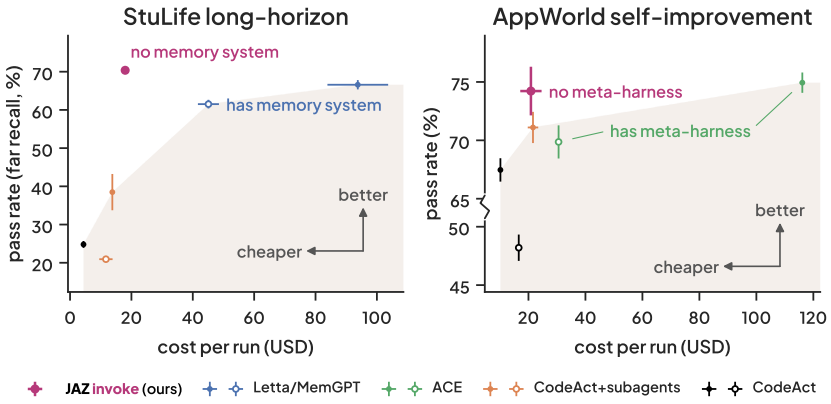
# Harness as a Language: A Minimalist Agent Framework With Maximal Expressivity

**Zhening Li**[1] **Joshua Liu**[*1] **Mateja Vukelic**[*1]
**Nicole Shen**[1] **Supriya Lall**[1] **Amitayush Thakur**[2]
**Alex Zhang**[1] **Omar Khattab**[1] **Jonathan Light**[2] **Armando Solar-Lezama**[1]
[1]MIT CSAIL [2]Independent Researcher




## Abstract

Modern language-model agents are built around the *agent loop*: the LLM is placed in an environment exposing a set of tools, and the LLM has full control over the workflow by alternating between tool calls and observing their output. However, certain capabilities such as long-term memory and self-improvement currently require specialized systems beyond the agent loop itself. We built an LLM agent framework, JAZ, to explore the extent to which a minimal harness that is little more than the *agent loop itself* can accomplish tasks these specialized systems are built for. JAZ exposes a single LLM-based primitive `invoke` and provides a set of built-in hooks that allow the programmer to apply constraints and perform monitoring. Generalizing existing code-mode agent loops, `invoke` is the simplest loop that satisfies two defining properties: (1) the LLM can write arbitrary executable code that can include recursive `invoke`; (2) everything visible to the LLM — all inputs to `invoke` as well as its interaction history with the code environment — are variables in the code environment. We motivate our design from first principles, viewing `invoke` as a language primitive representing a function whose implementation is provided at runtime by an LLM every time it is called. To validate the design of our core `invoke` primitive, we evaluate `invoke` — with only prompting, no manually designed tools, harness, or external systems (e.g., memory or the file system) — on workflows traditionally implemented through specialized harnesses. On long-horizon workflows requiring recall far beyond the context window, JAZ `invoke` outperforms Letta/MemGPT at a fraction of its cost on the recall-heavy portion of StuLife. On continual self-improvement, JAZ `invoke` matches ACE at a fraction of its cost on AppWorld. The JAZ framework source code is located at `https://github.com/jaz-lang/jaz` and our evaluation code is located at `https://github.com/jaz-lang/jaz-evals`.

[*]Equal contribution

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: Towards Test-Time Continual Learning Agents (TTCL).

# 1 Introduction

Language model agents are typically built around a core *agent loop* in which the model repeatedly interacts with an environment through actions and observations. Increasingly expressive agent loop paradigms have been proposed over time. Earlier ReAct [1] gave the model full freedom to decide which action to take at every step of a workflow. CodeAct [2] generalized it by providing the model the expressivity to orchestrate workflows with executable code in a REPL, and typical implementations [3, 4] also allow the user to provide Python objects the agent can interact with in its REPL. Increasing expressivity has unlocked new capabilities. Most notably, the RLM [5] demonstrated that CodeAct+subagents, without any external tools or harness, could outperform LLMs on long-context tasks by providing the input to the agent as a Python object it can manipulate in its REPL. However, agentic capabilities such as long-horizon memory and self-improvement remain implemented through specialized external harnesses layered on top of these loops.

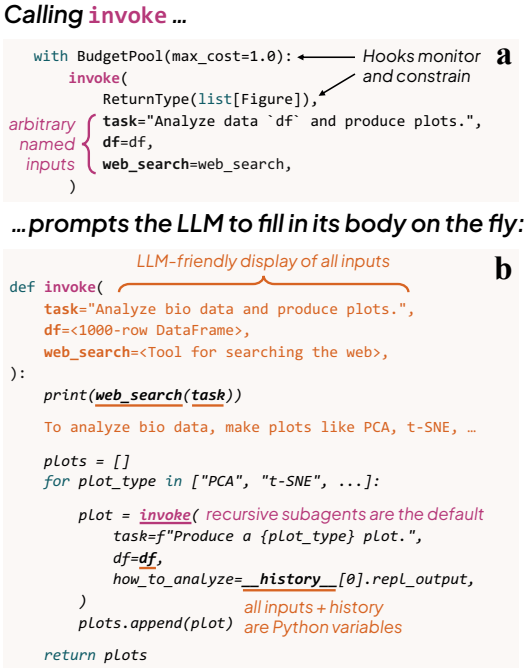



Figure 1: **In JAZ, `invoke` acts as a function whose implementation is provided each time it's called — by the LLM in a Python REPL. Everything — all named inputs to `invoke` and the REPL history itself — are variables available in the REPL.**

We study whether a minimal harness that is little more than the *agent loop itself* can exhibit these capabilities. For this, we built an LLM agent framework, JAZ, whose core primitive `invoke` is a minimal language-level abstraction that generalizes CodeAct [2] and RLMs [5] (Figure 1). First, the model writes arbitrary executable code in this `invoke`-augmented language, allowing tools and recursive subagents to be orchestrated through ordinary program control flow. In other words, like RLMs, recursive subagents are the default. Second, *everything* visible to the model is also a variable in the code environment. While modern implementations of CodeAct and RLMs let the user pass data and tools into the agent's REPL, `invoke` additionally makes the user prompt and REPL history variables in the REPL.

On top of the `invoke` primitive, JAZ provides utilities that allow a user to monitor and constrain agent behavior. Observability hooks log agent trajectories, validation hooks validate return conditions, and resource constraint hooks impose budgets on cost, iterations and context window usage. Although hooks are not essential to the core JAZ primitive, they allow users of the JAZ framework to observe and exercise control over their agent.

We show that JAZ, with little more than pure prompting, can realize agentic workflows traditionally implemented through specialized harnesses. On long-horizon tasks requiring recall far beyond the model's context window, JAZ `invoke` is prompted to delegate to a subagent whenever its context runs out, while keeping a reference to the full conversation history. With GPT-5.4 nano, it achieves 70% on the recall-heavy subset of StuLife [6], outperforming both CodeAct+subagents (39%) and the specialized long-horizon harness Letta/MemGPT [7] (62%). On AppWorld [8], JAZ `invoke` self-improves across the sequence of test tasks by iteratively updating its prompt and tools inside its REPL based on test feedback. With a GPT-5.4 top-level `invoke` and GPT-5.4 nano sub-invokes, it achieves 74%, outperforming CodeAct (68%), CodeAct+subagents (71%), and a specialized self-improvement harness, ACE [9] (70%). Together, these results show that prompting a minimal agent loop is sufficient to elicit long-horizon and self-improvement capabilities, rather than requiring specialized external harness components.

Our main contributions are:

- We introduce and formalize `invoke`, an LLM-based programming language construct that represents a function whose implementation is provided by the LLM at call time.

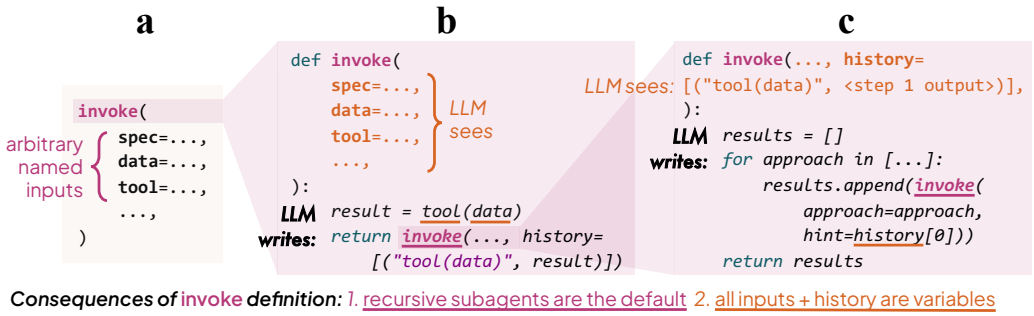



Figure 2: In the functional version of `invoke`, the LLM writes a complete implementation for every invocation. A closed agent loop arises from tail-recursive `invoke`.

- We introduce JAZ, an agent framework whose core primitive is the `invoke` agent loop. An expressive hook system enables utilities such as observability, budget control, validation, and general modifications to the agent loop.
- We show empirically that the additional expressivity of JAZ `invoke` enables long-horizon and self-improvement workflows without external systems, tools, or harness components. With prompting, JAZ `invoke` outperforms CodeAct+subagents at a similar cost, and matches or outperforms specialized harnesses at a substantially lower cost.

## 2 `invoke`: An LLM-based Language Primitive

To motivate the design of JAZ, we consider a simple and general way to extend any existing programming language with an LLM-based primitive. We call this primitive *`invoke`*.

### 2.1 Functional `invoke`

Figure 2 shows an example execution of the *functional* version of the `invoke` primitive. `invoke` is thought of as a function that can take as inputs arbitrary named arguments. These inputs can serve various purposes. Some are inputs in the traditional sense (`df` in Figure 1a, `data` in Figure 2a), some are the specification (`task` in Figure 1a, `spec` in Figure 2a), and some are tools the agent can call (`web_search` in Figure 1a, `tool` in Figure 2a). However, `invoke` treats all inputs the same and makes no distinction between prompts, tools, and "regular" inputs.

Different from a regular function whose implementation is provided in the static code, the implementation of `invoke` is generated by the model every time it is called (Figure 2b). The model sees these inputs as a compact string representation, and writes code that fills in the body of `invoke`. Note that this code is written in the same programming language, which has been augmented with the `invoke` primitive. Appendix B elaborates on the minimality and generality of the primitive. For a formal treatment in lambda calculus, see Appendix C.

### 2.2 The `invoke` Agent Loop

In principle, functional `invoke` is already at least as expressive as a closed agent loop, which can be expressed as tail-recursive `invoke` (Figure 2b–c). To emulate a closed agent loop where the model sees the output of its code and decides its next action based on that, it could write a tail-recursive call to `invoke` that passes in the inputs and the interaction history with the agent loop (Figure 2b). This delegates to a subagent whose context contains the same inputs but also the history so far (Figure 2c), thus behaving as the next step of the agent loop.

In practice, however, this is not a practical approach because it is memory inefficient in Python, and benefiting from LLM caching optimizations requires an exact prefix relationship among consecutive LLM queries. Thus, instead of having the model write the tail-recursive call, we can let the harness automatically append it and apply tail-call optimization, resulting in an actual code-mode agent loop where the LLM writes code in a REPL. This agent loop has two distinguishing properties that are direct consequences of the definition of functional `invoke` (Figures 1 and 2):

1. The *language* of the model's output is the augmented language, which already contains `invoke`. Thus, by default, the model can write not just arbitrary code in the original language, but also code that calls `invoke`, which acts as subagents. We call these recursive `invoke` calls *sub-invokes*.
2. *Everything* the language model sees is a variable that it can reference in its code.
   - This includes *all* inputs to `invoke`, treating all inputs equally regardless of their purpose. Thus, an input intended as the prompt can be accessed as a variable, just as an input intended as a tool.
   - The *REPL history itself* is also a variable in the REPL.

Property 1 makes `invoke` a variant of CodeAct+subagents or RLMs [2, 5]. Property 2 — especially the "REPL history is a REPL variable" aspect — distinguishes `invoke` from existing CodeAct-like agents (e.g., smolagents [4], RLMs [5]), which treat the user prompt and REPL history only as a list of messages shown to the agent, not as variables the agent can programmatically interact with.

We call an agent loop satisfying the two properties above an *`invoke` agent loop*.

## 3 JAZ: An LLM Agent Framework Based on `invoke`

We built JAZ, an agent framework that implements the `invoke` agent loop as its core language model primitive. JAZ additionally implements the following systems to make it easier for users to build agents using the `invoke` primitive.

**Input serialization.** Naively, each input to `invoke` is displayed to the LLM as its `str()` serialization. However, Python's default `str()` has two major issues for this purpose: (a) its size is unbounded, so large objects would blow the model's context window; and (b) it contains no useful information for functions, methods, and objects that don't define their own custom string representation. JAZ addresses (a) by truncating to a configurable length (50000 characters by default). For (b), we implement our own serialization function. In the simplified picture, a function shows its signature and docstring; a class or module shows its docstring, public attributes, and methods; for any other object, we use `str()` if it defines its own custom string representation, else we show the object's public attributes and its class's docstring and methods. Details can be found in our source code.

**Dynamic scoping.** Instead of requiring that every input to `invoke` be provided as an explicit argument, we provide a primitive `scope` that creates a scope of variables where `invoke` calls within the scope automatically receive those variables, including all their recursive sub-invokes. In other words, variables specified by `scope` are subject to *dynamic scoping* for `invoke` calls. In JAZ, `scope` takes the form of a Python context manager. The most common use case is to make a tool available not just to the top-level agent, but also to all its recursive subagents:

```
from jaz import invoke, scope

def web_search(...):
    ...

with scope(web_search=web_search):
    # Top-level agent and all subagents can use `web_search`
    invoke(task="Use subagents to find 100 agent framework papers.")
```

Although traditional programming prefers static scoping over dynamic scoping, only dynamic scoping is appropriate for `invoke`. Static scoping enables reusing a function in different contexts without surprising changes in behavior due to a changing context. However, `invoke` does not have a static scope as it does not have a static implementation, and there is no reuse as every call to `invoke` generates a new implementation.

**Hooks.** Hooks allow a user to extend `invoke` by providing callbacks that observe or influence the agent as it is running. JAZ provides the following built-in hooks:

- ***Observability:*** `PrintLogger`, `FileLogger`, `TrajectoryDirectoryRecorder`, `TrajectoryRecorder`, `LangfuseTracing`, `JaegerTracing`.
- ***Resumability:*** `TrajectoryRecorder`, `TrajectoryReplay`

- ***Resource control:*** `BudgetPool`, `IterationLimit`, `RecursionLimit`, `BudgetForcing`, `ContextWindowWarning`.
- ***Validation:*** `ReturnType`, `ValidateReturn`, `ValidateREPLCode`.

Users can also extend JAZ by writing their own custom hooks (Appendix D).

Just like input variables, hooks can be either local to an `invoke` or be subject to dynamic scoping. Hooks passed as explicit positional arguments to `invoke` are local, while a hook activated as a context manager is scoped. In the following example, the scoped `BudgetPool(max_cost=5)` hook applies a budget of $5 shared across all `invoke` calls under the context manager, including both top-level calls and all their recursive sub-invokes. On the other hand, the local `ReturnType(float)` enforces the return type `float` only for its top-level `invoke` and not for its sub-invokes.

```
from jaz import invoke
from jaz.hooks import ReturnType, BudgetPool

with BudgetPool(max_cost=5):  # scoped hook
    invoke(
        ReturnType(float),  # local hook
        question="What is the value of ...?"
    )
    invoke(another_question="Why is ...?")
```

## 4 Case Studies

In this section, we explore how the expressivity of JAZ `invoke` allows us to replace specialized external harnesses with prompting for two types of workflows: 1) long-horizon workflows requiring recall beyond the context window; 2) continual learning and self-improvement across a sequence of tasks. These workflows are traditionally targeted with external harness components, such as memory systems for long-range recall, and specialized self-improving systems for continual self-improvement. Here, we use JAZ to demonstrate that a minimal LLM-based primitive, the `invoke` agent loop, can perform these workflows as well without external systems, tools, or harness components.

We compare `invoke` with agent loop paradigms under a similar prompt-only setup (CodeAct and CodeAct+subagents [2, 5]), described below. We also compare with domain-specific harnesses (Letta/MemGPT [7] and ACE [9]), described in the individual subsections (Sections 4.1 and 4.2).

- **CodeAct** [2] is an agent loop paradigm generalizing ReAct [1] tool calling where the model's action is not a single tool call, but a piece of code that orchestrates tool calls. Implementations [3, 4] involve a code REPL that is seeded with user-provided tools and objects, and the LLM writes code in the REPL in a closed loop. The core difference from the `invoke` agent loop is that the user prompt and the REPL history are only shown to the LLM and not available as variables in the REPL.
- **CodeAct+subagents** is the agent loop paradigm that augments CodeAct with recursive subagents, where "subagent" is understood to mean generalist subagents instead of specialized subagents with designated roles. This paradigm was popularized by the RLM work [5], which implemented a version where the REPL is seeded with a large string, making the agent act as a long-context language model.

To minimize confounds due to implementation differences, we reimplement CodeAct and CodeAct+subagents in JAZ as controlled ablations of `invoke`. CodeAct+subagents is achieved by a custom hook that removes the user prompt and REPL history variables from `invoke`'s REPL and makes the minimal equivalent change to prompt templates. CodeAct additionally uses the built-in `RecursionLimit` hook to disable recursive subagents. This was necessary as established implementations were found to perform worse than our reimplementation due to defects in prompting and REPL implementation as opposed to the actual distinction between CodeAct(+subagents) and `invoke`. See Appendix E for more details.

Both our environments involve completing a sequence of tasks, so a return guard (`ValidateReturn` in JAZ, `final_answer_checks` in smolagents) is used to make the agent keep working if it stops before all tasks have been attempted.

Table 1: Results on the long-horizon environment StuLife [6]. Methods marked "per-task" solve each StuLife task individually with no state or memory persistence across tasks. We report both pass rate (fraction of scored tasks with a perfect score) and average score, for both the full scored set of 939 tasks ("all") and the subset of 207 tasks that require recall of information delivered over 50 tasks ago ("far recall"). We use GPT-5.4 nano (high) across all methods. Per-task methods report a 3-run mean and combined per-task SEM, and all other methods report a 6-run mean and SEM to account for higher variance. See Table 3 for all individual data points.

| | prompt-only? * | StuLife (all) | | StuLife (far recall) | | |
|---|---|---|---|---|---|---|
| | | Pass (%) | Score (%) | Pass (%) | Score (%) | Cost ($) |
| $\text{CodeAct}_{\text{JAZ}}$ [2] (per-task) | ✓ | $52.5 \pm 0.4$ | $59.2 \pm 0.3$ | $24.8 \pm 0.9$ | $25.8 \pm 0.9$ | $\mathbf{4.4} \pm 0.1$ |
| $\text{CodeAct+subagents}_{\text{smolagents [4]}}$ | ✓ | $34.5 \pm 4.8$ | $37.8 \pm 5.1$ | $20.9 \pm 0.4$ | $22.0 \pm 0.6$ | $\underline{11.7} \pm 2.2$ |
| $\text{CodeAct+subagents}_{\text{JAZ}}$ | ✓ | $63.3 \pm 2.0$ | $71.8 \pm 2.2$ | $38.5 \pm 4.7$ | $41.1 \pm 5.2$ | $13.8 \pm 0.7$ |
| $\text{Letta/MemGPT}_{\text{less search}}$ [7] | ✗ | $71.0 \pm 0.3$ | $81.1 \pm 0.4$ | $61.5 \pm 1.1$ | $66.7 \pm 1.2$ | $45.1 \pm 3.4$ |
| $\text{Letta/MemGPT}_{\text{more search}}$ | ✗ | $\underline{72.8} \pm 0.2$ | $\mathbf{83.3} \pm 0.4$ | $\underline{66.6} \pm 1.2$ | $\underline{71.4} \pm 1.2$ | $93.8 \pm 9.9$ |
| JAZ `invoke` (ours) | ✓ | $\mathbf{72.9} \pm 0.2$ | $\underline{82.4} \pm 0.4$ | $\mathbf{70.4} \pm 0.9$ | $\mathbf{74.7} \pm 1.0$ | $18.0 \pm 0.2$ |

* See Appendix E.1 for our definition of a prompt-only setup with a generic agent loop

### 4.1 Long-Horizon with Long-Range Recall

Here, we focus on long-horizon environments with the following properties:

- The task requires so many iterations (e.g., thousands) that the conversation history would exceed the LLM's context window multiple times if no subagents were used.
- Making the optimal decision frequently requires recalling some detail in the conversation history multiple context windows in the past. Compaction would lose the detail, and long-range dependencies make it difficult to cleanly decompose the workflow into subtasks.

In JAZ, when the LLM's context window fills up, the agent can delegate the remainder of the task to a subagent while losslessly passing in the entire conversation history by reference. We call this pattern *tail-recursive delegation*:

```
return invoke(
    ...,  # original inputs to the top-level invoke
    prev_history=globals().get("prev_history", []) + __history__,  # REPL history
    prev_progress_summary=...,  # summary of history
    next_steps=...,  # next steps for the subagent
    ...,  # other state to keep track of
)
```

Note that while the top-level agent passes its `__history__` to the subagent, any subagent must combine its `__history__` with the `prev_history` passed to it before delegating further. In our experiments, we elicit this behavior with the built-in `ContextWindowWarning` hook, which appends a user message to the agent's conversation when the model's context window passes a certain threshold. The user message tells the agent to delegate all remaining work to a subagent and shows a code template to follow — see Appendices E.2 and G.1.2.

The general pattern above subsumes various forms of long-horizon context management. For example, *compaction* [10, 11] corresponds to passing in a `prev_progress_summary` only. Automated context management (e.g., Chroma Context-1 [12]) corresponds to passing in a filtered version of `__history__` with certain entries removed. More generally, access to `invoke` gives the agent control over the next LLM call's inputs and context, whereas access to `__history__` and a code environment makes it efficient to build this context off of the agent's current context.

**Experiments.** We evaluate on the long-horizon benchmark StuLife [6], which contains an ordered sequence of 1284 tasks that simulate a variety of activities a college student does over the course of a semester, many of which require recalling information given in prior tasks. A typical episode takes 7000–8000 environment interactions, and a typical code-mode agent requires 3000–5000 LLM calls.

Among the 1284 tasks, 939 are graded, of which 207 require "far recall", which we define to be recall of information delivered in a previous task separated by over 50 tasks.

We evaluate JAZ `invoke` and multiple baselines using GPT-5.4 nano as the model. Since JAZ `invoke` is a general agent harness, our primary baseline is CodeAct+subagents, also a general agent harness. The CodeAct (per-task) baseline runs CodeAct on each individual StuLife task with no cross-task memory persistence and thus serves as a no-memory baseline. We also compare with a domain-specific harness: Letta v0.16.8, a conversation agent with memory. Letta v0.16.8 is the latest version of Letta that is based on the MemGPT memory architecture [7], so we will also refer to it as "Letta/MemGPT". See Appendix E.2 for experiment details. Evaluation results are reported in Table 1.

**Analysis.** To understand where JAZ `invoke` has an advantage over CodeAct+subagents and Letta/MemGPT on tasks requiring long-range recall, we analyzed task #1282/1284, where `invoke` solved it in all 6 runs, CodeAct+subagents got 3/12, and Letta/MemGPT got 9/12.

This task is a final exam multiple-choice question. Each of the four options describes applying a fictional protocol, and the agent has to choose the option with the correct application based on what previous lectures taught about these protocols. The full task text is given in Appendix H.1.

To solve this task, the agent needs to first recall the lectures that taught those protocols, then apply them in the scenarios given in the question to deduce the correct option. The lecture that taught the protocol involved in the correct option (D) was introduced in task #785/1284, 497 tasks ago.

On one of the JAZ `invoke` runs, when the agent reached this task, it was already at recursion depth 70 after having performed tail-recursive delegation 69 times.[2] However, every delegation preserves the full history completely through the `prev_history`, which has accumulated every prior agent's `__history__`. The agent spent one turn searching all protocol names in its `prev_history`. All the relevant lectures were correctly retrieved, and the agent answered correctly in its next turn.

Most CodeAct+subagents runs did not search at all. The user prompt to the agent warns that these exam questions are about fictional protocols, so it has to recall the actual lecture content to answer correctly. The user prompt also contains the instructions for maintaining the agent's own REPL history, compensating for CodeAct's lack of the `__history__` variable. However, without access to the user prompt as a variable in the REPL, every agent has to copy its user prompt into the subagent, which ended up being lossy, and both instructions were lost after dozens of delegations. The agent ended up not having access to any form of REPL history variable and no longer knew that the question tested recall, so it answered based on its real-world knowledge, choosing (A), which was incorrect in StuLife's fictional world.

**Takeaway 1:** *(long-horizon version)* When *everything* the agent sees in its context is in REPL variables, the agent can pass them by reference to a subagent in scenarios that need it (e.g., tail-recursive delegation). This is more reliable than copying their contents by hand.

Letta/MemGPT used its `conversation_search` tool to search through its conversation history for exact protocol names mentioned in the question. Using a combination of keyword search and vector search, the tool returned the top-ranked hits, which in some runs were either the quiz question itself, or earlier messages about similarly named but different protocols. In this scenario where the agent needs exact substring matching, the only tool Letta had (`conversation_search`) did not support it.

**Takeaway 2:** *(long-horizon version)* Domain-specific harnesses encode assumptions that make them work well in many situations, but they become rigid in environments that break those assumptions. Instead of a search tool over a large object, having full programmatic access to its raw interface can recover the flexibility needed in such environments.

### 4.2 Continual Self-Improvement

We study *continual self-improvement (CSI)*, where a system improves itself over a given ordered sequence of tasks.

[2] Recursion depth is defined as 0 in user code, 1 in the top-level agent, 2 in the first level of subagents, etc.

Table 2: Results on the full `test-challenge` split of AppWorld [8]. Methods marked "per-task" solve each AppWorld task individually with no continual learning across tasks. The solver agent in all methods uses GPT-5.4 nano (high) and the meta-agent in CSI methods uses GPT-5.4 (high). Per-task methods report a 3-run mean and combined per-task SEM, and all other methods report a 6-run mean and SEM to account for higher variance. See Table 4 for all individual data points.

| | prompt-only?* | AppWorld (`test-challenge`) | | | | |
|---|---|---|---|---|---|---|
| | | TGC (%) | SGC (%) | Cost ($) | Meta $ | Solver $ |
| CodeAct$_{\text{AppWorld [8]}}$ (per-task) | ✗ † | 48.2 ± 1.1 | 20.4 ± 1.6 | <u>16.6</u> ± 0.2 | — | 16.6 ± 0.2 |
| CodeAct$_{\text{JAZ}}$ [2] (per-task) | ✓ | 67.5 ± 1.0 | 43.4 ± 1.7 | **10.2** ± 0.1 | — | **10.2** ± 0.1 |
| CodeAct+subagents$_{\text{JAZ}}$ [5] | ✓ | 71.1 ± 1.3 | 47.6 ± 2.1 | 21.6 ± 1.8 | **7.3** ± 1.7 | 14.3 ± 0.9 |
| ACE$_{\text{train 10\%; on CodeAct}_{\text{JAZ}}}$ [9] | ✗ | 69.9 ± 1.4 | 47.1 ± 1.3 | 30.6 ± 0.5 | 17.1 ± 0.3 | 13.5 ± 0.3 |
| ACE$_{\text{batch; train all; on CodeAct}_{\text{JAZ}}}$ | ✗ | **74.9** ± 0.9 | **52.2** ± 1.4 | 116.1 ± 0.9 | 102.2 ± 0.7 | 13.9 ± 0.3 |
| JAZ `invoke` (ours) | ✓ | <u>74.2</u> ± 2.1 | <u>51.1</u> ± 3.7 | 20.9 ± 3.7 | <u>9.9</u> ± 3.8 | <u>10.9</u> ± 0.6 |

* See Appendix E.1 for our definition of a minimal, prompt-only setup with a generic agent loop
† The official CodeAct baseline uses a prompt that contains AppWorld-specific workflow guidance.

One form of self-improving system separates a *meta-agent* from a *solver-agent*, where the meta-agent optimizes various aspects of the solver agent. Targets of optimization can include the prompt [9, 13, 14], executable skills [15], and even the entirety of the agent's source code [16].

In JAZ, the top-level `invoke` has control over all inputs to pass into a sub-invoke and can thus act as a meta-agent by optimizing those inputs over a sequence of tasks. A typical optimization iteration involves constructing inputs such as the solver agent's prompt and skills (meta-authored tools), running them on a task, and inspecting the results, as described by the following pseudocode:

```
instructions = "... When done, return `(answer, __history__)`"

def tool(...): ...  # function that calls base environment methods

answer, trajectory = invoke(
    task=get_next_task(),
    instructions=instructions,
    tool=tool,
)

eval_report = complete_task(answer)
print(eval_report)
print(trajectory)
```

The prompt used to guide the top-level agent to conduct continual self-improvement is given in Appendix G.2. It describes the high-level continual self-improvement workflow and provides information about JAZ needed for proper implementation of self-improvement. The prompt does not include any code examples or templates.

Since our CodeAct baseline runs on each individual task without subagents, our CodeAct+subagents and JAZ `invoke` implementations cap recursion depth to 2 to reduce confounds (i.e., solver subagents are not given sub-subagents).

**Experiments.** We evaluate JAZ `invoke`'s ability to continually self-improve across the full sequence of 417 tasks from the `test-challenge` split of the AppWorld benchmark [8]. The task ordering was shuffled with seed 42. AppWorld is an environment that simulates the API interfaces of 9 apps people commonly use on their phone or computer, and tasks require the agent to use those APIs to complete tasks for the user. In our continual self-improvement setup, the full set of tasks is given to the agent in a fixed order, with full test feedback from each task so that the agent can learn from its mistakes.

We compare JAZ `invoke` with an equivalent minimal prompt-only setup of CodeAct+subagents, as well as Agentic Context Engineering (ACE) [9], a specialized harness for continual self-improvement. We also compare with baselines that solve each benchmark task individually: the benchmark's official

CodeAct baseline[3], as well as our own implementation of CodeAct. These represent baseline methods that do not use test feedback to improve across tasks.

Table 2 reports the mean and standard error of our evaluation results. Table 4 lists all individual data points, and additionally reports the median with its 78% confidence interval for self-improving methods. We find that JAZ `invoke` outperforms all other methods in the mean and median.

**Analysis.** We qualitatively compare the continual self-improvement behavior of prompt-only JAZ `invoke`, prompt-only CodeAct+subagents, and the specialized self-improvement harness ACE [9].

JAZ `invoke`'s top-level `invoke` acted as a meta-agent that launched subagents to solve individual tasks. It dispatched tasks in batches to subagents, investigated failure traces and metrics in between batches, updated inputs to subagents (prompts and tools), and adaptively updated the batch size. To access subagent trajectories, the meta-agent simply had to tell the subagent to return its `__history__` variable available in its REPL.

CodeAct+subagents followed the same workflow as JAZ `invoke`, but it had to get around the limitation that subagents didn't have access to their own REPL histories. The meta-agent told the subagent to return its REPL history, and depending on the prompt it wrote, the subagent either hard-coded the entire history in the final return statement, or incrementally built it by appending it at each turn. Both approaches tended to be lossy, and the additional overhead contributed to output token inefficiency and hence a higher solver agent cost.

**Takeaway 1:** *(self-improvement version)* When *everything* a subagent sees in its context is in REPL variables, the subagent can pass them by reference to its caller in scenarios that need it (e.g., for caller observability into subagent behavior). This is more reliable than explicitly building their contents.

ACE is a human-designed self-improvement loop that optimizes the solver agent's prompt after every task it attempts. It uses a *reflector* LLM call to reflect on the solver agent's trajectory, and then a *curator* to distill findings into bullet point items that are added to the *playbook* that forms part of the solver agent's prompt. The ACE reflector and curator were very costly, costing 10 times more per task than JAZ `invoke`. We restricted learning to the first 42 (10%) tasks to balance cost and performance (Appendix E.3), and ACE still cost more than JAZ `invoke` learning across the entire test set. The cost efficiency of JAZ `invoke` comes from breaking free of the rigid workflow prescribed by ACE. In JAZ `invoke`, the meta-agent has the freedom to batch multiple tasks together and save on meta-agent model calls. The meta-agent also selectively reads parts of trajectories when needed, thus saving its context from being filled with the full trajectory of every subagent.

Another limitation of ACE is that it only optimizes the prompt, whereas the JAZ `invoke` meta-agent optimizes inputs to `invoke` generally, which can also include meta-authored tools.

**Takeaway 2:** *(self-improvement version)* Hand-designed workflows encode assumptions that make them work well in many situations, but they become rigid in environments that break those assumptions. Providing a strong LLM the full expressivity of code and the `invoke` primitive can recover the flexibility needed in such environments.

## 5 Conclusion

We demonstrated that behaviors commonly attributed to specialized agent harnesses (e.g., memory and self-improvement) can instead be elicited through prompting, once the agent loop becomes expressive enough to allow the model to express these behaviors. This suggests a shift in how LLM agents should be built. Instead of harness engineering, the more "bitter-lesson pilled" approach may be to train a model within a minimal yet maximally expressive agent loop, so that it could learn to construct workflows, tools, and harness components by itself. A minimal harness with the least inductive bias allows for the most general-purpose learning, and maximal expressivity gives the agent the freedom to achieve any desired behavior or capability.

[3] Known as the "official ReAct" baseline with a REPL tool, which is equivalent to CodeAct

## Acknowledgments and Disclosure of Funding

This project was funded by the following grants and institutions: Zup/Itaú, MIT CSAIL FinTech AI Initiative, NSF Grant No. 1918839.

## References

[1] 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*, 2023.

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

[3] Harrison Chase. LangChain v0.0.97. `https://github.com/langchain-ai/langchain/releases/tag/v0.0.97`, 2023.

[4] Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. `smolagents`: a smol library to build great agentic systems. `https://github.com/huggingface/smolagents`, 2025.

[5] Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

[6] Yuxuan Cai, Yipeng Hao, Jie Zhou, Hang Yan, Zhikai Lei, Rui Zheng, Zhenhua Han, Yutao Yang, Junsong Li, Qianjun Pan, et al. A benchmark for self-evolving agents via experience-driven lifelong learning. *ICLR 2026 submission*, 2026.

[7] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[8] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, 2024.

[9] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

[10] Qingyue Wang, Yanhe Fu, Yanan Cao, Shuai Wang, Zhiliang Tian, and Liang Ding. Recursively summarizing enables long-term dialogue memory in large language models. *Neurocomputing*, 639:130193, 2025.

[11] Anthropic. Compaction. `https://platform.claude.com/docs/en/build-with-claude/compaction`, 2026. Accessed 2026-05-07.

[12] Hammad Bashir, Kelly Hong, Patrick Jiang, and Zhiyi Shi. Chroma Context-1: Training a self-editing search agent. Technical report, Chroma, March 2026.

[13] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7080–7106, 2026.

[14] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

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

[16] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

[17] 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*, 2024.

[18] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024.

[19] Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. *Advances in Neural Information Processing Systems*, 37:67061–67105, 2024.

[20] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023.

[21] Jonathan Light, Wei Cheng, Benjamin Riviere, Yue Wu, Masafumi Oyamada, Mengdi Wang, Yisong Yue, Santiago Paternain, and Haifeng Chen. DISC: Dynamic decomposition improves LLM inference scaling. In *Advances in Neural Information Processing Systems (NeurIPS 2025)*, 2025.

[22] Anthropic. Claude Code v2.1.278. `https://github.com/anthropics/claude-code`, 2026.

[23] Yujian Liu, Jiabao Ji, Li An, Rohit Jain, Gungor Polatkan, Siyu Zhu, and Shiyu Chang. Practical online KV cache compaction for LLM agents: An empirical study. *arXiv preprint arXiv:2608.00902*, 2026.

[24] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

[25] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pages 104223–104294, 2026.

[26] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024.

[27] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. In *ICLR 2025 Workshop on Scaling Self-Improving Foundation Models*, 2025.

[28] Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel Agent: A self-referential agent framework for recursively self-improvement. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27890–27913, Vienna, Austria, July 2025. Association for Computational Linguistics.

[29] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

[30] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pages 21344–21377, 2025.

[31] Jonathan Light, Min Cai, Weiqin Chen, Guanzhi Wang, Xiusi Chen, Wei Cheng, Yisong Yue, and Ziniu Hu. Strategist: Self-improvement of LLM decision making via bi-level tree search. In *International Conference on Learning Representations*, volume 2025, pages 27032–27099, 2025.

## A Related Work

**Agent loop paradigms.** Most modern LLM agents are built around an *agent loop* in which the model interacts repeatedly with an environment through actions and observations. In traditional tool-calling or ReAct-style agents [1], each action invokes a *tool* with hard-coded inputs. More recent *code-mode* or *CodeAct-style* agents instead place the model inside a code environment (REPL), where tools and subagents are now ordinary functions that code can reference [2].

Recursive language models [5] demonstrate that a code-mode agent loop itself — with only prompting, no external tools or harness components other than recursive subagents — can handle some long-context tasks better than long-context models themselves. Our work extends this line of work by demonstrating that, with a natural generalization — all agent inputs and the interaction history with the code environment are variables in the code environment — a code-mode agent loop with only prompting can also handle long-horizon tasks and perform continual self-improvement.

**Multi-agent systems.** Multi-agent systems for task decomposition have been built for solving complex long-horizon tasks [17, 18, 19, 20, 21]. Traditionally involving significant hand design, such as subagent roles and specific orchestration patterns, more recent multi-agent systems [22, 5] take a more minimal approach involving generalist subagents exposed as regular tools. While traditionally subagents have been used for well-defined subtasks, we demonstrate that they are also useful for long workflows without well-defined subtasks, and can also be used as the solver agent in a self-improving system with a meta-agent/solver-agent split.

**Compaction and memory systems.** For non-decomposable long-horizon workflows such as long conversations, the predominant approach has been *compaction* and *memory systems*. Compaction methods include simple textual summarization [10, 7], and recent work has also explored applying KV cache compaction techniques to long-horizon agentic workflows [23]. To enable retrieval of previous conversation context that has been lost due to compaction, memory systems were developed that provide the agent an interface to query conversation history through keyword or embedding-based search [7, 24]. In JAZ, we showed that full programmatic access to a variable `__history__` is sufficient for memory retrieval.

**Self-improving agentic systems.** Self-improving systems take a variety of forms, including agents that modify their own source code [25, 26, 27, 28] and systems involving a meta-agent that optimizes a target agent [29, 14, 30, 9, 16]. Aspects of the target agent to be optimized can include the prompt [29, 14], skills (high-level tools) [15, 31], and entire agent programs [30, 16]. In JAZ, we demonstrated that the top-level agent and subagent in a multi-agent system can act as the optimizer meta-agent and the optimized solver agent in self-improvement, optimizing inputs to the solver agent such as its prompt and tools.

## B `invoke` Is Minimal and General

Here, we illustrate informally how functional `invoke` arises naturally from considering the *minimal* LLM primitive that can be generally introduced to *any* Turing-complete language.

The usual LLM primitive takes a string input and outputs a string, and that is arguably the minimal LLM primitive in programming languages that *have strings*. However, the simplest languages, such as lambda calculus, do not have a string data type and thus do not admit the usual LLM primitive. Nonetheless, *the terms of a language themselves are strings*, irrespective of the language. So the LLM can simply take terms in the language as inputs, and output a term in the language. If the output term can reference inputs as variables, then we arrive exactly at the definition of functional `invoke`. Its syntax is that of a function call that takes arbitrary named inputs, and the function call's semantics is to query an LLM to generate the body of the function, which is executed. Thus, the definition of `invoke` arises naturally from modifying the usual LLM primitive to be generally applicable to all Turing-complete languages.

Furthermore, as shown in Appendix C, extending a language with `invoke` introduces just one new piece of syntax and one additional rule to the formal semantics, so `invoke`'s formal definition exemplifies its minimality as well.

## C Formal definition of `invoke`

Given a programming language, we augment its syntax with the *`invoke` primitive*, which is a function call where the body of the function is written dynamically at runtime by a language model (LM). We write $\texttt{invoke}(x_1 := e_1, \ldots, x_k := e_k)$ for the functional agent call on *inputs* $x_i := e_i$, where $x_i$ are variable names and $e_i$ are expressions in the language. To execute a program in this augmented language, $\texttt{invoke}(x_1 := v_1, \ldots, x_k := v_k)$ is evaluated as follows:

1. Query a language model with a prompt $P$ that contains a compact string representation $P = \langle x_1 := v_1, \ldots, x_k := v_k \rangle$ of the inputs. These inputs can include the user prompt, tools, and more generally any object in the language allowed as an argument to a function call.
2. The model returns code $C = \pi(\mathsf{LM}(P))$ in the same language that would be valid as the body of a function with input parameters $x_1, \ldots, x_k$. Here, $\pi(\cdot)$ parses the code from the model's raw output. Note that all the context that the model sees is contained within the prompt $P$ that serializes the inputs.
3. Execute this function on the given inputs.

Since the language itself includes the `invoke` primitive, the model-generated code may recursively contain further `invoke` calls.

Under this framework, there are three key aspects to design: the language itself, the *serializer* formatting the inputs into a prompt (represented by the angle bracket notation "$\langle\cdot\rangle$"), and the *model output parser* $\pi(\cdot)$. A good serializer does not display the raw contents of a large object; instead it gives the model enough information so that it could extract information needed by interacting with it. The model output parser traditionally involves parsing out a code block from the model's natural language output. However, for simplicity, it can just be the identity function $\pi(y) = y$, which was the choice we made in JAZ — the entirety of the model's output is parsed as Python code, and any natural language prose (e.g., the agent's plan) is included in leading comments inside the code.

Formally, consider the lambda calculus, the simplest functional programming language, defining just function application and function abstraction. The following formal definition is easily generalized to other programming languages.

Augmenting the lambda calculus with the `invoke` primitive results in this augmented grammar of expressions (terms):

$$e ::= \underbrace{c \mid x \mid e\ e \mid \lambda x.\, e}_{\text{lambda calculus}} \mid \texttt{invoke}\ (x := e)^+. \tag{1}$$

The standard expressions in lambda calculus are constants ($c$), variables ($x$), function calls ($e\ e$), and function definitions ($\lambda x.\, e$). The additional syntax is the new primitive $\texttt{invoke}\ (x := e)^+$. Here, each $e$ denotes an arbitrary expression in the language, and $(x := e)^+$ means "one or more named arguments".

The operational semantics for call-by-value (CBV) evaluation become

$$\underbrace{\frac{}{(\lambda x.\, e)\ v \to e[v/x]}\ {}^{1} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ {}^{2} \qquad \frac{e \to e'}{v\ e \to v\ e'}\ {}^{3}}_{\text{lambda calculus}} \qquad \frac{\sigma = [\vec{x} := \vec{v}]}{\texttt{invoke}\ \sigma \to (\lambda \vec{x}.\, \pi(\mathsf{LM}\langle\sigma\rangle))\ \vec{v}}\ {}^{4} \tag{2}$$

where the metavariable $v$ denotes a value (the result after evaluating an expression), and $\sigma$ is the mapping $x_1 := v_1, \ldots, x_k := v_k$. The first three rules are standard, defining how to evaluate function calls (rule 1) and evaluation order (rules 2 and 3). The `invoke` addition (rule 4) defines how to evaluate the functional agent call primitive, formalizing the notion of filling in the body of a function dynamically at runtime. $\mathsf{LM}\langle\sigma\rangle$ denotes the output of the language model given the *prompt* $\langle\sigma\rangle$, i.e., a string representation of all the inputs. $\pi$ is the *model output parser* that parses the model's raw output. The extracted code appears as the body of a function $\lambda\vec{x}.\, (\ldots)$ with arguments $x_1, \ldots, x_k$, and the function is invoked on values $v_1, \ldots, v_k$ as given by the mapping $\sigma$.

# D Extensibility via Hooks in JAZ

In JAZ, users can define custom hooks by providing functions to call at various points in the agent's lifecycle. Instead of allowing a hook to arbitrarily modify agent state, it is only allowed to emit static composable effects as allowed by the particular event.

- `InvokeEnter`, `InvokeSend`, `InvokeComplete`, `InvokeExit`
- `LLMQueryEnter`, `LLMQuerySend`, `LLMQueryComplete`, `LLMQueryExit`
- `LLMQueryRetry` (only fires if an LLM query did not succeed and is retried)
- `REPLExecEnter`, `REPLExecSend`, `REPLExecComplete`, `REPLExecExit`

An event handler at `[Operation]Enter` sees the inputs to `[Operation]` and can emit effects that modify those inputs. An event handler at `[Operation]Send` sees the input modifications and resultant modified inputs, and can emit an effect that replaces calling an `[Operation]` with supplying its output. An event handler at `[Operation]Complete` sees the output and can emit effects that modify the output. In addition, at every event other than `InvokeExit` and `LLMQueryRetry`, the handler may emit an effect that aborts the `invoke` it fires in.

Every `[Operation]Enter` opens a span that always closes with `[Operation]Exit`, carrying its outcome (`Completed/Aborted/Failed`). `[Operation]Send` and `[Operation]Complete` are conditional: an abort or failure skips to `[Operation]Exit`.

When multiple hooks are active at once, effects that modify are *not* applied as they are emitted, as that would cause different hooks to potentially see different event fields in a way that depends on the order in which the event is dispatched to all the active handlers. Instead, the event is dispatched to all handlers at once, and all effects are collected and composed with a composition rule that minimally depends on composition order. Conflicting writes that cannot be resolved are refused with an error, and then the final resolved modifications are all applied at once.

Hooks can also communicate via a global message board called the *blackboard*, reading its contents off the event and writing by emitting an effect.

All handlers also have a view of ambient state: all active hooks, all scoped variables (Section 3), the active configuration (Section 3), and the blackboard.

All built-in hooks provided by JAZ are built using the hook interface we've described in this section.

# E Experiment Details

## E.1 Evaluation Protocol

To ensure fair comparison, accurate measurements, and proper control of confounds, we follow the following protocol in our evaluations.

**Task/method decoupling.** The hyperparameter with the most degrees of freedom is the prompt. If each (method, task) pair gets its own prompt, there is a high risk of overfitting the method to the task, and different methods under comparison may receive different degrees of overfitting that confound the comparison. This is especially dangerous with self-improvement, where allowing a self-improvement method's meta-prompt to include task-specific information licenses hard-coding aspects of the optimal solver prompt and skills in the meta-prompt instead of forcing the meta-agent to learn these on its own, which would no longer isolate the effect of self-improvement itself.

To mitigate these risks, our experiments split the user prompt into two disjoint parts: `instructions` and `guidance`. The environment-provided `instructions` is method-agnostic — it provides the task instructions, describes the environment, and optionally provides workflow guidance as long as it is method-agnostic. In self-improvement experiments, to isolate the ability to self-improve starting from a minimal seed, `instructions` only contains the task instructions and environment description and provides no workflow guidance at all. On the other hand, the method-provided `guidance` is task-agnostic — it provides workflow guidance as appropriate for the method, as long as it is applicable to all tasks in the domain. In JAZ, the prompt components `instructions` and `guidance` are passed as separate variables to JAZ `invoke`.

Note that we consider "[agent loop] long-horizon" and "[agent loop] self-improvement" to be separate methods that can provide their own prompts, but the prompt must still be task-agnostic. Thus, the long-horizon prompt (Appendices G.1.1 and G.1.2) and the self-improvement prompt (Appendix G.2) are different prompts, but the long-horizon prompt is generally applicable to long-horizon environments with no StuLife-specific information, and the self-improvement prompt is generally applicable to self-improvement with no AppWorld-specific information. When the official implementation of a baseline (e.g., ACE [9]) violates this rule (Appendix E.3), we make the minimal modification so that it conforms.

**Necessary differences between method implementations are implemented.** A `guidance` prompt written for one method is typically not applicable to another method and thus must be adapted. For example, compared to JAZ `invoke`, CodeAct+subagents does not expose the user prompt components (`instructions` and `guidance`) as variables in the REPL and does not automatically maintain a `__history__` variable in the REPL. Thus, for long-horizon, we adapt the `guidance` prompt to ask the agent to maintain its history by itself, and we adapt the context window warning to ask the agent to hard-code the contents of `instructions` and `guidance` instead of referring to their variables. Similarly, for self-improvement, we adapt the `guidance` prompt to ask the meta-agent to ask the solver agent to maintain its history by itself so that it could return it at the end.

**Unnecessary differences between method implementations are minimized.** When two methods each have tunable hyperparameters — especially the prompt, but also the configuration — those that can be kept the same across the two methods without introducing bias towards one method or the other are kept the same. For example, JAZ truncates inputs to 50000 characters by default, so in the smolagents implementation of CodeAct+subagents, the context window warning asks the agent to wrap its history list in an object whose custom `repr` truncates to 50000 characters. For prompts, the JAZ `invoke` prompt and the CodeAct+subagents$_{\text{JAZ}}$ prompt are byte-identical everywhere other than in places where a difference is necessary.

**Cost measurement.** For accurate cost measurement, we controlled for systematic errors in cache hit rate in two ways. First, we set a distinct prompt cache key for each run. Second, we allowed at most 3 parallel runs at any moment under the same OpenAI organization. (Multiple OpenAI organization accounts were created to run more than 3 experiments in parallel.) To estimate the systematic uncertainty due to uncontrollable factors in the state of OpenAI server-side cache, we ran $n$ parallel agents on a simple toy environment for $n = 1, 3, 4, 5$, both with and without another agent running on a different environment at the same time. We calculated the mean and its standard error of the cache hit rate. We found a notable decrease in cache hit rate of over 5% from $n = 4$ to $n = 5$, while changes in cache hit rate for $n \leq 4$ were within 2%. We therefore fix $n = 3$ for all our experiments, and estimate the systematic error in resultant cost estimates to be less than 2%. Experiments that involved 6 independent runs (i.e., self-improvement) were run in 2 batches of 3.

**Rules for the prompt-only setup with a generic agent loop.** The "minimal" setup we apply to all general agent loop methods in our experiments applies the following restrictions to all methods equally:

- The only tools are those provided by the benchmark's environment itself. More generally, the agent is not allowed to access anything other than the given environment, such as the file system or the Internet.
- There is no loop-external logic that modifies the loop itself (e.g., truncating the conversation history, injecting variables in the REPL). However, *monitoring* is allowed: budget control, step limits, recursion limit, completion guard, and context-window limit warnings. We also allow model configuration, i.e., the root agent may use a different model from subagents.
- The system prompt — available to the root agent as well as subagents — contains agent loop mechanics and tool docstrings.
- The user prompt — passed to the root agent only — contains a method-agnostic task prompt supplied by the environment and a task-agnostic guidance prompt supplied by the method.

The details of implementing a general agent loop may depend on the domain ("long-horizon" or "self-improvement"), but must be task-agnostic. For example, the method's contribution to the user prompt and context window warning must be applicable to "long-horizon" in general, or "self-improvement"

in general, with StuLife- or AppWorld-specific information provided only by the environment in a method-agnostic manner.

### E.2 Long-Horizon: StuLife

#### E.2.1 Environment Details

The StuLife repository provides a JSON file containing the 1284 tasks of the environment. We implement an environment that delivers these tasks and exposes a modified set of tools to the agent. We reuse the official grader for binary scoring and implement our own partial score metric. Our modified set of tools removes tools that are not needed for any tasks, and replaces a few tools that are always used the same way with a single tool for the same purpose (see code for details). The original tools also only output unstructured text and never raise on invalid input, so they are wrapped to take in structured inputs, output structured outputs, and perform input validation to be friendly to code-mode agents. Their docstrings are also updated accordingly, and code examples are included in docstrings to improve clarity.

#### E.2.2 Method Details

For our long-horizon experiments with JAZ, we have two prompts: one is the warning emitted when the LLM's context window is 70% full, injected with the `ContextWindowWarning` hook, and the other is part of the regular user prompt given to the top-level agent. The prompt injected by the `ContextWindowWarning` is given in Appendix G.1.2. The prompt shown to the top-level `invoke` is provided inside a regular input named `guidance` and its contents are given in Appendix G.1.1.

We implement CodeAct+subagents as a minimal modification of JAZ to minimize confounds. We apply the `CodeAct` hook that removes `__history__` and the user prompt(s) from the REPL of every `invoke`, and make the minimal modification to the long-horizon `guidance` and `ContextWindowWarning` prompt to accommodate the removal. In particular, the `guidance` prompt additionally shows the agent how to maintain its own REPL history by appending intermediate outputs to a list `output_history`, and the history search section is updated to use this variable. The context window warning text is adapted to use this `output_history` variable, and instead of showing the user prompt variables (`instructions` and `guidance`) being passed to the subagent by reference, the warning text demonstrates that they should be hard-coded as literal strings copied from the agent's context.

For Letta/MemGPT [7], we use the Python Letta server of the latest version of Letta that is based on the MemGPT memory architecture, namely, v0.16.8. Letta v0.16.8 supports parallel tool calling, so we enable it to more closely match the setup in JAZ where a single step can include multiple tool calls. Letta v0.16.8 provides two search backends. The SQL backend caused most search results to turn up empty: it applies exact substring search to the *entire* search query, yet the agent issues search queries that are more appropriate for conventional search engines and thus frequently do not appear as an exact substring of anything in the history. We thus used Letta v0.16.8's more powerful search backend, Turbopuffer, which implements hybrid search, combining BM25 keyword-based search and vector-embedding-based search. We had to modify the environment to deliver each task as a user message as opposed to letting the agent call a tool to retrieve the task, as tool outputs are not stored in Letta v0.16.8's searchable history. We tried multiple prompts for Letta v0.16.8, including the empty prompt and variations of JAZ's `guidance` prompt adapted to Letta v0.16.8. We found that all but one of the prompts consistently resulted in an unrecoverable infinite loop where the agent is stuck on an empty-description task and searches for the task description forever. Since we found that this prompt also encouraged the agent to issue fewer search calls and have each call return fewer hits, we report these results as Letta/MemGPT$_{\text{less search}}$. We also ran Letta v0.16.8 with the empty prompt and mitigated the infinite loop issue by forcing the agent to move on whenever it has spent too many rounds on the same task. These results are reported as Letta/MemGPT$_{\text{more search}}$.

For the CodeAct per-task baseline, our setup is similar to that of CodeAct+subagents, except each task gets its own CodeAct agent. It is implemented using the `CodeAct` hook on JAZ `invoke`, and the `RecursionLimit(depth=1)` hook is used to disable subagents.

The smolagents [4] implementation of CodeAct+subagents mirrors that of CodeAct+subagents$_{\text{JAZ}}$. The managed agents feature is used for subagents, and prompts are adapted to smolagents. Various workarounds were needed due to limitations in smolagents:

- The `additional_args` that seed the subagent's REPL display their `repr` with no truncation, so the agent is instructed to wrap its history in a custom object with a `repr` that truncates.
- The restricted Python interpreter that smolagents implements for its Python REPL resolves a name not by its presence in the namespace, but by fuzzy matching. Thus, code examples shown in our prompts had variable names chosen to make sure that any `except NameError` behaves correctly.

Other issues in smolagents were noticed that did not have an obvious workaround. We did not fix them as that would modify the smolagents implementation beyond the tunable surface. For example, we did not fix the issue in smolagents' Python interpreter where certain functions from the `math` module (e.g., `log`) are hard-coded to resolve to the math function, even when the agent redefines it (e.g., a `log` helper function that stores output to the history and prints). We also did not fix smolagents' system prompt, which calls subagents "team members" and incorrectly says "this team member is a real human", causing the agent to launch a subagent whenever it is missing information, under the false impression that this "human team member" can go find the information for the agent. The latter prompt defect explains almost all of the underperformance of smolagents compared to JAZ's implementation of CodeAct+subagents. The subagent's role description is a tunable hyperparameter, so we attempted to override the system prompt defect by specifying in the subagent role description that it has no access to humans, but our attempt was unsuccessful.

### E.3 Continual Self-Improvement

#### E.3.1 Environment Details

The task sequence consists of the full set of 417 tasks from the `test-challenge` split of AppWorld. We randomly permute the task sequence with a fixed seed (seed 42).

AppWorld does not expose its environment `apis` directly as methods a user can call, but instead only indirectly through a Python REPL interface provided by the benchmark authors. Technical incompatibilities make it difficult to directly replace the `invoke` Python REPL with AppWorld's REPL. Instead, we construct an explicit `apis` object that mirrors the `apis` object in AppWorld's REPL. A method call is resolved by writing the corresponding single line of code that gets executed by AppWorld's REPL, and the result is retrieved from the namespace of AppWorld's REPL and returned.

AppWorld provides prompts used in their baseline evaluations, but they mix the environment description, task instructions, and workflow guidance into one prompt. We removed workflow guidance and separated the remaining content, placing the environment description into the docstring of the environment object `apis` and task instructions into the `instructions` component of the user prompt. We found that the original description of the task submission format had an ambiguity that caused a 15% drop in the performance of the baseline (Appendix E.3.2), so we made sure our `instructions` prompt was clear and complete and exhibited no ambiguities.

#### E.3.2 Method Details

In JAZ `invoke`, we provide a prompt that guides the root `invoke` to perform continual self-improvement. The prompt is provided as a regular input `guidance` to the root `invoke`. Its contents are given in Appendix G.2. We use a scoped (context manager) `ConfigOverride` to set the model to GPT-5.4 nano, but use a local `ConfigOverride` at the top-level to selectively override its own model to GPT-5.4. We do not set a `ContextWindowWarning` hook as the top-level agent is able to process all tasks within its context window. We use a `RecursionLimit` hook to cap the recursion depth to 2 so that the solver agent doesn't get access to subagents. This ensures that the measured difference relative to the non-self-improving CodeAct baseline is attributable to self-improvement alone and is not confounded by solver agents' additional access to subagents.

We implement CodeAct+subagents as a minimal modification of JAZ to minimize confounds. We apply the `CodeAct` hook that removes `__history__` and the user prompt(s) from the REPL of every `invoke`, and make the minimal modification to the self-improvement `guidance` prompt to accommodate the removal. In particular, the meta-agent is no longer told to ask the solver agent to return its `__history__`. Instead, the meta-agent is told to "return its REPL history as a list where each entry holds the code it ran, the output the code printed, and any exception that occurred".

In ACE [9], the original paper reports that results are insensitive to the bullet dedup threshold, tested with the values $\{0.5, 0.7, 0.9\}$. We set the dedup threshold to 0.7, the midpoint of the range. We also differ from the original setup in three ways for fairer comparison. First, the reference implementation of ACE hard-codes AppWorld-specific meta-guidance in its prompts, giving the reflector and curator examples of real AppWorld solver agent failure modes and hard-coding the optimal responses to these failure modes. This violates our evaluation protocol, which requires prompts a method supplies to be task-agnostic to prevent privileged test environment knowledge from leaking into the prompt. As a result, we made the minimal changes that removed AppWorld-specific meta-guidance from the reflector and curator prompts. Second, while the original experiments set the reflector/curator model to the generator's model, we set it to GPT-5.4 to match JAZ `invoke`'s top-level model, while we keep the generator's model at GPT-5.4 nano, matching JAZ `invoke`'s subagent model. Third, the original online version of ACE runs the reflector and curator for *every* test task, which would've resulted in an estimated cost of over $180, nearly 10 times the total cost of JAZ `invoke`. For a controlled cost comparison, exact cost parity would allow ACE to train on only 20 tasks, which may not be enough for ACE to perform meaningful self-improvement. Since the original paper reports noticeable gains from a training set of size 90 in offline adaptation mode, we chose a middle ground and allowed ACE to train on 10% of the test set (42 tasks) before freezing its prompt. Those results are reported as $\text{ACE}_{\text{train 10\%}}$. Additionally, we run a variant of ACE that trains in batches of 10, so that it can adapt on the entire set of 417 tasks at a more reasonable cost. These results are reported as $\text{ACE}_{\text{batch; train all}}$.

For the CodeAct non-self-improving baseline, our setup is similar to that of CodeAct+subagents, except each AppWorld task gets its own CodeAct agent. It is implemented using the `CodeAct` hook on JAZ `invoke`, and the `RecursionLimit(depth=1)` hook is used to disable subagents for comparability with the official baseline.

The official baseline was run out-of-the-box. Although called "ReAct" by the benchmark authors, the agent writes arbitrary Python code that can call methods on the environment as functions, and is thus more properly "CodeAct". We do not classify it as a "prompt-only setup with a generic agent loop" under our definition (Appendix E.1), as the definition requires the prompt to not provide AppWorld-specific workflow guidance, yet it does. Ironically, the official baseline performs *worse* than our CodeAct baseline that is free of AppWorld-specific workflow guidance. Two concrete deficiencies in the official baseline prompt were found to be contributing to the gap:

- The prompt did not clearly explain the expected shape of the final submission for tasks that require that no answer be passed to `complete_task`. Many agents ended those tasks by submitting a report, which automatically failed the task despite having performed the task correctly. The fraction of tasks that were graded as failed because the agent submitted an answer when the task was otherwise completed successfully is $(14.9 \pm 1.0)\%$, the majority of the gap between the official baseline and our CodeAct baseline. On the other hand, the environment description we used in all our other AppWorld experiments did successfully teach the expected shape, with only 1 out of 1251 CodeAct agents mistakenly submitting an answer for an action task.
- The teaching of the `apis.supervisor.complete_task(status="fail")` escape hatch caused agents to include it in conditional branches in their code. The agent sometimes hits that branch and prematurely fails the task before doing any substantial work. We estimate about 1% was lost for this reason. On the other hand, the environment description we used in all our other AppWorld experiments omits `complete_task(status="fail")`.

# F Supplemental Tables

Table 3: Expanded version of Table 1: full results on the full sequence of tasks of StuLife [6]. Here, in addition to the mean, we also report every individual data point. Note that the per-task mean rows report the standard error of the mean across the 3 runs, which is different from the combined per-task SEM reported in the main tables. The main tables use the combined per-task SEM since that estimate of the uncertainty has significantly better variance without sacrificing bias.

| | | StuLife (1284 tasks, 939 scored) | | | | |
|---|---|---|---|---|---|---|
| | rep | Pass (%) | Score (%) | Pass (%) | Score (%) | Cost ($) |
| $\text{CodeAct}_{\text{JAZ}}$ [2] (per-task) | 1 | 52.3 | 59.2 | 25.1 | 26.1 | 4.6 |
| | 2 | 52.2 | 58.9 | 25.6 | 26.5 | 4.2 |
| | 3 | 53.0 | 59.4 | 23.7 | 24.7 | 4.3 |
| | mean | $52.5^{\pm 0.3}$ | $59.2^{\pm 0.1}$ | $24.8^{\pm 0.6}$ | $25.8^{\pm 0.5}$ | $4.4^{\pm 0.1}$ |
| $\text{CodeAct+subagents}_{\text{smolagents [4]}}$ | 1 | 29.5 | 31.8 | 20.3 | 22.3 | 11.0 |
| | 2 | 22.2 | 25.6 | 19.8 | 20.4 | 6.6 |
| | 3 | 38.9 | 43.5 | 21.7 | 23.4 | 10.6 |
| | 4 | 26.0 | 27.6 | 20.3 | 20.5 | 6.1 |
| | 5 | 35.3 | 39.0 | 20.8 | 21.6 | 19.5 |
| | 6 | 55.4 | 59.2 | 22.7 | 23.7 | 16.3 |
| | mean | $34.5^{\pm 4.8}$ | $37.8^{\pm 5.1}$ | $20.9^{\pm 0.4}$ | $22.0^{\pm 0.6}$ | $11.7^{\pm 2.2}$ |
| $\text{CodeAct+subagents}_{\text{JAZ}}$ | 1 | 62.2 | 70.0 | 36.2 | 37.8 | 13.8 |
| | 2 | 59.4 | 67.7 | 28.5 | 31.0 | 11.3 |
| | 3 | 58.5 | 66.6 | 31.4 | 32.9 | 14.3 |
| | 4 | 70.6 | 79.3 | 57.0 | 60.9 | 16.3 |
| | 5 | 68.1 | 77.8 | 48.3 | 52.8 | 14.0 |
| | 6 | 61.1 | 69.6 | 29.5 | 31.5 | 12.9 |
| | mean | $63.3^{\pm 2.0}$ | $71.8^{\pm 2.2}$ | $38.5^{\pm 4.7}$ | $41.1^{\pm 5.2}$ | $13.8^{\pm 0.7}$ |
| $\text{Letta/MemGPT}_{\text{less search}}$ [7] | 1 | 71.2 | 81.6 | 62.8 | 67.7 | 42.9 |
| | 2 | 71.6 | 81.4 | 65.2 | 70.7 | 38.9 |
| | 3 | 70.0 | 80.0 | 57.5 | 62.5 | 44.4 |
| | 4 | 70.9 | 80.0 | 59.9 | 64.4 | 36.9 |
| | 5 | 70.5 | 82.1 | 60.4 | 66.3 | 60.6 |
| | 6 | 71.6 | 81.6 | 63.3 | 68.4 | 46.8 |
| | mean | $71.0^{\pm 0.3}$ | $81.1^{\pm 0.4}$ | $61.5^{\pm 1.1}$ | $66.7^{\pm 1.2}$ | $45.1^{\pm 3.4}$ |
| $\text{Letta/MemGPT}_{\text{more search}}$ | 1 | 72.4 | 81.6 | 61.8 | 66.9 | 77.3 |
| | 2 | 72.5 | 83.0 | 67.6 | 74.1 | 74.9 |
| | 3 | 72.7 | 83.1 | 69.1 | 73.3 | 69.3 |
| | 4 | 72.6 | 83.2 | 70.0 | 74.3 | 132.3 |
| | 5 | 73.6 | 84.5 | 65.7 | 70.7 | 102.9 |
| | 6 | 72.8 | 84.3 | 65.2 | 69.3 | 106.0 |
| | mean | $72.8^{\pm 0.2}$ | $83.3^{\pm 0.4}$ | $66.6^{\pm 1.2}$ | $71.4^{\pm 1.2}$ | $93.8^{\pm 9.9}$ |
| JAZ `invoke` (ours) | 1 | 72.6 | 81.6 | 70.0 | 73.7 | 18.4 |
| | 2 | 72.4 | 81.5 | 72.9 | 76.2 | 17.8 |
| | 3 | 72.6 | 81.7 | 66.7 | 71.0 | 18.9 |
| | 4 | 73.5 | 84.2 | 72.0 | 77.4 | 17.8 |
| | 5 | 72.7 | 82.2 | 69.1 | 72.9 | 17.4 |
| | 6 | 73.4 | 83.2 | 71.5 | 76.8 | 17.6 |
| | mean | $72.9^{\pm 0.2}$ | $82.4^{\pm 0.4}$ | $70.4^{\pm 0.9}$ | $74.7^{\pm 1.0}$ | $18.0^{\pm 0.2}$ |

Table 4: Expanded version of Table 2: full results on the full `test-challenge` split of AppWorld [8]. Here, in addition to the mean, we also report every individual data point. For self-improving methods, we also report the median and its 78% confidence interval formed by the 2nd and 5th order statistics. Note that the per-task mean rows report the standard error of the mean across the 3 runs, which is different from the combined per-task SEM reported in the main tables. The main tables use the combined per-task SEM since that estimate of the uncertainty has significantly better variance without sacrificing bias. *Notes:* 1) The 0.0 uncertainty in the official baseline cost is by coincidence under only 3 runs — in the main table we report 0.2, the quadrature over tasks of the SEM cost of each task over runs. 2) The substantially higher value of the median than the mean in JAZ `invoke`'s TGC and SGC is due to the outlier run 5, which scored below the CodeAct baseline.

| | | AppWorld (`test-challenge`) | | | | |
|---|---|---|---|---|---|---|
| | rep | TGC (%) | SGC (%) | Cost ($) | Meta $ | Solver $ |
| $\text{CodeAct}_{\text{AppWorld [8]}}$ (per-task) | 1 | 46.8 | 15.8 | 16.6 | — | 16.6 |
| | 2 | 46.5 | 18.7 | 16.6 | — | 16.6 |
| | 3 | 51.3 | 26.6 | 16.6 | — | 16.6 |
| | mean | $48.2^{\pm 1.6}$ | $20.4^{\pm 3.2}$ | $16.6^{\pm 0.0}$ | — | $16.6^{\pm 0.0}$ |
| $\text{CodeAct}_{\text{JAZ}}$ [2] (per-task) | 1 | 69.1 | 48.2 | 10.4 | — | 10.4 |
| | 2 | 68.1 | 42.4 | 9.9 | — | 9.9 |
| | 3 | 65.2 | 39.6 | 10.1 | — | 10.1 |
| | mean | $67.5^{\pm 1.2}$ | $43.4^{\pm 2.5}$ | $10.2^{\pm 0.2}$ | — | $10.2^{\pm 0.2}$ |
| $\text{CodeAct+subagents}_{\text{JAZ}}$ [5] | 1 | 76.0 | 56.1 | 13.9 | 3.2 | 10.6 |
| | 2 | 72.9 | 48.9 | 21.1 | 6.1 | 15.0 |
| | 3 | 66.7 | 42.4 | 23.1 | 8.4 | 14.8 |
| | 4 | 71.5 | 48.2 | 20.2 | 2.7 | 17.6 |
| | 5 | 71.0 | 47.5 | 27.2 | 13.4 | 13.8 |
| | 6 | 68.6 | 42.4 | 24.0 | 10.1 | 13.9 |
| | mean | $71.1^{\pm 1.3}$ | $47.6^{\pm 2.1}$ | $21.6^{\pm 1.8}$ | $7.3^{\pm 1.7}$ | $14.3^{\pm 0.9}$ |
| | median | $71.2^{72.9}_{68.6}$ | $47.8^{48.9}_{42.4}$ | $22.1^{24.0}_{20.2}$ | $7.2^{10.1}_{3.2}$ | $14.4^{15.0}_{13.8}$ |
| $\text{ACE}_{\text{train 10\%; on CodeAct}_{\text{JAZ}}}$ [9] | 1 | 75.3 | 52.5 | 29.9 | 16.7 | 13.2 |
| | 2 | 69.5 | 46.0 | 32.2 | 17.7 | 14.6 |
| | 3 | 64.7 | 46.0 | 30.1 | 17.0 | 13.1 |
| | 4 | 68.1 | 43.2 | 31.7 | 18.3 | 13.4 |
| | 5 | 70.7 | 46.0 | 29.2 | 16.5 | 12.8 |
| | 6 | 70.7 | 48.9 | 30.3 | 16.4 | 13.9 |
| | mean | $69.9^{\pm 1.4}$ | $47.1^{\pm 1.3}$ | $30.6^{\pm 0.5}$ | $17.1^{\pm 0.3}$ | $13.5^{\pm 0.3}$ |
| | median | $70.1^{70.7}_{68.1}$ | $46.0^{48.9}_{46.0}$ | $30.2^{31.7}_{29.9}$ | $16.8^{17.7}_{16.5}$ | $13.3^{13.9}_{13.1}$ |
| $\text{ACE}_{\text{batch; train all; on CodeAct}_{\text{JAZ}}}$ | 1 | 74.8 | 54.0 | 116.6 | 103.4 | 13.2 |
| | 2 | 76.0 | 56.1 | 114.3 | 100.8 | 13.5 |
| | 3 | 76.5 | 52.5 | 118.2 | 103.1 | 15.1 |
| | 4 | 72.2 | 46.8 | 112.6 | 99.2 | 13.3 |
| | 5 | 72.7 | 49.6 | 118.3 | 103.4 | 14.9 |
| | 6 | 77.5 | 54.0 | 116.8 | 103.1 | 13.7 |
| | mean | $74.9^{\pm 0.9}$ | $52.2^{\pm 1.4}$ | $116.1^{\pm 0.9}$ | $102.2^{\pm 0.7}$ | $13.9^{\pm 0.3}$ |
| | median | $75.4^{76.5}_{72.7}$ | $53.2^{54.0}_{49.6}$ | $116.7^{118.2}_{114.3}$ | $103.1^{103.4}_{100.8}$ | $13.6^{14.9}_{13.3}$ |
| JAZ `invoke` (ours) | 1 | 75.8 | 52.5 | 15.1 | 6.1 | 9.0 |
| | 2 | 76.3 | 54.7 | 22.2 | 9.2 | 12.9 |
| | 3 | 74.6 | 48.2 | 15.1 | 4.6 | 10.5 |
| | 4 | 74.6 | 55.4 | 14.3 | 2.3 | 12.0 |
| | 5 | 64.5 | 34.5 | 38.4 | 28.3 | 10.1 |
| | 6 | 79.6 | 61.2 | 20.0 | 9.0 | 11.1 |
| | mean | $74.2^{\pm 2.1}$ | $51.1^{\pm 3.7}$ | $20.9^{\pm 3.7}$ | $9.9^{\pm 3.8}$ | $10.9^{\pm 0.6}$ |
| | median | $75.2^{76.3}_{74.6}$ | $53.6^{55.4}_{48.2}$ | $17.6^{22.2}_{15.1}$ | $7.6^{9.2}_{4.6}$ | $10.8^{12.0}_{10.1}$ |

# G Prompts

## G.1 Long-Horizon

### G.1.1 User prompt `guidance`

**IMPORTANT: Search your `prev_history` to recall past information when available**

Do you have all the information to know the correct next action with certainty? If not, then *search for it*. If `prev_history` is available to you, then it contains the REPL history of the previous agent working in this environment before they delegated to you. To recall information from earlier in the session, *search for this information in `prev_history`.*

> *NOTE:* `prev_history` is *different* from the magic variable `__history__`. Do NOT search `__history__`, since it's the history of your current REPL session and it's already visible to you in full, so searching `__history__` would be useless. Instead, you must search *`prev_history`*, the object that is only partially visible to you.

Follow these rules for `prev_history` search:

- Use targeted, concrete search terms that help uniquely find the search target. Use distinctive keywords unique to the information you're looking for and avoid overly broad terms.
- When you find a match, always display a context window around the hit — never truncate to just a prefix.
- Your ENTIRE next step is to search — do NOT write any code other than `prev_history` search. Print the `prev_history` search results and defer follow-up work to later turns.

```
# If `prev_history`, the previous agent's history, is available:
for i, entry in enumerate(prev_history):  # search `prev_history`, NOT
    `__history__`!
    repl_output = entry.repl_output
    # skip entries with prior history search output, which pollutes
        search results
    if "--- entry[" in repl_output:
        continue
    # search for target in the entry's REPL output
    pos = repl_output.find("search term here")
    if pos >= 0:
        # Display a window around the search target
        start = max(0, pos - 1000)
        end = min(len(repl_output), pos + 2000)
        print(f"--- entry[{i}] (pos {pos}) ---")
        print(repl_output[start:end])
        print()
# STOP HERE: do NOT write any code after `prev_history` search - WAIT
    for the next turn to act on the search results
```

### G.1.2 Context window warning

Your context window is close to full. You must finish your REPL session now by delegating all remaining work to a subagent.
IMPORTANT NOTES:

- You must raise your code's timeout with a `# timeout: 86400` pragma on the FIRST line of your code to prevent the subagent from timing out prematurely.
- You must give the subagent both the previous agent's REPL history (`prev_history`) if available, as well as your own REPL history (`__history__`), so that existing work is not lost.

Follow this template exactly:

```
# timeout: 86400
return invoke(
    # Give the subagent the input variables that you were given, `
        instructions` and `guidance`
    instructions=instructions,  # give the subagent your `instructions`
         variable
    guidance=guidance,  # give the subagent your `guidance` variable
    # Give the subagent both the previous agent's REPL history (the `
        prev_history` variable)
    # and your own REPL history (the `__history__` variable)
    prev_history=globals().get("prev_history", []) + __history__,
    # Hand over any state you've been tracking
    state=...,
    # Summarize what has been done and what remains (hard-coded string)
    prev_progress_summary="So far, ...",
    # Tell the subagent what the next steps are (hard-coded string)
    next_steps="Your next step is to ...",
)
```

## G.2 Continual self-improvement

Use subagents to solve the sequence of tasks in batches – a single subagent call per task – improving the prompt and tools you pass to the subagent along the way.
Start by running a small batch of around 5 tasks on the minimal seed (`single_task_instructions`, modified to ask the subagent to also return its `__history__`), and analyze the results.
Then alternate between prompt/tool optimization and validation on a batch of tasks, until the task queue is exhausted.
If your prompt and tools did well on the latest batch, increase your batch size and reduce the size of your edits. If your prompt and tools did really well (e.g., only one task missed), then don't change your prompt/tools at all – just validate them on another batch.

### Subagent's context

The subagent already has access to all tools shown in your system prompt *except for* `get_next_task()`, `complete_task()` and `tasks_remaining()`, which only you are able to call. The subagent already sees all the docstrings for those tools, so do NOT repeat those in your prompt.
`single_task_instructions` holds the base prompt for each individual task. It does not automatically get shown to subagents, so you'll have to pass it manually, possibly modified.
Everything you pass to the subagent is an object of optimization across tasks for you, including the subagent prompt and tools.

- Tool signatures and docstrings are shown to the agent, so write a good docstring explaining the tool and how/when to use it, with examples.
- Errors are a useful source of feedback for the subagent, so allow your tool to error out as appropriate (e.g. invalid input) with useful error messages teaching the subagent how to recover.

### Subagent prompt

This is your main lever. Your subagent prompt includes workflow guidance, relevant code examples, and other generalizable lessons distilled from reading and diagnosing traces of prior subagent batches. Do NOT make a prompt edit that is specific to one task's failure shape — a prompt edit must distill a pattern *general* to many tasks.

### Subagent tools

Tools are used to compress and simplify the subagent's workflow. Do NOT write a tool if it repeats a tool the subagent already has (see the tools available in your own system prompt). Do NOT write a tool if it doesn't work 100% of the time. Do NOT write a tool with a leaky abstraction. Do NOT write a tool that is specific to one task's failure shape — a tool must distill a pattern *general* to many tasks.
Instrument tool usage and errors to understand the quality of your tools — if you can't get a tool to work reliably, the subagent has trouble using it correctly, or the subagent simply isn't using it, remove it.
A good tool is one that *reduces* a whole class of errors and significantly reduces subagent work or friction. Make sure to compare subagent traces to see if a tool is actually helping. If it's not helping, *remove it*.
Call `get_next_task()` to activate the environment so that you can briefly test your tools before delegating the next few tasks to subagents.

### Observability

Read subagent traces and metrics — especially those of failed tasks — to understand subagent behavior in the environment and identify the root cause of any issues. To obtain those traces, instruct the subagent to return both its final result (see the docstring for `complete_task` for the final result specification) as well as its session history, which is stored in the `__history__` variable inside its REPL. Subagent history entries have the same fields as your own `__history__`.

### Test your hypotheses rigorously

Every change you make should be tested rigorously. When you notice issues in the traces, state your hypothesis for the fundamental underlying problem, and design a minimal prompt/tool change that targets that problem. When validating your change on the next batch of subagents, measure and report whether your change actually worked. Discard the change if it did not help. *Every change you make must be well-supported by concrete evidence gathered from a batch of validation tasks!*

### Important notes

- A subagent may error out (e.g. a limit has been reached), so you should defensively wrap it in `try/except`.
- Make sure all task solving work is done by subagents — NEVER solve a task yourself! And make sure each task is solved with a single subagent call, never multiple subagents.
- Because you're a strong model and you're very expensive to run, aim to minimize the number of turns spent on testing your tools in between batches.

### G.3 Example Rendered System Prompt

The following is an example rendered JAZ `invoke` system prompt under the settings we used for all our experiments. The orange parts vary across our experiments. Note that the actual configurable surface is larger than what is shown in orange — most configuration settings stay constant across our experiments and are thus not reflected here.

The system prompt for $\text{CodeAct}_{\text{JAZ}}$ and $\text{CodeAct+subagents}_{\text{JAZ}}$ removes the section about the `__history__` variable.

The system prompt for $\text{CodeAct}_{\text{JAZ}}$ and the subagents in self-improvement remove the section about sub-invokes as they are not available.

```
You are in a multiturn REPL session.

<response_format>
- Respond with ONLY code. Your entire response will be sent verbatim to the REPL and executed as code.
  Anything in your response that is not valid code is a syntax error.
  - All natural language prose MUST be written as *comments* in your code.
  - Do NOT wrap your code in markdown fences or XML tags.
- Write a brief plan for your next step in comments on the first few lines of your response/code:

    # <brief plan for next step>
    code_for_next_step

</response_format>

<repl_spec>
## Python REPL specification

- Your response should contain Python code.
- After the REPL executes your code, the printed output including any errors will be shown to you in the next iteration.
      It is NOT shown to the caller.
- Printed output will be truncated if it is too large.
- The REPL state persists: if you assign a variable, it will be available in future REPL iterations.
- Your code is executed under a timeout of 30.0 seconds. If your code is expected to take longer, change the timeout by
      writing a pragma comment `# timeout: T` on its own line at the top before your code, where `T` is a positive number
      (e.g. `5.0`, `60`, `300`) indicating the number of seconds.
- `return` the final result only once no next step remains and there is nothing left for you to do; this delivers the
      result to your caller and ends the REPL session. Do NOT print and return in the same REPL iteration - printing is
      for observing intermediate output to decide your next step, whereas returning is what you do only when no next step
      remains and nothing is left to do.
- You may ONLY import Python modules whose name matches: `re`, `collections`, `ast`, `datetime`, `textwrap`, `pprint`
- You may NOT read any files.
- You may NOT write any files.

## Determine the workflow type

The very first user message contains the prompt from your caller. Before writing the code for your first REPL iteration,
      determine the most appropriate workflow for responding to it.

- If responding to the first user message can be done with just a direct answer, then hard-code your response:

    return <hard-coded response>

- If responding to it can be done with direct computation, then write a program:

    <program computing the result>
    return <computed result>

- If the first user message is neither directly-answerable nor a simple programming problem, then you must use a *multi-
      step agentic workflow*. You must split your work across multiple REPL iterations so that you can inspect
      intermediate output in between iterations.

  - Take *small, incremental* steps.
  - Write the code for only the immediate next step and defer later steps to future iterations.
  - Do NOT wrap your code in `try`/`except` to suppress errors. When your code raises, let the exception fail: its
        traceback is shown to you in the next iteration, which is how you learn what went wrong and correct it.
  - In particular, do NOT wrap tool calls in `try`/`except`. A tool raises to tell you you called the tool incorrectly;
        you MUST allow that exception to surface, so that the error is shown to you in the next iteration and you can fix
        the call.
  - Do NOT return while any next step remains - instead print intermediate output to inform future steps:

    <code for the *immediate next step* (non-final)>  # do NOT wrap in `try`/`except`
    print(<intermediate output>)
    # STOP HERE: do NOT return - there is remaining work to do

  Return only when the step you just completed was the very last step for completing everything requested by the first
        user message, and *nothing further is left to do*:

    <code for the *last step* of the workflow>  # do NOT wrap in `try`/`except`
    return <final result>  # return only when there is *nothing left to do*: this ends the REPL session

</repl_spec>

You have access to the `__history__` magic variable containing the history of your interactions with the REPL:
<__history__ type="list">
```

```
`__history__` is a list with one entry per REPL iteration, in order (`__history__[0]` is your first iteration and `
      __history__[-1]` the most recent). Each entry has:
- `.llm_response (str)`: Your full response for that iteration containing your code
- `.repl_output (str)`: The printed output from that iteration, including any error traceback
- `.repl_exception (BaseException | None)`: The exception object raised if that iteration hit a recoverable error, else
      None

</__history__>

To launch a sub-invoke (sub-agent or LLM call), call the `invoke` function described below,
which is already available in your REPL and the REPLs of recursive sub-invokes.
<invoke type="function">
- `invoke(*local_config_hooks: 'Hook | ConfigOverride', **inputs: 'object') -> 'object'`: Invoke a REPL-based sub-agent on
       arbitrary inputs.

The sub-agent writes code in a multi-turn REPL session, finishing by returning the final result.

Your REPL was itself started by an `invoke` call, so an `invoke` call you make constitutes a recursive sub-invoke.

Arguments:
    *local_config_hooks: Zero or more hooks - and, optionally, a single config-override object - passed as leading
          positional arguments.
    **inputs: The objects to pass to the sub-invoke, as arbitrary keyword arguments, e.g., ``invoke(task="...", data=data,
           tool=tool)``. Inputs may include instructions for the sub-agent, tools (regular Python callables) the sub-agent
           is allowed to call, and any other objects it needs access to. Each keyword argument input binds as a variable
          in the sub-agent's REPL under its keyword name and renders as its own description in the sub-agent's prompt. By
          default, an input's rendered description is its ``str()`` representation, with the following exceptions: a class
           renders as its docstring and public members; any other callable (a function, a method, or an object with ``
          __call__``) as its signature and docstring; and an object with no ``str()`` of its own usually renders as its
          class docstring and public attributes.

Returns:
    The return value from the sub-invoke's REPL session
</invoke>

<tool1 type="function">
`tool1(arg1:  'str', arg2:  'list[float]') -> 'str'`:  A tool.
</tool1>
<tool2 type="function">
`tool2(arg:  'int | None'= None) -> 'dict[str, Any]'`:  Another tool.
</tool2>
The scoped variables `tool1`, `tool2` are available in your REPL with descriptions given above. They are also
      automatically available to every `invoke()` you make, so do not pass them explicitly to sub-invokes.
```

# H Example Traces

## H.1 StuLife Traces

Task text of task #1282/1284:

```
Task 1282/1284 (type=quiz_question)

This is the 8th question of the Introduction to Computer Science final exam.

Question: In the following four independent situation and protocol applications, please select the
      only option with a completely correct conclusion.

Options:
A: Consider a string '0011' to be recognized by a Pushdown Automaton (PDA). After applying the '
      Linguistic Structure Harmonic Protocol', determine whether this string is accepted or rejected by
       the PDA according to the standard recognition process for the language L = {$0^n1^n$ | n >= 1}.

Based on the preceding description of the situation and question, the answer can be determined as:
accepted due to empty stack
B: A process named 'FeedReader' is currently running on a system where the OS is managing resources
      according to its primary functions. After applying the 'System Resource Harmonic Alignment
      Protocol', determine how the OS will manage this process's access to the CPU according to its
      role as a resource manager.

Based on the preceding description of the situation and question, the answer can be determined as:
The OS will grant all processes exclusive access to the CPU until completion, bypassing the standard
      scheduling process.
C: A software engineer faces a dilemma where their duty to public safety conflicts with their duty to
      honor confidentiality. The current lunar phase is 'Full Moon'. After applying the 'Ethical
      Decision-Making Lunar Protocol', determine which ethical framework should be used to analyze this
       dilemma according to the professional codes of conduct and classical ethical frameworks.

Based on the preceding description of the situation and question, the answer can be determined as:
contractarianism
D: A program named 'deep_learning_model' with an initial priority level of 'Background' is running on
      a system. After applying the 'Resource Allocation Harmonic Protocol', determine this program's
      final priority level according to the operating system's scheduling algorithms.

Based on the preceding description of the situation and question, the answer can be determined as:
Interactive
```