtime objects of structural integrity and executable properties. CaveAgent uses *Variable Injection* to treat the Python runtime itself as high-fidelity external memory, allowing variables to persist in their native object form without re-tokenization overhead.

### 2.4 Multi-Agent Coordination

Li et al. (2023) proposed *CAMEL* for role-playing cooperation; Qian et al. (2024) introduced *ChatDev* with chat-chain workflows; Hong et al. (2023) developed *MetaGPT* encoding SOPs into prompts; and Saadaoui & Alonso (2025) more recently studied coordinated LLM multi-agent systems for collaborative question–answer generation. Although these frameworks differ in role assignment and orchestration patterns, they share a common substrate: agents communicate via **text-based message passing**, which introduces serialization bottlenecks when complex state must be transferred between agents. A separate, longer-running line of research on intelligent-agent architectures predates the LLM era — for example, fuzzy-BDI agent models for cyber-physical systems (Karaduman et al., 2024) — and offers complementary perspectives on stateful, autonomous reasoning, although it does not directly address the text-serialization issue we focus on here. CaveAgent instead enables *Runtime-Mediated State Flow*: agents collaborate by directly injecting and retrieving variables in a shared runtime, shifting coordination from "communication by talking" to "communication by shared state."

Figure 5 illustrates the evolution of architectural approaches to agentic tool use that motivates our work.

### 2.5 Comparison with Related Stateful Agent Frameworks

Table 1 positions CaveAgent against three closely-related frameworks. CodeAct (Wang et al., 2024a) shares CaveAgent's persistent code-execution kernel, but interaction with the runtime is one-way and text-bound: external data must be loaded via file I/O within generated code, and intermediate state is surfaced through stdout or text-formatted return values rather than as native Python objects. CaveAgent's three distinctive advances over CodeAct are: *(i)* bidirectional `inject()`/`retrieve()` APIs for lossless object exchange at the runtime boundary; *(ii)* an Agent-Skills–compatible[3] packaging layer that delivers executable extensions — not merely text instructions — into the runtime (Section 3.2); and *(iii)* runtime-mediated multi-agent coordination demonstrated qualitatively in Appendix E (rigorous quantitative evaluation is future work). MemGPT (Packer et al., 2023) addresses long-context memory via tiered text summarization — an orthogonal axis to runtime-state management. TaskWeaver (Qiao et al., 2023) preserves data structures across turns and provides a framework-specific plugin format that predates the Agent Skills standard; inter-agent state sharing is also not part of its design. Among the four frameworks, only CaveAgent provides first-class bidirectional object exchange together with Agent Skills compatibility, the architectural combination that underpins the experimental gains reported in Section 4.

Table 1: Capability comparison: CaveAgent vs. closely-related stateful agent frameworks. ✓: supported as a first-class capability; "partial": supported with caveats (see footnotes); ✗: not part of the system's design. CaveAgent's column is shaded for emphasis.

| **Capability** | **CaveAgent** | **CodeAct** | **MemGPT** | **TaskWeaver** |
|---|---|---|---|---|
| Persistent code-execution kernel | ✓ | ✓ | ✗ | ✓ |
| External → runtime variable injection | ✓ | ✗ | ✗ | partial[†] |
| Runtime → external object retrieval | ✓ | ✗ | ✗ | partial[†] |
| Lossless object exchange at runtime boundary | ✓ | ✗ | ✗ | ✗ |
| Agent Skills open-standard support | ✓ | ✗ | ✗ | ✗[‡] |
| Runtime-mediated multi-agent coordination | ✓[§] | ✗ | ✗ | ✗ |

[†] TaskWeaver exposes tools and data to its runtime via plugins, but lacks a first-class API for injecting or retrieving arbitrary external Python objects directly. [‡] TaskWeaver (Nov. 2023) predates the Agent Skills standard (Dec. 2025); its plugin format is framework-specific. [§] Demonstrated qualitatively in Appendix E; rigorous quantitative evaluation is left for future work.

[3] Agent Skills is an open specification for portable skill packaging (Anthropic, 2025); see Section 3.2 for details of CaveAgent's extension.

# 3 CaveAgent: Stateful Runtime Management

## 3.1 Core Methodologies

CaveAgent adopts a **dual-stream architecture** (Figure 2): a *Semantic Stream* for lightweight reasoning, and a *Runtime Stream* for stateful execution. Unlike conventional agents where the LLM's text context serves as the primary workspace and tools are auxiliary services, CaveAgent inverts this relationship: the persistent runtime becomes the central locus of data storage, computation, and state management, while the semantic stream is reduced to a lightweight controller generating code to operate on the runtime.

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:

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

$$\mathcal{S}_t = \text{Exec}(c_t, \mathcal{S}_{t-1}) \qquad \text{(Runtime Stream: Persistent Environment)} \tag{2}$$

where $h_t$ represents the semantic history ("in-prompt context") and $c_t$ is the executable code generated by the agent. The key design choice is the decoupling of $h_t$ and $\mathcal{S}_t$: the semantic stream tracks *intent* and lightweight *reasoning* for code generation, while the runtime stream maintains all *data* and *execution state* via the code generated by the semantic stream.

**The Runtime Stream:** The execution kernel of the runtime stream is a persistent Python kernel (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.
- **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.

The runtime stream can also assign values to new variables during interaction and inject them into the persistent namespace (in-runtime context). This enables large context in complex tasks, such as large DataFrames, graphs, or other data structures, to be managed entirely by the Python runtime stream as stateful variables. Their values are preserved natively in persistent runtime memory without requiring repeated serialization into text, eliminating the risk of hallucination from lossy textual representations.

**Code as Action, State as Memory** The agent stores key information (such as reasoning chains and intermediate data analysis results) as persistent variables in the runtime context, retaining only a lightweight description and reference in its in-prompt context. The runtime thus functions as an external memory, allowing the agent to retrieve stored data as native Python objects, achieving context compression and reducing progressive context degradation in multi-turn interactions. This property addresses persistent challenges in agentic tool use, specifically memory, dynamic decision-making, and long-horizon reasoning (Patil et al., 2025).

A natural question arises: what distinguishes storing intermediate results in runtime variables from persisting them to files? We identify three key differences. First, *type fidelity*: runtime variables preserve native Python object types with full method interfaces (e.g., a DataFrame retains `.groupby()`, `.merge()` operations), whereas file-based storage requires serialization that may lose type information or fail entirely for non-serializable objects such as database connections, trained models with custom layers, or open file handles. Second, *access latency*: runtime variables enable zero-cost retrieval within the same memory space,

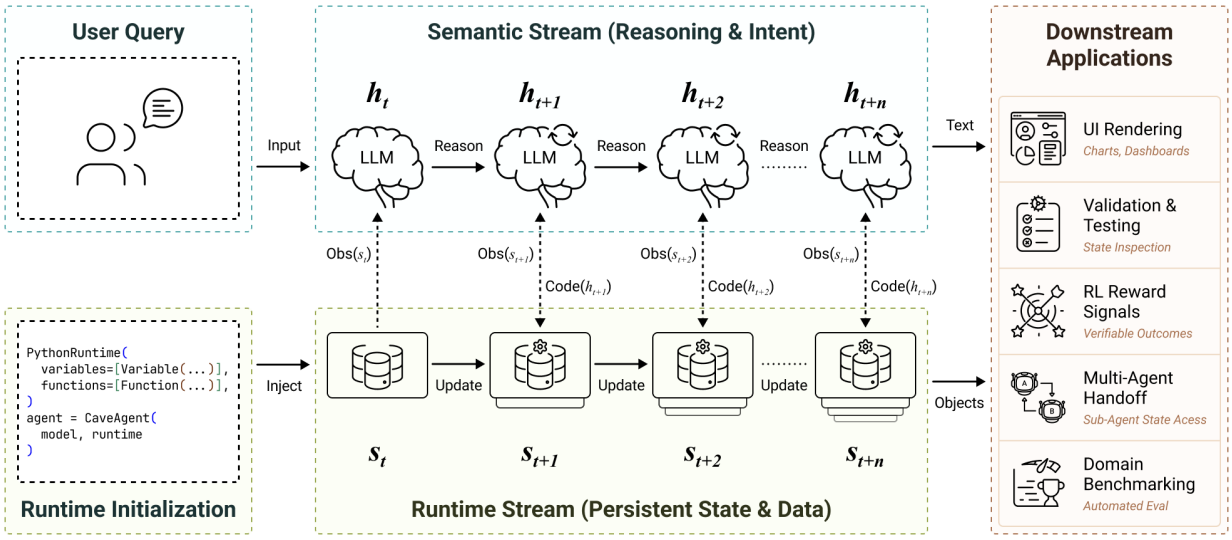



Figure 2: Framework Overview

while file I/O introduces disk latency and requires explicit read/write operations in generated code. Third, *lifecycle management*: runtime variables are automatically scoped to the agent session and garbage-collected appropriately, whereas file-based approaches require explicit cleanup logic to avoid accumulating temporary artifacts. That said, file-based persistence offers advantages for large-scale data exceeding memory capacity and for checkpointing across agent restarts. We note that CaveAgent also supports storing intermediate results in files, but storing them in runtime variables better respects the persistent runtime paradigm, yielding a more unified system.

Programmatic state retrieval enables the extraction of manipulated Python objects for direct use in downstream applications. Unlike conventional agents that produce text outputs requiring parsing and reconstruction, CaveAgent exposes native objects (DataFrames, class instances, arrays) with full type fidelity. This enables UI rendering via direct object binding, RL reward computation through programmatic state inspection, validation via unit test assertions against returned structures, and lossless object passing in multi-agent systems (Appendix E). The agent thus transforms the LLM from an isolated text generator into the operator of a stateful, interoperable computational component.

**The Semantic Stream:** Parallel to the runtime stream, the semantic stream uses the LLM to generate code that manipulates the runtime. 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 guides the agent to interact with the persistent state efficiently, prioritizing concise and relevant information over verbose raw dumps in the in-prompt context $h_{t+1}$.

This split addresses the "Context Explosion" problem: large data remains in $\mathcal{S}_t$ while only high-level reasoning flows through $h_t$, avoiding the token overhead inherent in text-centric architectures. The bidirectional interface for **injecting** and **retrieving** structured objects of any type distinguishes CaveAgent from JSON-based function calling (Lu et al., 2025) and from code-based approaches with internalized runtimes; detailed contrasts appear in Section 2.5. Algorithm 1 (Appendix A) shows the iteration loop; we next describe CaveAgent's core mechanisms.

Table 2: Comparison of Standard Agent Skills vs. CaveAgent Skills.

| **Property** | **Standard Skills** | **CaveAgent Skills** |
|---|---|---|
| Skill format | `SKILL.md` | `SKILL.md` + `injection.py` |
| LLM interaction | Reads text prompts | Operates on injected objects |
| Capability delivery | Textual instructions | Executable runtime artifacts |
| Data flow | Text-in, text-out | Object-in, object-out |
| Tool delivery | Cannot deliver tools | Injects tools into runtime |

#### 3.1.1 Variable and Function Injection

CaveAgent treats Python objects and functions as first-class citizens within the runtime. Each injectable entity is wrapped in a container that automatically extracts metadata: signatures, type hints, and docstrings for functions; names, types, and descriptions for variables. This metadata is aggregated into the system prompt as a lightweight "API reference," while the actual objects are mapped directly into the execution engine's namespace as global symbols. This design enables *Object-Oriented Interaction*: instead of stateless JSON calls (e.g., `tool: "sort", args: {...}`), the model invokes methods on stateful objects directly (e.g., `processor.process(data)`), chaining method calls and manipulating attributes naturally. The agent interacts via executable Python programs with native control flow (loops, conditionals) and stateful data passing, delivering final output as a native Python object rather than a textual approximation.

#### 3.1.2 Dynamic Context Synchronization

The Semantic Stream is "blind" to the Runtime Stream by default. To inspect runtime state, the agent must explicitly generate code (e.g., `print(df.head())`), enforcing an **Active Attention** mechanism that selectively pulls only relevant slices into the token context. To prevent context explosion from verbose outputs, an **Observation Shaping** layer enforces a length constraint: when output exceeds $L_{\max}$, the system returns a structured error prompting the agent to use summary methods. This feedback loop guides efficient interaction with persistent state.

#### 3.1.3 Security Check via Static Analysis

CaveAgent mitigates code execution risks via AST-based static analysis with modular policy rules: *ImportRule* (blocking unauthorized modules), *FunctionRule* (prohibiting dangerous calls like `eval()`), and *AttributeRule* (preventing sandbox bypass). Violations return structured errors to the semantic stream, enabling self-correction without breaking interaction continuity.

### 3.2 Runtime-Integrated Skill Management

CaveAgent extends the Agent Skills open standard (see Section 2.5) by introducing an `injection.py` module alongside the standard `SKILL.md` file. While standard skills provide text-based prompts that guide LLM behavior, CaveAgent skills additionally export **Functions**, **Variables**, and **Type** definitions that are injected directly into the persistent runtime upon activation. The framework employs progressive disclosure: skill metadata (name and description) is loaded at startup for routing decisions, while full instructions and runtime injections are loaded on-demand when the agent invokes `activate_skill()`. This bridges declarative skill definitions with CaveAgent's stateful runtime paradigm: skills deliver executable artifacts into the runtime, not merely textual instructions to the LLM. As shown in Figure 3, domain expertise is packaged as both human-readable instructions for the language model and machine-executable artifacts for the runtime.

**Discussion: From Tool Disclosure to Object Disclosure.** The skill activation pattern (lazy, metadata-driven retrieval of executable artifacts) generalizes beyond tools to any runtime object. Intermediate results stored as persistent variables ("Code as Action, State as Memory") can similarly be discovered via semantic search over their metadata (names, types, descriptions), bringing relevant state into the agent's context only when needed. Unlike retrieval-augmented generation over text chunks, the retrieved items are

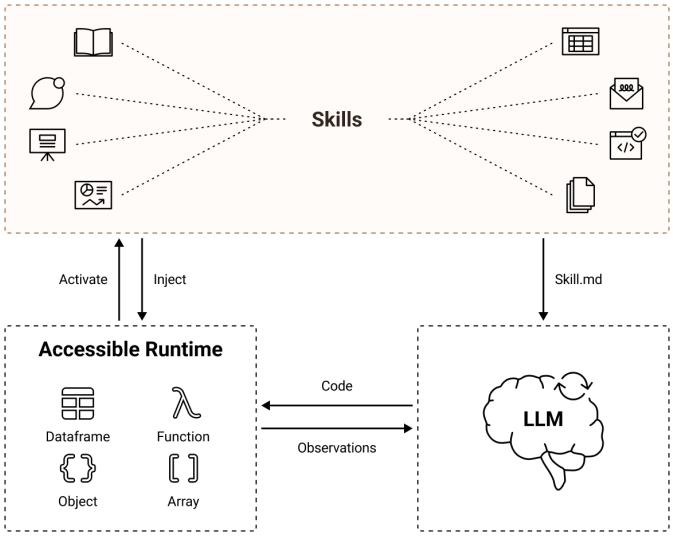



Figure 3: CaveAgent's Skill Management extends the Agent Skills open standard with runtime injection. Standard skills provide text prompts; CaveAgent skills additionally export functions, variables, and types into the persistent runtime.

live Python objects with full type fidelity. This extends progressive disclosure from tool management to general workflow-state management.

## 4 Experiments

In this section, we 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

We evaluate on Tau$^2$-bench (Barres et al., 2025) and BFCL (Patil et al., 2025) using six SOTA LLMs: **DeepSeek-V3.2** (685B MoE), **Qwen3 Coder** (30B MoE), **Kimi K2 0905** (1000B MoE), **Claude Sonnet 4.5**, **GPT-5.1**, and **Gemini 3 Pro**. For each model, we compare its native function-calling mechanism against CaveAgent, where the LLM serves solely as a text generation engine bypassing internal function-calling modules. Per-model sampling configurations are listed in Table 3.

#### 4.1.1 Results on Tau$^2$-bench

Tau$^2$-bench evaluates multi-turn tool use in realistic conversational scenarios (Airline and Retail domains), requiring agents to maintain consistency across turns. We follow the original evaluation protocols with

Table 3: Model details and sampling configuration for the Tau$^2$-bench evaluation. Architecture column reports MoE configuration as *total (active) parameters*; "N/A" indicates that the parameter count or architecture has not been publicly disclosed.

| **Model** | **Reasoning** | **Size** | **Temp.** | **Arch.** | **Temperature notes** |
|---|---|---|---|---|---|
| Qwen3-Coder 30B | None | 30B (3B active) | 0.2 | MoE | For stable code generation |
| Kimi-K2-0905 | None | 1000B (32B active) | 0.6 | MoE | Official recommendation |
| DeepSeek-V3.2 | None | 685B (37B active) | 0.2 | MoE | For stable code generation |
| Claude Sonnet 4.5 | None | N/A | 0.2 | N/A | For stable code generation |
| GPT-5.1 | None | N/A | 1.0 | N/A | Only default value supported |
| Gemini 3 Pro Preview | Low thinking | N/A | 1.0 | N/A | Official recommendation |

Table 4: Performance comparison on the Tau$^2$ Benchmark across different models and domains ($n$=3 runs). Avg. values are reported as mean±std where std is the sample (Bessel-corrected) standard deviation across the three runs. **Bold** indicates that CaveAgent's mean exceeds Function Calling's mean; see the variance analysis paragraph below for which gains exceed run-to-run variance. Cell values are task success rates (%).

| **Model** | **Domain** | **Function Calling** | | | | **CaveAgent** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Run 1 | Run 2 | Run 3 | Avg.↑ | Run 1 | Run 2 | Run 3 | Avg.↑ |
| ***Open Source*** | | | | | | | | | |
| DeepSeek-V3.2 (685B) | Airline | 56.0 | 56.0 | 54.0 | 55.3±1.2 | 62.0 | 60.0 | 58.0 | **60.0**±2.0 (+4.7) |
| | Retail | 79.8 | 77.2 | 74.6 | 77.2±2.6 | 85.1 | 82.5 | 78.1 | **81.9**±3.5 (+4.7) |
| Qwen3-Coder (30B) | Airline | 36.0 | 40.0 | 38.0 | 38.0±2.0 | 36.0 | 42.0 | 44.0 | **40.7**±4.2 (+2.7) |
| | Retail | 41.2 | 43.0 | 39.5 | 41.2±1.8 | 51.8 | 54.4 | 57.9 | **54.7**±3.1 (+13.5) |
| Kimi-K2-0905 (1000B) | Airline | 52.0 | 56.0 | 54.0 | 54.0±2.0 | 58.0 | 54.0 | 54.0 | **55.3**±2.3 (+1.3) |
| | Retail | 62.3 | 60.5 | 59.6 | 60.8±1.4 | 69.3 | 72.8 | 71.9 | **71.3**±1.8 (+10.5) |
| ***Closed Source*** | | | | | | | | | |
| Claude Sonnet 4.5 | Airline | 56.0 | 54.0 | 62.0 | 57.3±4.2 | 56.0 | 52.0 | 62.0 | 56.7±5.0 (-0.7) |
| | Retail | 68.4 | 67.5 | 81.6 | 72.5±7.9 | 73.7 | 75.4 | 80.7 | **76.6**±3.7 (+4.1) |
| GPT-5.1 | Airline | 50.0 | 58.0 | 50.0 | 52.7±4.6 | 58.0 | 56.0 | 54.0 | **56.0**±2.0 (+3.3) |
| | Retail | 64.0 | 66.7 | 66.7 | 65.8±1.6 | 65.8 | 69.3 | 73.6 | **69.6**±3.9 (+3.8) |
| Gemini 3 Pro | Airline | 64.0 | 62.0 | 58.0 | 61.3±3.1 | 68.0 | 68.0 | 68.0 | **68.0**±0.0 (+6.7) |
| | Retail | 72.8 | 72.8 | 66.7 | 70.8±3.5 | 77.2 | 76.3 | 75.4 | **76.3**±0.9 (+5.5) |

DeepSeek V3 as user simulator, testing each model three times per domain. Since CaveAgent executes Python code, we employ runtime instrumentation with wrapper functions to capture and compare function invocations against ground truth, ensuring fair cross-paradigm evaluation.

**Performance Analysis.** The results on Tau$^2$-bench are summarized in Table 4. Key findings include:

*(1).* CaveAgent outperforms JSON-based function calling in 11 out of 12 settings across models from 30B to over 1000B parameters, with consistent improvements for **DeepSeek-V3.2** (**+4.7%**) and **Gemini 3 Pro** (**+6.1%**), showing that offloading state management to a deterministic code runtime improves performance.

*(2).* Gains are amplified in state-intensive *Retail* scenarios, where complex transaction modifications require maintaining state consistency across turns. CaveAgent achieves double-digit gains for Qwen3 and Kimi K2, validating that **Stateful Runtime Management** reduces serialization-induced errors (detailed trajectory analysis in Appendix G.1).

Table 5: BFCL Benchmark Summary (avg. of 3 runs). **Bold** indicates CaveAgent outperforms Function Calling.

| Model | FC Avg.(%) | CaveAgent Avg.(%) | Δ |
|---|---|---|---|
| DeepSeek-V3.2 (685B) | 86.9 | **94.0** | +7.1 |
| DeepSeek-V3.2 (w/o prompt) | 53.1 | **94.0** | +40.9 |
| Qwen3-Coder (30B) | 89.8 | **94.4** | +4.6 |
| Kimi-K2-0905 (1000B) | 89.2 | **94.7** | +5.5 |
| Claude Sonnet 4.5 | 94.4 | 94.4 | 0.0 |
| GPT-5.1 | 89.6 | 88.9 | -0.7 |
| Gemini 3 Pro | 94.3 | 94.3 | 0.0 |

*(3).* The code-specialized **Qwen3-Coder (30B)** exhibits the largest improvement (**+13.5%** in Retail), rivaling larger models. CaveAgent **leverages the inherent coding proficiency of LLMs**, allowing code-centric models to focus on logic generation rather than verbose context tracking.

*(4) Variance analysis.* With $n$=3 runs we report sample standard deviations alongside the per-condition means in Table 4; the combined standard deviation for a difference of means is $\sigma_c = \sqrt{(\sigma_{FC}^2 + \sigma_{CA}^2)/n}$. Several Airline-domain gains lie within or close to run-to-run noise: Claude Sonnet 4.5's small Airline regression ($-0.7$, $-0.2\sigma_c$) is statistically indistinguishable from zero; Kimi-K2 ($+1.3$, $0.8\sigma_c$) and Qwen3-Coder ($+2.7$, $1.0\sigma_c$) Airline gains lie at or within $1\sigma_c$; and GPT-5.1's Airline gain ($+3.3$, $1.2\sigma_c$) is only marginally above. By contrast, Airline gains for **DeepSeek-V3.2** ($+4.7$, $3.5\sigma_c$) and **Gemini 3 Pro** ($+6.7$, $3.8\sigma_c$) clearly exceed $2\sigma_c$. On the Retail domain, gains are markedly more robust overall: **Qwen3-Coder** ($+13.5$, $6.6\sigma_c$), **Kimi-K2** ($+10.5$, $8.0\sigma_c$), **Gemini 3 Pro** ($+5.5$, $2.6\sigma_c$), **DeepSeek-V3.2** ($+4.7$, $1.9\sigma_c$), and **GPT-5.1** ($+3.8$, $1.6\sigma_c$) all exceed $1.5\sigma_c$; the one exception is **Claude Sonnet 4.5** ($+4.1$, $0.8\sigma_c$), where a high Function-Calling baseline variance ($\sigma_{FC}$=7.9, driven by an outlying run-3 score of 81.6 against 67.5 and 68.4) absorbs the gain. The pattern aligns with our overall thesis: CaveAgent's stateful runtime management offers its most robust advantage on multi-turn, state-intensive workflows (Retail), where serialization overhead and context drift accumulate; on the lighter-weight Airline domain, the architectural advantage is smaller and run-to-run variance can dominate at $n$=3. We acknowledge this limitation and note that larger-$n$ replications, beyond the API-cost budget of this study, would tighten the per-condition confidence intervals.

#### 4.1.2 Results on BFCL

To complement Tau$^2$-bench's multi-turn evaluation, we assess atomic function-calling precision on the **Berkeley Function Calling Leaderboard (BFCL) v3** (Patil et al., 2025). We evaluate on the four expert-curated single-turn categories of BFCL v3 — `simple` (400 entries), `multiple` (200), `parallel` (200), and `parallel_multiple` (200), totaling 1,000 question-function-answer pairs of increasing structural complexity. Because BFCL v3 also includes *live*, *multi-turn*, and *multi-step* subsets, we restrict evaluation to the four AST-evaluated categories to keep this benchmark disjoint from our multi-turn evaluation on Tau$^2$-bench. We use Executable Evaluation (functional correctness) by executing generated code and comparing results against ground truth. The summary is shown in Table 5 (detailed per-run results in Appendix Table 9).

**Performance Analysis.** DeepSeek-V3.2 without explicit parallel-execution prompting achieves only 53.1% under the JSON paradigm due to its strong inductive bias toward sequential execution (Liu et al., 2025). CaveAgent achieves **94.0%** without any prompt intervention, as Python code naturally supports parallel execution via independent statements while preserving inter-tool dependency reasoning. The 30B **Qwen3-Coder** with CaveAgent (94.4%) outperforms the much larger **GPT-5.1** (89.6%) and matches **Claude Sonnet 4.5**, demonstrating that CaveAgent leverages the coding proficiency of smaller LLMs. For SOTA models already at near-ceiling performance (Claude Sonnet 4.5, Gemini 3 Pro), gains are negligible since remaining errors stem from ambiguous queries rather than model incapacity. The advantages of our paradigm are most apparent in tasks requiring manipulation of **complex data objects** over long-horizon interactions, which we assess next.

Table 6: 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.

| | **Type Proficiency (%)** | | | | **Multi-Variable (%)** | | | | | | **Multi-Turn (%)** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | 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 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 90.0 | 100 | 95.0 |
| Gemini 3 Pro | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 97.5 | 100 | 98.7 |

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

We design a benchmark targeting dimensions of state manipulation that existing benchmarks do not address, measuring an agent's ability to read, modify, and persist variables across turns. A key design principle is *programmatic validation*: we directly inspect runtime state after execution against ground-truth expectations, rather than parsing text outputs. For each dimension, we curate test cases with linearly dependent queries and initial variable states (see Appendix D). Results are shown in Table 6.

**Type Proficiency** evaluates manipulation of Python primitives, user-defined class instances, and scientific types (DataFrames, ndarrays). Results yield uniformly high scores (96.5%–100%), validating that code-based manipulation of complex types is tractable for current LLMs.

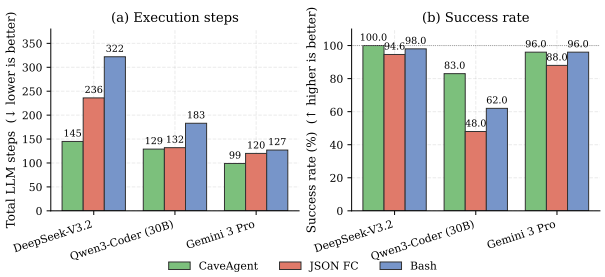
**Multi-Variable** tests how accuracy scales with 5–25 concurrent variables across five tiers (15 evaluation points each). Top models maintain 100% accuracy throughout, demonstrating that concurrent state management scales effectively within CaveAgent's architecture.

**Multi-Turn** assesses state persistence across 40-turn interactions in two scenarios: Smart Home (device state consistency) and Financial Account (numerical precision over multi-step operations). While DeepSeek-V3.2 maintains perfect accuracy, other models exhibit degradation on long-horizon state tracking. The consistently high accuracy across top models validates our thesis: when LLMs interact through code with persistent runtime state, **reliable** and **verifiable** agent behavior becomes achievable.

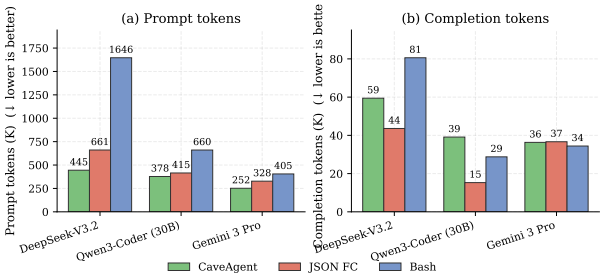
**Discriminability note.** Type Proficiency (96.5–100%) and Multi-Variable (90.9–100%) compress most evaluated models near the ceiling, providing limited model-ranking signal. We retain both dimensions because their purpose is *structural* rather than competitive: a near-100% score validates that CaveAgent's runtime architecture *does* reliably support manipulation of any tested object type and any concurrent variable count up to 25, removing a class of potential architectural failure modes from subsequent claims. Multi-Turn (81.3–98.7% across the non-saturated models) is the most discriminative dimension and serves as the model-ranking signal of this section, stressing the 40-turn state persistence regime where in-context tracking would otherwise accumulate drift. Designing strictly harder Type-Proficiency and Multi-Variable tasks (e.g., adversarial type coercion across boundaries, 50+ concurrent variables under interleaved updates) is a natural extension and is left for future work.

### 4.3 [Q3] Token Efficiency Study

**Setup.** We evaluate token efficiency across three domains (IoT, finance, e-commerce) with logically interdependent tool operations. We extend the original evaluation along two axes: (i) two additional model




(a) Compare Execution steps and task success rates.




(b) Compare Prompt tokens and completion tokens.

Figure 4: Q3 token-efficiency comparison across three models (DeepSeek-V3.2, Qwen3-Coder 30B, Gemini 3 Pro) and three paradigms (CaveAgent, JSON-based function calling, and a bash filesystem agent), focusing on execution steps, success rates, and prompt / completion token consumptions, aggregated over three scenarios (IoT, finance, e-commerce). 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). Aggregate numerical values are also reported in Table 7.

Table 7: Q3 Token Efficiency Study: aggregate results across three domains (IoT, finance, e-commerce) for each (model, paradigm) cell. The two DeepSeek-V3.2 rows for CaveAgent and JSON FC reproduce the original Q3 results; the remaining rows extend the study to two additional model families and to a third paradigm (a bash filesystem agent). Best success rate per model is shown in bold; CaveAgent rows are shaded for readability. $n$=1 per cell.

| Model | Paradigm | Prompt | Compl. | Total | Steps | Success Rate |
|---|---|---|---|---|---|---|
| DeepSeek-V3.2 | **CaveAgent** | 444,679 | 59,440 | 504,119 | 145 | **100%** |
| | JSON FC | 660,588 | 43,600 | 704,188 | 236 | 94.6% |
| | Bash (filesystem) | 1,646,476 | 80,589 | 1,727,065 | 322 | 98% |
| Qwen3-Coder (30B) | **CaveAgent** | 377,599 | 39,111 | 416,710 | 129 | **83%** |
| | JSON FC | 415,204 | 15,283 | 430,487 | 132 | 48% |
| | Bash (filesystem) | 660,378 | 28,760 | 689,138 | 183 | 62% |
| Gemini 3 Pro | **CaveAgent** | 251,774 | 36,327 | 288,101 | 99 | **96%** |
| | JSON FC | 327,574 | 36,659 | 364,233 | 120 | 88% |
| | Bash (filesystem) | 404,860 | 34,419 | 439,279 | 127 | **96%** |

families, Qwen3-Coder (30B) and Gemini 3 Pro,[4] matched to the model labels of Section 4; and (ii) a third paradigm, a single-tool bash agent that persists state through the filesystem. The bash agent exposes a single tool `bash(command: str)` with a per-conversation sandbox; scenario tools are reached through `python -c "import tools; ..."`, and the system prompt explicitly directs filesystem persistence with worked dump → load examples. All runs share the same scenarios, validators, and `max_steps` = 40; sampling temperatures follow the per-model defaults of Table 3 — 1.0 for Gemini 3 Pro (its only supported value) and 0.2 for DeepSeek-V3.2 and Qwen3-Coder (for stable code generation) — with $n$=1 per cell. Aggregate results across the three domains are reported in Table 7 and visualised in Figure 4.

**Results on DeepSeek-V3.2.** CaveAgent achieves 28.4% lower total token consumption (504K vs. 704K JSON FC) while improving success rate from 94.6% to 100%; the gain stems from resolving multiple dependencies in single code executions, reducing steps from 236 to 145 and prompt tokens by 32.7%. CaveAgent

[4]We use `qwen3-coder-30b-a3b-instruct` (the canonical 30B-class identifier) and `gemini-3.1-pro-preview` as a drop-in substitute for the deprecated `gemini-3-pro-preview`.

consumes 36.3% more completion tokens (code is more verbose than JSON), but prompt tokens dominate overall consumption and accumulate across turns. The bash filesystem baseline confirms that the runtime is not over-engineering: filesystem persistence reaches a comparable 98% success rate but at **3.7× the prompt cost** (1.65M vs. 445K) and 2.2× the LLM-call count (322 vs. 145), with the surplus traceable not to filesystem I/O itself but to the per-call system-prompt overhead required to teach the dump/load protocol on every API call (decomposition below).

**Cross-model directional claim.** The qualitative finding — CaveAgent reduces prompt tokens at parity-or-better success rate — reproduces on Qwen3-Coder (30B) and Gemini 3 Pro. Aggregate prompt savings of CaveAgent over JSON-based function calling are 32.7% on DeepSeek-V3.2, 23.1% on Gemini 3 Pro, and 9.1% on Qwen3-Coder. The shrinking aggregate gap on Qwen is an artifact of *premature exit*: JSON FC succeeds on only 48% of Qwen scenarios, terminating before its prompts accumulate, which deflates its denominator. Normalising by the count of *successfully completed* turns reverses the impression: CaveAgent's per-success-turn prompt savings are +37% (DeepSeek), +30% (Gemini), and +48% (Qwen) — the largest savings appear on the weakest model.

**Cross-model robustness.** Across the three models, CaveAgent's success rate spans 17 percentage points (100%, 96%, 83%), against 46.6 pp for JSON-based function calling (94.6%, 88%, 48%) and 36 pp for the bash filesystem baseline (98%, 96%, 62%). Within this directional study (single seed, three models), CaveAgent is therefore the least sensitive of the three paradigms to model substitution; larger-$n$ replication beyond the API-cost budget of this revision would be needed to make this a statistical claim. The runtime-mediated mechanism — variables persist as native objects rather than being re-serialised through the prompt or the filesystem at each turn — degrades gracefully under weaker models, while both message-passing and filesystem-protocol following amplify the weak-model penalty.

**Why bash costs more: a per-call prompt-tax decomposition.** On the Finance domain (where the gap is most visible on Gemini 3 Pro), bash uses 184K prompt tokens versus CaveAgent's 90K. The 94K excess decomposes into two terms: a system-prompt term (calls × instruction size) and a conversation-history term. The system-prompt term contributes 137K to bash and 23K to Cave — bash carries roughly 2,400 tokens of always-resident dump/load instructions per call, against Cave's 580. The conversation-history term is in fact *lower* for bash (47K) than for CaveAgent (67K), since filesystem persistence keeps tool results on disk rather than in the dialogue. The mechanism cost of CaveAgent's runtime is therefore one-time runtime infrastructure that amortises away the per-call instruction overhead a filesystem agent must repeat on every API call.[5]

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

We evaluate three architectures on a data-intensive benchmark comprising 30 tasks across data query, analysis, and visualization using stock market data (Apple and Google, 2020–2025). The setup also serves as a **partial ablation of CaveAgent's two principal architectural components**: *CodeAct Style* disables CaveAgent's variable injection/retrieval API while retaining the persistent code-execution kernel (its success/failure isolates the contribution of inject/retrieve); *JSON-based Function Calling* additionally removes code execution (isolating the joint contribution of code execution and inject/retrieve relative to the JSON-payload baseline). Results are shown in Table 8.

**Results.** On *Data Query*, CaveAgent achieved 100% accuracy (123K tokens) by storing results in runtime variables, while both baselines failed at 80% due to context overflow from serializing large datasets. On *Data Analysis*, CaveAgent and CodeAct both achieved 100% with comparable tokens (∼116–119K), but Function Calling managed only 10% (1.3M tokens) without code execution. On *Visualization*, CaveAgent achieved

[5]The two added models follow the bash system prompt very differently: Qwen3-Coder writes files in 41% of bash commands (literally following the dump/load recommendation), while Gemini 3 Pro writes files in only 8% and instead batches several tool invocations into single Python calls in 67% of bash commands. CaveAgent avoids this strategy choice by construction — variables persist across turns at zero per-call cost — so the same architectural mechanism produces uniform behaviour across model families.

Table 8: Performance comparison across three task categories, framed as a partial ablation of CaveAgent's two principal architectural components: *CodeAct Style* is CaveAgent with the variable injection/retrieval API removed (functionally equivalent to CodeAct's interface), and *JSON-based FC* additionally removes code execution. **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 | **CaveAgent** | **100.0%** (+20%) | 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 | **CaveAgent** | **100.0%** (Tie) | 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 | **CaveAgent** | **90.0%** (+50%) | 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 |

90% (405K tokens) by retrieving chart data from runtime variables; CodeAct reached 40% (1M tokens) and Function Calling 30% (662K tokens). These results demonstrate that decoupling intermediate state from prompt context avoids the token accumulation causing context overflow in conventional architectures, with advantages growing with task complexity and data volume.

**Ablation reading.** Read as an ablation, the table isolates the contribution of CaveAgent's two principal components. Removing the inject/retrieve API (*CodeAct Style*) costs **20 percentage points** on *Data Query* (100% → 80% success) and **50 points** on *Visualization* (90% → 40%): the API is what lets the agent reference large datasets and chart data by handle rather than serializing them into context. Further removing code execution (*JSON-based FC*) is catastrophic on *Data Analysis* (10% success, 1.3M tokens), confirming that code execution is essential when intermediate computation is data-intensive. This is a partial ablation over two components; ablations of the remaining architectural elements (persistent state vs. ephemeral kernel, the Agent-Skills–compatible extension layer, and the multi-agent shared-runtime mode) require separate experimental setups and are left for future work.

**Tracing the abstract's "up to 51%" claim.** Throughout this section, we follow Table 8's convention and compute per-task token reductions relative to the *best (i.e., most token-efficient) baseline* for that task: CodeAct Style on *Data Query* (250K) and *Data Analysis* (119K), and JSON-based Function Calling on *Visualization* (662K). The largest reduction across the three tasks is on *Data Query*, where CaveAgent's 123K total tokens are 51% below CodeAct Style's 250K — this is the "up to 51%" figure quoted in the abstract, and corresponds directly to the `(-51%)` annotation in Table 8. We deliberately do *not* cite CaveAgent's 91% reduction over JSON-based Function Calling on *Data Analysis* (116K vs. 1.3M tokens), even though it is numerically larger: that baseline's 1.3M-token consumption coupled with its 10% task-success rate jointly indicates a *context-overflow failure mode* — repeated re-serialization of large DataFrames into the prompt window — rather than a working baseline whose token cost reflects an actual solution attempt. Reporting a reduction against a failure-mode trajectory would inflate the headline figure, so we exclude this comparison from "up to" claims while still reporting the underlying numbers transparently in Table 8. (By contrast, the lower success rates on *Visualization* reflect task difficulty rather than context overflow — both baselines consume comparable token budgets — so we retain those comparisons.)

## 5 Application Scenarios and Deployment Considerations

While Section 4 evaluates CaveAgent on standard benchmarks, a reader may reasonably ask how the framework translates into deployed AI/ML systems. This section consolidates four representative scenarios drawn from the qualitative case studies in the appendix — each generalized from a single demonstration to a class

of applications — and then discusses deployment considerations not surfaced by benchmark evaluation alone (memory budgeting, tool-wrapping cost, auditability, and selection guidance).

### 5.1 Application Scenarios

**Stateful device control.** The *Smart Home* case study (Appendix G, Figure 16) shows how the dual-stream split substitutes for hand-coded state machines in rule-based home-automation AI/ML systems. Variables such as `Thermostat` and `Door` are initialized once and persist across turns; the agent generates Python conditionals (e.g., `if not door_lock.is_locked:`) rather than blind API calls. The same pattern fits any IoT or supervisory-control AI/ML system where (i) device state must remain consistent across user interactions and (ii) decision logic is more naturally expressed as code than as a fixed rule table — an alternative deployment style to engines such as openHAB or Home Assistant rule chains.

**Scientific decision support over non-serializable inputs.** The *Geospatial Analysis* case study (Appendix G.3) illustrates a pattern we expect to recur in scientific AI/ML systems: the input is a complex non-textual object (here, GeoJSON polygons with high-precision floating-point coordinates) that loses precision or fails entirely when serialized to a JSON payload. CaveAgent injects the polygon as a first-class Python variable and resolves the spatial query in a single turn against domain libraries such as `osmnx`; the same task under JSON-based function calling requires at least five sequential turns and risks coordinate truncation. The same property generalizes to medical imaging (NIfTI volumes), computational geometry / CAD (mesh objects), and bioinformatics (BioPython records) — application classes where ML practitioners face the same serialization wall.

**Hierarchical pipeline orchestration.** The *AutoML Training Loop* (Appendix E.4, Figure 14) exhibits a different reuse pattern: an orchestrator agent injects raw data into a feature-engineering sub-agent's runtime via `inject()`, retrieves the transformed DataFrame, and forwards it to a trainer sub-agent — all without serialization between stages. This is the structure of typical MLOps pipelines, in which adapter code between stages is often the dominant integration cost; CaveAgent's typed bidirectional flow replaces such adapters with native Python object handoff and supports automated convergence checks via inspection of the trainer's runtime metrics.

**Multi-agent shared-world simulation.** The *Town Simulation* (Appendix E, Figure 13) demonstrates peer-to-peer coordination through a shared runtime: when the meta-agent modifies a global `weather` entity, all resident agents observe the change through direct attribute access rather than through inter-agent messaging. This is the structural pattern behind digital-twin and agent-based simulation AI/ML systems (city modeling, supply-chain simulation, epidemiological models) — domains in which message-passing implementations are prone to message-ordering ambiguity on shared world state. Under CaveAgent's single-threaded, turn-based runtime, state updates are linearized through the kernel namespace, which avoids that class of bug at the cost of forgoing concurrent execution between agents.

### 5.2 Deployment Considerations

**Memory budget and cold start.** The persistent IPython kernel carries a baseline memory footprint plus payload proportional to the size of injected and retained objects, and the first turn of each session pays a kernel-boot cost not incurred by stateless JSON function calling. We hypothesize two mitigations as future systems work rather than measured deployment guidance: a kernel pool that amortizes cold-start across requests for high-throughput deployments, and explicit `del` of intermediate variables or session checkpointing for long sessions to limit payload growth. Cold-start, throughput, and production-scale memory profiling were not in the scope of this paper; both mitigations are testable in any specific deployment.

**Tool wrapping at the Python boundary.** As acknowledged in our Limitations, CaveAgent's design couples execution to a Python interpreter; tools exposed only through non-Python interfaces require wrapper creation. In practice the wrapper is small: a tool reachable via REST or gRPC is typically wrapped in a few lines of `requests` or grpc client code, and an in-process Java/Go service can be exposed through a thin

RPC shim. ML practitioners with substantial existing capability in other-language services therefore face a per-tool integration cost rather than a wholesale rewrite.

**Auditability and runtime-state inspection.** CaveAgent's static-analysis security checks (*ImportRule*, *FunctionRule*, *AttributeRule*) filter dangerous code at execution time; complementarily, the runtime namespace is post-hoc inspectable, so any variable created or modified during a session can be examined for compliance review or fault diagnosis. This is a structural property of the architecture rather than a fully tooled deliverable: comprehensive audit-grade tooling (chain-of-custody logs, immutable run histories) remains future work, but the underlying inspectability is what regulated-deployment use cases (healthcare decision support, financial advisory) typically require.

**Selection guidance.** CaveAgent is the appropriate choice when an application combines several of: multi-turn state spanning many turns, non-serializable or precision-sensitive inputs, multi-agent state handoff, or audit trails over intermediate state. Conversely, JSON-based function calling remains preferable for short single-turn queries where memory footprint and cold-start latency dominate, and CodeAct's text-bound interface is sufficient when persistence across sessions is not desired (for example, fully ephemeral execution environments). We recommend treating CaveAgent as one design point among these alternatives rather than a uniform replacement.

### 5.3 Implications for Deployed ML Systems

**Programmatic verifiability and audit trails.** Because runtime state is a deterministically inspectable Python namespace, agent behavior can be evaluated without relying on subjective human annotation — a property already noted as a foundation for reinforcement learning with verifiable rewards. Reframed for deployment, the same property supports compliance review in regulated domains: every variable created or modified during a session can be queried after the fact, enabling fine-grained audit trails over intermediate variable state. Call-level logging under JSON-based function calling provides only the surface trace of tool invocations and their textual outputs and does not natively expose the intermediate runtime state from which those outputs were derived.

**Skill portability via the Agent Skills standard.** CaveAgent's `injection.py` extension to the Agent Skills standard (Section 3.2) allows domain expertise — medical-coding rules, financial-product validators, geospatial analysis recipes — to be packaged as portable, versionable artifacts. This is the modern analogue of the rule packs traditionally distributed for expert-system shells such as CLIPS or JESS, with the difference that the skill itself contributes executable functions and typed objects to the runtime, not only natural-language instructions.

**Lossless object handoff to downstream pipelines.** Agents return native Python objects rather than text approximations, allowing downstream consumers (BI dashboards, training harnesses, automated test rigs) to bind to results directly rather than re-parsing serialized output. This avoids a serialize / deserialize round-trip whose cost, on data-intensive tasks, accounted for substantial overhead under JSON-based function calling (Section 4).

## 6 Conclusion

We present **CaveAgent**, a framework for LLM tool use based on persistent, object-oriented stateful runtime management as an alternative to stateless JSON function calling. CaveAgent enables agents to maintain high-fidelity memory of complex objects and execute sophisticated logic via Python code. By extending the Agent Skills open standard with runtime injection, CaveAgent demonstrates that the persistent runtime paradigm enables a new mode of tool distribution, skills that deliver executable artifacts rather than solely textual instructions, unifying tool registration and skill management into a single portable mechanism. Experiments on Tau$^2$-bench show that this approach consistently outperforms SOTA baselines in multi-turn success rates (11 out of 12 settings) and token efficiency. On BFCL, the three open-source models we evaluate (DeepSeek-V3.2, Qwen3-Coder 30B, and Kimi-K2) all reach 94.0–94.7% under CaveAgent, comparable

to the closed-source Claude Sonnet 4.5 (94.4%) and Gemini 3 Pro (94.3%) and exceeding GPT-5.1 (89.6%) under their native function-calling protocols; the 30B Qwen3-Coder matching Claude Sonnet 4.5 (both at 94.4%) further suggests that for code-capable LLMs the function-calling protocol can be as significant a performance bottleneck as model scale. Beyond performance gains, a key contribution is the **programmatic verifiability** enabled by CaveAgent's architecture: because runtime state is deterministically inspectable, agent behavior can be evaluated automatically without human annotation, establishing a structural foundation for Reinforcement Learning with Verifiable Rewards and runtime-mediated multi-agent coordination. Qualitative case studies are provided in Appendix G.

**Limitations.** *Reliance on a Python runtime.* CaveAgent's design fundamentally couples the agent's execution to a Python interpreter: tools must be Python-callable or wrapped as Python functions, and the persistent state lives in a Python kernel namespace. Tools exposed only through non-Python interfaces (e.g., native binaries, proprietary REST APIs in other languages, microservices written in Java/Go/Rust) require manual wrapper creation, and the runtime state cannot be transparently shared with non-Python downstream systems — a generalizability bound intrinsic to this design choice rather than an implementation gap. Extending the dual-stream pattern to language-agnostic runtimes (e.g., WebAssembly-based execution that admits multi-language tool implementations) is a natural future direction. *Memory.* The persistent runtime consumes memory proportional to the complexity of stored objects; for extremely long sessions with large data artifacts, memory management becomes a concern. *Stateful evaluation.* While agent behavior is in principle programmatically verifiable, designing comprehensive fine-grained benchmarks for stateful evaluation remains future work. *Multi-agent.* The multi-agent coordination capabilities are demonstrated qualitatively; rigorous quantitative evaluation of runtime-mediated multi-agent systems is left for future investigation. *Q3 multi-model and bash-baseline scope.* The two added model families and the bash filesystem paradigm in Table 7 are at $n{=}1$ per cell; we report directional findings rather than statistical significance and recommend $n{\geq}3$ replication for any specific deployment decision. Extending the bash filesystem baseline to the Q4 data-intensive setup is left for future work — Q4's validators inspect runtime state directly and would need parallel file-fallback paths, and Q3's small structured-dict data is in fact the configuration most favourable to a filesystem agent, so the runtime-versus-filesystem gap on Q4 is expected to be larger, not smaller.

## A Pseudo Code

Algorithm 1 shows the general workflow of CaveAgent.

## B What Happens in Semantic Stream

The following sections detail the prompt templates used to instruct the Semantic Stream in CaveAgent. The system prompt is dynamically constructed by combining the Agent Identity, Context Information (functions, variables, types), and 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**

```
{agent_identity}

Current time: {current_time}

You have access to:
```

**Algorithm 1** CaveAgent Interaction Loop

**Require:** Query $q$, Tools $\mathcal{T}$, Max Turns $T_{\max}$

$\mathcal{S}_0 \leftarrow \textsc{InitKernel}()$; $\textsc{Inject}(\mathcal{S}_0, \mathcal{T})$ ▷ Init Runtime Stream
$D \leftarrow \textsc{GenSigs}(\mathcal{T})$; $H_0 \leftarrow \{\text{Sys}(D), \text{User}(q)\}$ ▷ Init Semantic Stream
**for** $t = 1$ to $T_{\max}$ **do**
  **Phase 1 (Reasoning):** $R_t \leftarrow \text{LLM}(H_{t-1})$ ▷ Sample thought & code
  **if** $R_t$ contains code block $c_t$ **then**
    **Phase 2 (Security):** $V \leftarrow \text{AST}\textsc{Check}(c_t, \Pi)$ ▷ Pre-exec validation
    **if** $V \neq \emptyset$ **then**
      $o_t \leftarrow \text{FormatError}(V)$
    **else**
      **Phase 3 (Execution):** $o_t, \mathcal{S}_t \leftarrow \textsc{Run}(\mathcal{S}_{t-1}, c_t)$ ▷ Stateful update
      **Phase 4 (Shaping):** $o_t \leftarrow \textsc{Shape}(o_t, L_{\max})$ ▷ Truncate & format
    **end if**
    $H_t \leftarrow H_{t-1} \cup \{(R_t, o_t)\}$ ▷ Sync observation to history
  **else**
    **return** $R_t$ ▷ Output final answer
  **end if**
**end for**
**return** "Max steps reached"

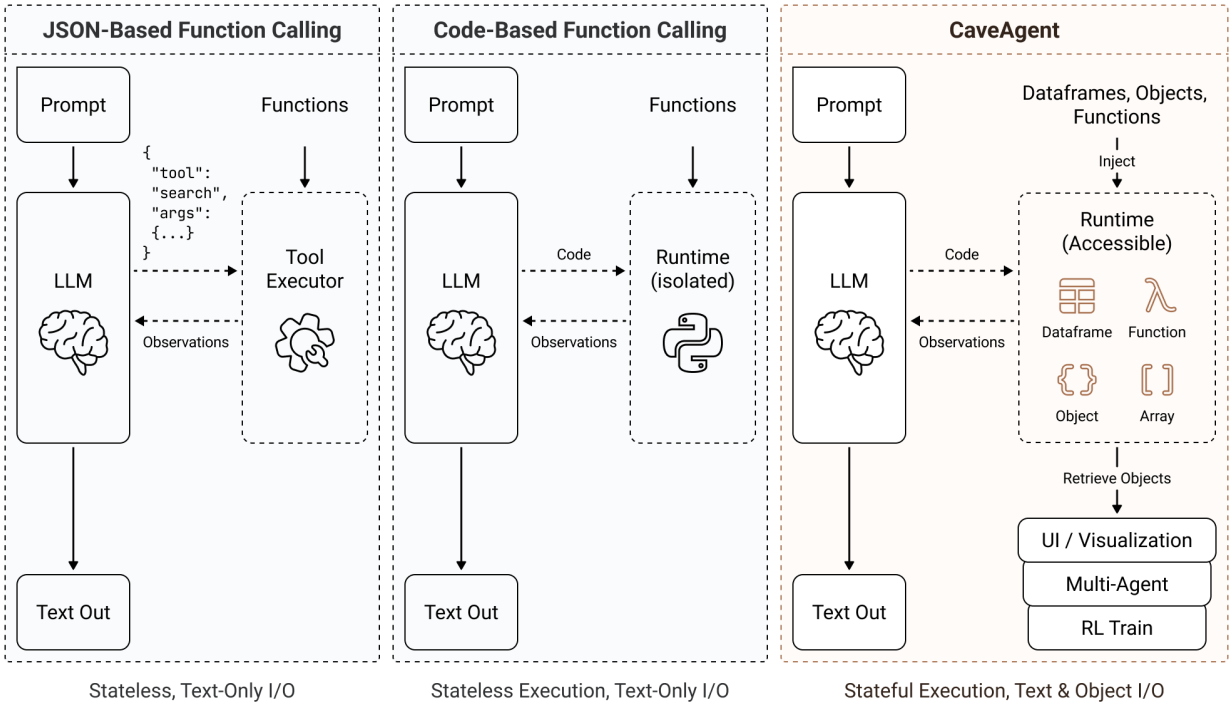



Figure 5: Evolution of Agentic Tool Use

<functions>
`{functions}`
</functions>

<variables>
{variables}
</variables>

<types>
{types}
</types>

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 reasoning and planning, the **Runtime Stream** serves as the execution engine and persistent memory. This stream operates as a dedicated Python kernel where data manipulation, tool invocation, and state transitions occur. The two streams follow a strict chronological topology, synchronized through interleaved exchange of code instructions and execution feedback.

### C.1 Environment Initialization via Injection

The runtime lifecycle begins with **Context Injection**. Before the reasoning cycle starts, 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 DataFrames, graph structures, or class instances, are instantiated within the runtime stream's memory.

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 $S_t \to S_{t+1}$. 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 ($T_1 \to T_2$):** 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 ($T_3 \to T_4$):** 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.

Viewing the runtime stream as a Jupyter notebook with multiple cells, where each cell corresponds to the execution at each time step, helps illustrate how states remain persistent across steps.

**Timeline of Interleaved Execution (Semantic vs. Runtime)**

| Time | Semantic Stream (Reasoning & Code) | Runtime Stream (State & Execution) |
|---|---|---|
| $T_0$ | *(Waiting for Input)* | **[Initialization]**<br>• `df = pd.read_csv("market.csv")` loaded into memory.<br>• `<variables>` metadata sent to Left Stream. |
| $T_1$ | **User:** "Find stocks with volume > 1M."<br>**Agent Thought:** I need to filter the dataframe `df`. I will save the result to a new variable.<br>**Code:**<br>`high_vol = df[df['volume'] > 1e6]`<br>`print(f"Filtered. Rows: {len(high_vol)}")` | *(Waiting for Instruction)* |
| $T_2$ | *(Waiting for Feedback)* | **[Execution]**<br>1. Evaluates filter logic on `df`.<br>2. **New State:** Variable `high_vol` created in RAM.<br>3. **Output:** `"Filtered. Rows: 42"` sent to Left Stream. |
| $T_3$ | **Input:** `"Filtered. Rows: 42"`<br>**Agent Thought:** Now I need to calculate the average price of these high-volume stocks.<br>**Code:**<br>`avg_price = high_vol['price'].mean()`<br>`print(round(avg_price, 2))` | *(Waiting for Instruction)* |
| $T_4$ | *(Waiting for Feedback)* | **[Execution]**<br>1. Accesses persistent object `high_vol`.<br>2. Computes mean.<br>3. **Output:** `154.20` sent to Left Stream. |

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 reloading or serialization, illustrating the efficiency of Stateful Runtime Management.

# 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 ability to perform precise, state-aware manipulation of Python runtime elements. This section tests the agent's working memory across three structural tiers: *Simple Types* (primitive types such as lists, dictionaries, and strings), *Object Types* (custom classes), and *Scientific Types* (high-dimensional complex data). Proficiency in these domains is a prerequisite for complex reasoning tasks.

### D.1.1 Simple Types

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

**Simple Types**

| Turn | User Query (Input) | Immediate Validation (State Assertion) |
|---|---|---|
| ***Case:* `string_split_join`** | | |
| T1 | "Set text to 'a,b,c', split by comma, and rejoin with ' \| ' as separator..." | `validate_str_split`<br>• Assert `text` == "a \| b \| c". |
| T2 | "Sort the parts of text alphabetically while keeping the ' \| ' separator format." | `validate_str_sort`<br>• Assert `text` remains "a \| b \| c".<br>• Checks persistence of structure. |
| T3 | "Reverse the order of parts in text but keep the ' \| ' separator..." | `validate_str_reverse`<br>• Assert `text` == "c \| b \| a". |
| ***Case:* `dict_nested`** | | |
| T1 | "Change the math score to 90 in data['scores']['math']." | `validate_dict_nested_update`<br>• Assert `data['scores']['math']` == 90. |
| T2 | "The student just took a science test. Add a science score of 88 to data['scores']." | `validate_dict_nested_add`<br>• 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." | `validate_dict_increment`<br>• Assert `math` == 95 (90+5).<br>• Assert `science` == 93 (88+5).<br>• Assert `english` == 95 (Initial 90+5). |

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

### D.1.2 Object Types

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

### D.1.3 Scientific Types

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

**Object Types**

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

**Scientific Types**

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

### 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.

**Multi-Variable Management: Tracking 20 Concurrent States (Full Context)**

| Turn | Complex User Query (Full Text) | State Verification (Partial View) |
|---|---|---|
| *Case: `startup_journey` (20 Variables)* | | |
| 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'." | `validate_startup_init`<br>• `employees` → 50<br>• `revenue` → 5,000,000.0<br>• `profit_margin` → 0.1<br>• `public` → False<br>• `stock_price` → 0.0 *(Initial)* |
| 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': 100000000 to financials while keeping existing entries. Add 'support': '555-0200' to contacts while keeping existing entries." | `validate_startup_growth`<br>• `employees` → 150<br>• `depts` → [..., 'HR', 'Finance']<br>• `financials` → +{'valuation': 100M}<br>• Assert `founded_year == 2020` *(Unchanged)* |
| 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." | `validate_startup_ipo`<br>• `public` → True<br>• `stock_price` → 25.0<br>• `company_name` → "TechStart Inc."<br>• `market_cap` → 500,000,000.0 |

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.

### D.3 Multi-turn Cases

These test cases evaluate the agent's capability to process 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 persistent memory of the system's status, as subsequent queries depend on the outcome of previous actions. We categorize these multi-turn benchmarks into two 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.
- 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 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 key contributions for multi-agent coordination; Figure 13 illustrates an example. In this paper, we focus on qualitative analysis and provide case studies to facilitate understanding, leaving rigorous quantitative evaluation for future work. We introduce the high-level ideas below.

**Multi-Turn Scenario: Smart Home "Weekend Party" (Selected Turns)**

| Time / Turn | User Query (Intent & Context) | State Evolution & Validation |
|---|---|---|
| Turn 3<br>**1:00 PM** | "Party prep! Guests arriving soon. Adjust thermostat for comfort, set music to medium, open blinds fully, make lights bright." | `validate_party_turn_3`<br>• Music: OFF → **40% (Medium)**<br>• Blinds: Closed → **100% (Full)**<br>• Light: Dim → **80% (Bright)** |
| Turn 5<br>**4:00 PM** | "Party mode! Full swing now. **Turn up the music** and make lights **very bright**. Verify camera is recording." | `validate_party_turn_5`<br>• Music: 50% → **60% (Party)**<br>• Light: 80% → **90% (Very Bright)**<br>• Camera: Assert status == **Recording** |
| Turn 7<br>**7:00 PM** | "Evening party. **Close blinds completely**, set mood lighting... turn up music **more**." | `validate_party_turn_7`<br>• Blinds: Partial → **0% (Closed)**<br>• Music: 60% → **70% (Up More)**<br>• Light: 90% → **60% (Mood)** |
| Turn 10<br>**10:00 PM** | "Guests leaving. **Lower music more**, lock door, turn off bedroom light." | `validate_party_turn_10`<br>• Music: 80% → **< 60% (Lowered)**<br>• Door: Unlocked → **Locked**<br>• Bed Light: ON → **OFF** |
| Turn 17<br>**Sun 10 AM** | "Lazy morning... Finally getting up. Turn on bedroom light, open blinds, raise thermostat." | `validate_party_turn_17`<br>• *Long-horizon consistency check*<br>• Thermostat: Eco (18) → **Comfort (21)**<br>• Blinds: Closed → **70% (Open)** |

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', **Balance 500**, Status 'standard', Interest 8% (Loan rate), **Loan 2000**." | `validate_carol_turn_1`<br>• Balance: 500<br>• Loan: 2000<br>• Status: 'standard' |
| T2 | "Monthly loan interest due. Apply interest rate (8%) to loan balance and add to debt." | `validate_carol_turn_2`<br>• Interest = $2000 \times 0.08 = 160$<br>• New Loan = $2000 + 160 =$ `2160` |
| T4 | "Pay the **smaller of** 15% of balance or 15% of loan_balance. Subtract from both."<br>*(Context: T3 Paycheck +800 → Balance 1300)* | `validate_carol_turn_4`<br>• **[Logic]** *IF* $\min(1300 \times .15, 2160 \times .15)$<br>• Calc: $\min(195, 324) = 195$<br>• New Loan = $2160 - 195 =$ `1965` |
| T8 | "Pay the **larger of** 40% of balance or 500 toward loan."<br>*(Context: Balance grew to 1574 after T7)* | `validate_carol_turn_8`<br>• **[Logic]** *Compare:* $1574 \times 0.4$ (629) vs 500<br>• Action: Pay `629`<br>• Verify exact integer subtraction. |
| T14 | "Check upgrade: **IF** loan_balance < balance, upgrade status to 'premium'."<br>*(Context: Loan reduced to 1172, Balance 1646)* | `validate_carol_turn_14`<br>• **[Logic]** *Condition:* $1172 < 1646$ (True)<br>• Status → **'premium'**<br>• *Triggers T15 bonus paycheck.* |
| T16 | "**IF** balance > loan_balance, pay off entire loan. Otherwise pay 75%..." | `validate_carol_turn_16`<br>• **[Logic]** *Action:* Payoff Condition Met.<br>• Loan → `0.0`<br>• Balance reduced by remaining debt. |

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.

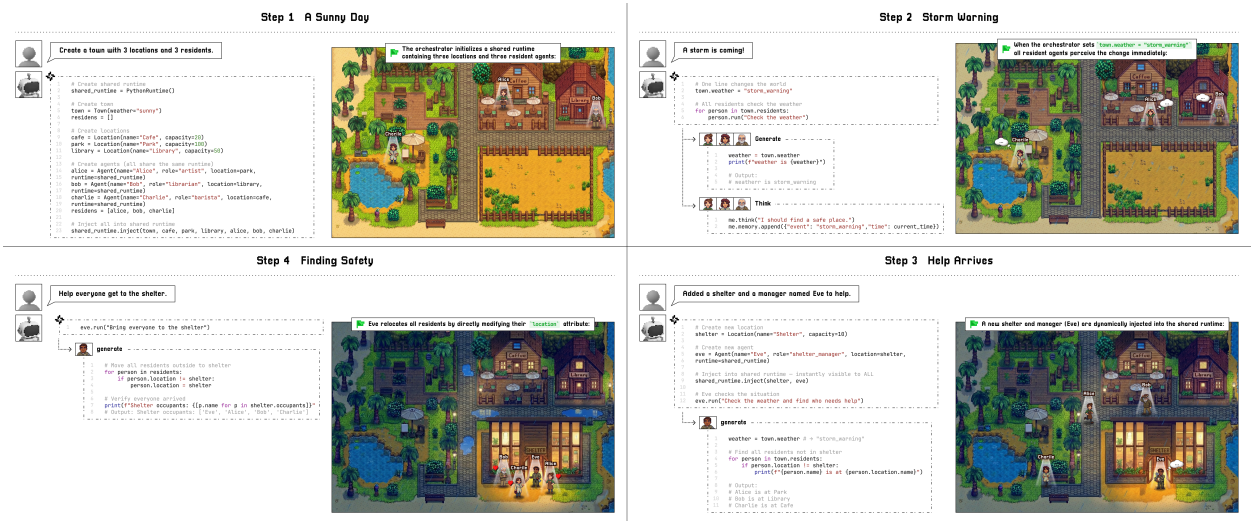



Figure 13: Town Simulation: a toy example for Stateful Runtime-Mediated Multi-Agent Collaboration. Multiple resident agents share a single Python runtime; when the meta-agent modifies a global object (e.g., the weather entity), all residents observe the change through direct reference rather than message passing.

### E.1 Meta-Agent Runtime Control

Sub-agents are injected as first-class objects into a 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 collaborative manipulation of a unified world model with low coordination overhead.

**How the 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, enabling automated validation of inter-agent handoffs and integration with downstream pipelines.

### E.4 AutoML Training Loop

We demonstrate CaveAgent's hierarchical agent coordination through an AutoML training loop where an orchestrator agent programmatically manages sub-agent runtimes (Figure 14). The orchestrator injects

raw data into a feature engineering agent's runtime via `inject()`, triggers execution, then retrieves the transformed DataFrame via `retrieve()` and injects it into a trainer agent's runtime, all as native Python objects without serialization. After training, the orchestrator extracts evaluation metrics directly from the trainer's runtime, validates against target requirements via a `check_requirements()` function, and injects performance feedback back into both sub-agents for the next iteration. Crucially, sub-agents themselves are injected as variables into the orchestrator's runtime, enabling the orchestrator to dynamically access and manipulate their internal states through generated code. This iterative refinement loop continues until programmatic convergence criteria are met, demonstrating CaveAgent's unique capability for hierarchical multi-agent coordination with typed, bidirectional state flow and automated convergence verification.

## F Detailed BFCL Benchmark Results

Table 9 shows the detailed per-run results of the BFCL benchmark.

Table 9: 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. = Simple Function, Mult. = Multiple Function, Para. = Parallel Function, P-M. = Parallel Multiple Function, Ov. = Overall.

| **Model** | **Run** | **Function Calling** | | | | | | **CaveAgent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Simp. | Mult. | Para. | P-M. | Ov. | **Avg.(%)** | Simp. | Mult. | Para. | P-M. | Ov. | **Avg.(%)** |
| ***Open Source*** | | | | | | | | | | | | | |
| 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 | **94.0** (+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 | **94.0** (+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 | **94.4** (+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 | **94.7** (+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 | |
| ***Closed Source*** | | | | | | | | | | | | | |
| 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 | **94.4** (∼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 | **94.3** (∼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 | |

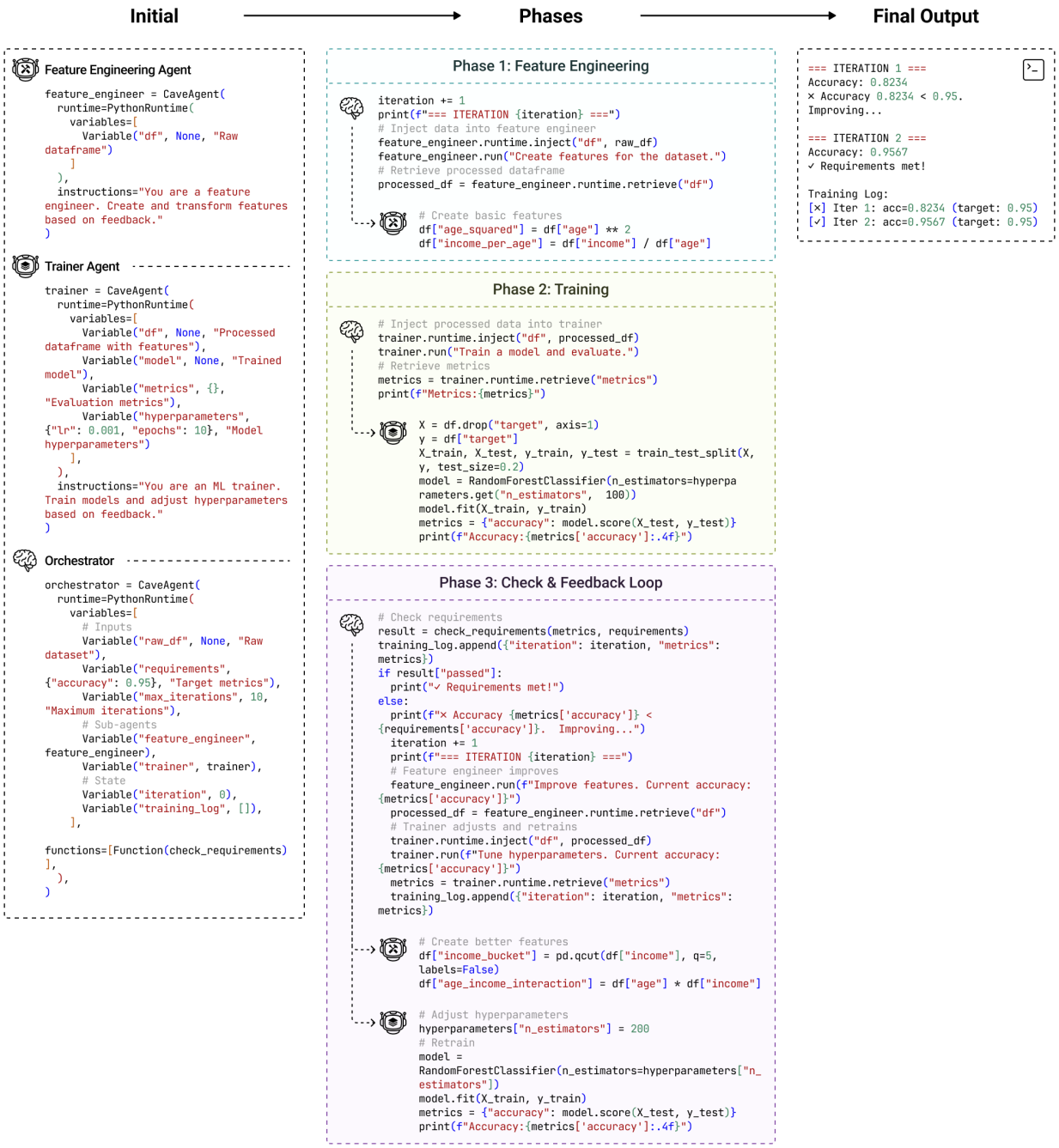



Figure 14: Multi-agent AutoML pipeline: an orchestrator coordinates feature engineering and training sub-agents through runtime variable injection. Native Python objects (DataFrames, trained models) flow losslessly between agents across iterative refinement cycles.

# G Features

## G.1 Case Analysis in Tau$^2$-bench

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

### G.1.1 Failure Taxonomy of the Baseline

Baseline failures were categorized into five distinct patterns (Figure 15). 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.

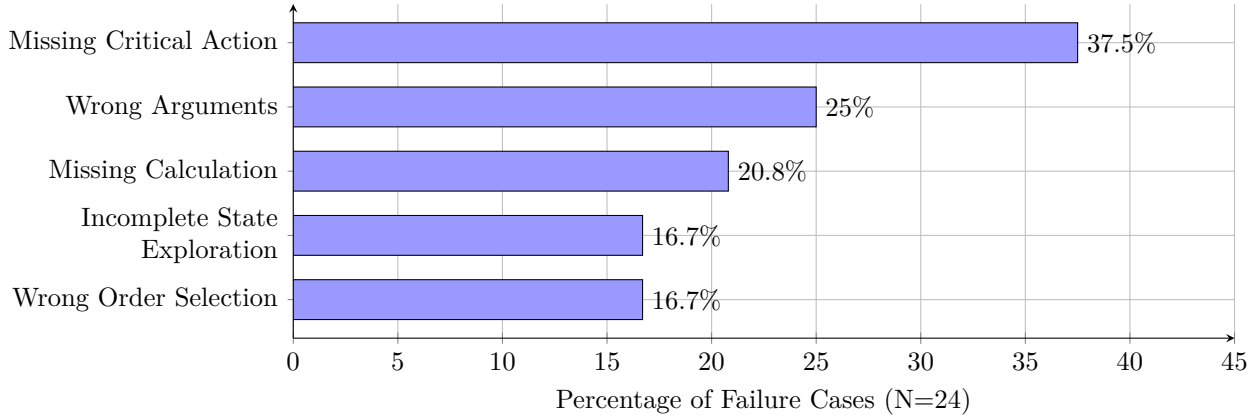



Figure 15: Distribution of failure modes in baseline JSON agent trajectories. Note: Categories are non-exclusive as complex tasks may exhibit multiple failures.

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

The analysis reveals that CaveAgent's improvements stem 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.

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.

### G.2 Smart Home

Figure 16 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 key 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.

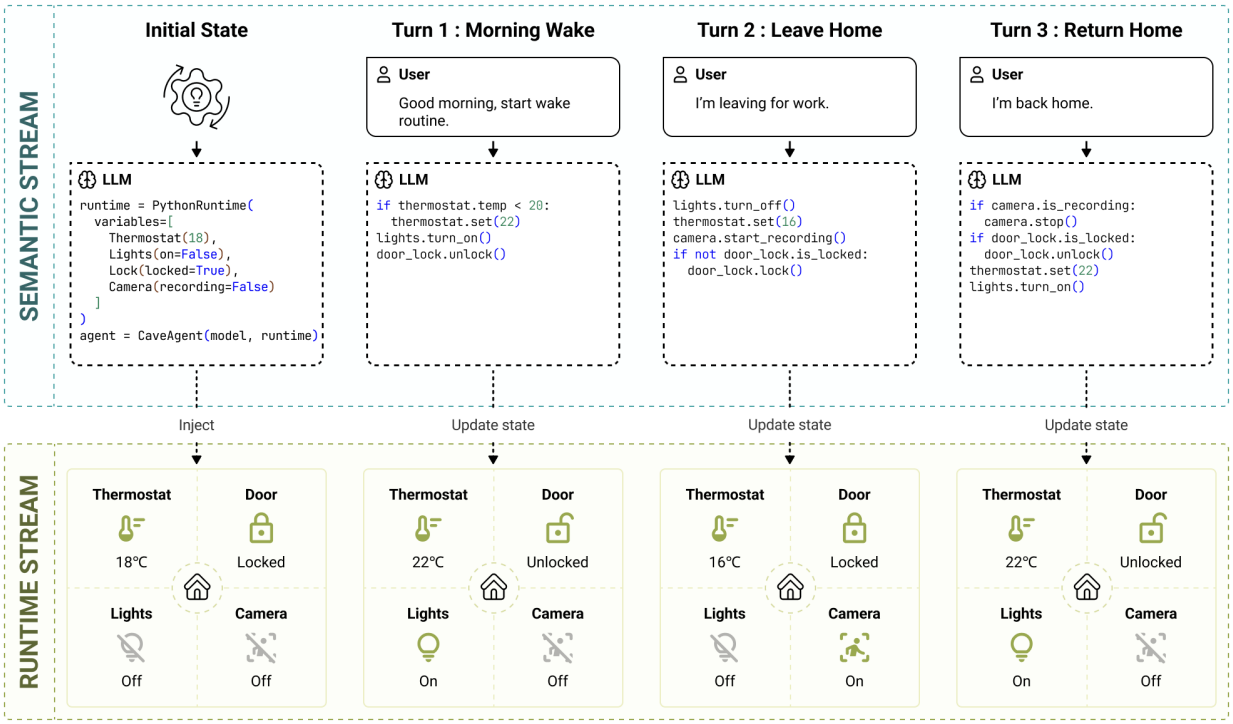



Figure 16: Demonstration of CaveAgent in a smart-home example: the Semantic Stream interacts with the Runtime Stream by generating code that manipulates stateful objects (variables) in the persistent runtime.

### G.3 Geospatial Analysis

To illustrate CaveAgent's advantages in domain-specific applications, we consider an urban planning scenario in which a user draws two arbitrary regions on an interactive map and queries the differences in population density and urban development between them. The interactive map converts user-drawn regions into

GeoJSON polygon objects, variable-length coordinate arrays with high-precision floating-point pairs, which are injected directly into CaveAgent's persistent runtime as first-class Python variables. Upon receiving the natural language query, the LLM generates a single code block that references these injected geometries to perform zonal statistics against WorldPop raster data and extract land use features from OpenStreetMap via `osmnx`, chaining multiple domain-specific operations through native variable passing. Under the conventional JSON function calling paradigm, the same task would require at least five sequential LLM turns, querying population and land use statistics separately for each region before synthesizing text-serialized results, while also confronting the challenge of encoding complex polygon geometries as JSON string parameters, which risks truncation and introduces serialization overhead. CaveAgent resolves the entire query in a single turn with lossless data flow: geometries maintain full numerical precision, intermediate results (e.g., GeoDataFrames) persist as manipulable runtime objects, and the LLM synthesizes the final response from deterministic execution output (Figure 17). This case study demonstrates CaveAgent's suitability for scientific and analytical domains where computation involves complex non-serializable data structures and precision-sensitive results.

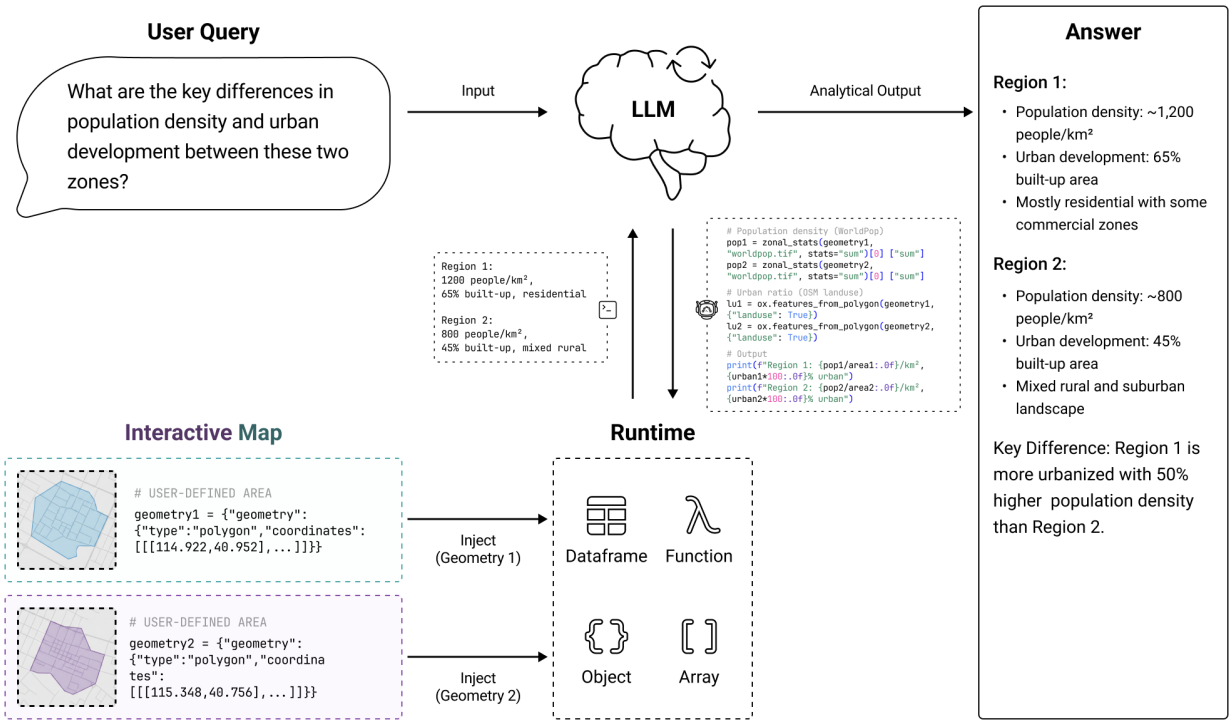



Figure 17: Geospatial Analysis: a user draws two regions on an interactive map. CaveAgent injects the GeoJSON polygons as Python variables and resolves the entire spatial query in a single turn with lossless data flow.

## Broader Impact Statement

CaveAgent enables LLM agents to execute arbitrary Python code in a persistent runtime. While we apply static-analysis security checks (*ImportRule*, *FunctionRule*, *AttributeRule*) and recommend sandboxing for deployment, code-execution agents inherit the dual-use risks of their underlying LLM (e.g., generation of harmful or exploit code) and additionally expose process-level system risks not present in JSON-only agents. Deployers should pair the runtime with container-level isolation, network egress controls, and resource quotas before exposing the framework to untrusted inputs. The qualitative multi-agent results raise no incremental societal risks beyond those of single-agent CaveAgent.

## Reproducibility Statement

Code, the bash-baseline agent, and the matplotlib scripts that regenerate Figure 4 are publicly available at `https://anonymous.4open.science/r/cave-agent-826D`. Per-model sampling configurations (temperatures, `max_steps`) are listed in Table 3; n=1 disclosure for the Q3 multi-model and bash-baseline cells is repeated in the Limitations paragraph of Section 6. Tau$^2$-bench scenarios and the BFCL v3 categories used here are publicly distributed by their original authors.

## 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.

Anthropic. Agent Skills: An open specification for portable skill packaging. `https://agentskills.io`, 2025. Open specification, released December 2025; adopted across AI coding-agent ecosystems including Claude Code, OpenAI Codex, Microsoft VS Code, Google Gemini CLI, and Cursor.

Xingyuan Bai, Shaobin Huang, Chi Wei, and Rui Wang. Collaboration between intelligent agents and large language models: A novel approach for enhancing code generation capability. *Expert Systems with Applications*, 269:126357, 2025. doi: 10.1016/j.eswa.2024.126357.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. $\tau^2$-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

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.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems*, 7, 2025.

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, et al. PaLM-E: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 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*, pp. 10764–10799. PMLR, 2023.

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

Ruihui Hou, Dongge Xue, Hongli Sun, Ping He, Weiyan Zhang, and Tong Ruan. CDAFlow: Enhancing LLM clinical decision-making through agentic workflow. *Expert Systems with Applications*, 316:131806, 2026. doi: 10.1016/j.eswa.2026.131806.

Burak Karaduman, Baris Tekin Tezel, and Moharram Challenger. On the impact of fuzzy-logic based BDI agent model for cyber-physical systems. *Expert Systems with Applications*, 238:122265, 2024. doi: 10.1016/j.eswa.2023.122265.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36:39648–39677, 2023.

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

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.

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.

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. *arXiv preprint arXiv:2310.08560*, 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*, pp. 1–22, 2023.

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.

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 *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

Francesco Piccialli, Diletta Chiaro, Sundas Sarwar, Donato Cerciello, Pian Qi, and Valeria Mele. AgentAI: A comprehensive survey on autonomous agents in distributed AI for industry 4.0. *Expert Systems with Applications*, 291:128404, 2025. doi: 10.1016/j.eswa.2025.128404.

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)*, pp. 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.

Sami Saadaoui and Eduardo Alonso. Coordinated LLM multi-agent systems for collaborative question-answer generation. *Knowledge-Based Systems*, 330:114627, 2025. doi: 10.1016/j.knosys.2025.114627.

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*, pp. 11888–11898, 2023.

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*, 2024a.

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. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024b.

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.

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.

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*, pp. 2165–2183. PMLR, 2023.