The World Won't Stay Still: Programmable Evolution for Agent Benchmarks

Guangrui Li [1] **Yaochen Xie** [*1] **Yi Liu** [*1] **Ziwei Dong** [*1] **Xingyuan Pan** [*1] **Tianqi Zheng** [1] **Jason Choi** [1] **Michael Morais** [1] **Binit Jha** [1] **Shaunak Mishra** [1] **Bingrou Zhou** [1] **Chen Luo** [1] **Monica Cheng** [1] **Dawn Song** [2]

Abstract

LLM-powered agents fulfill user requests by interacting with environments, querying data, and invoking tools in a multi-turn process. Yet, most existing benchmarks assume static environments with fixed schemas and toolsets, neglecting the evolutionary nature of real-world and agents' robustness to environmental changes. In this paper, we study a crucial problem, how to evolve the agent environment in a scalable and controllable way, thereby better evaluating agents' adaptability to real-world dynamics. We propose PROEVOLVE, a graph-based framework that makes environment evolution *programmable*. At its core, a typed relational graph provides a unified, explicit representation of the environment - data, tools, and schema. Under this formalism, adding, removing, or modifying capabilities are expressed as graph transformations that coherently propagate updates across tools, schemas, and data access. Building on this, PROEVOLVE can 1) program the evolutionary dynamics as graph transformations to generate the environments automatically, and 2) instantiate task sandboxes via subgraph sampling and programming. We validate PROEVOLVE by evolving a single environment into 200 environments and 3,000 task sandboxes, and benchmark representative agents accordingly.

1. Introduction

LLM-powered agents interact with an environment — external data resources and tool interfaces — in which agents reason over external knowledge, make plans, and invoke tools in an interleaved manner (Yao et al., 2023; Barres et al., 2025; Shridhar et al., 2020; Yang et al., 2018). Despite recent progress, most existing approaches typically evaluate agents in *static* environments, characterized by fixed toolsets and fixed data schemas that specify accessible fields and entities in a deterministic fashion (Liu et al., 2023; Qin et al., 2023; Mialon et al., 2023; Zhou et al., 2023; Jimenez et al., 2023). This assumption conflicts with real-world deployment, where environments evolve continuously and progressively: new capabilities are introduced incrementally, existing tools are iterated, and outdated tools are gradually deprecated. As a result, this gap inevitably hinders the comprehensive assessment of agents' adaptability to environment dynamics.

Recently, there has been growing interest in scaling agent environments to better evaluate agents' capabilities. However, previous work largely adopts simplified paradigms, scaling along a single axis (e.g., more tools or data) (Li et al., 2024; Shi et al., 2025; Yao et al., 2024; Chen et al., 2024) or focusing on curated discrete domains (Zhang et al., 2025). Although these approaches increase environmental variation, they fail to endow environments with evolutionary dynamics, as each environment is treated as an isolated snapshot. In addition, these approaches neglect the *coherence* among the environment components— tools, data, and schemas —which are typically tightly integrated. For example, when adding an order-cancellation tool to an existing order-placement system, one would not typically design a new data schema from scratch. Instead, the cancellation functionality builds upon the existing order schema, data entities, and tool interfaces, which evolve jointly in a coherent manner.

In this paper, we move beyond static collections of environments and introduce *ProEvolve*, a framework that explicitly models the environment and evolve it in a programmable manner. By capturing structured transitions between successive environment versions, our framework enables programmable and automatic environment generation and task generation, paving the way for evaluating agents' adaptability under environmental change.

First, we investigate a crucial question: *how to evolve environments in a controllable manner?* We identify two core challenges: 1) **Scalability versus Coherence**: scaling up environment elements while preserving coherent dependen-

[*]Second authors with equal contribution. [1]Amazon [2]UC Berkeley. Correspondence to: Guangrui Li <guangrli@amazon.com>, Yi Liu <yiam@amazon.com>.

Preprint. March 6, 2026.

cies among them, and 2) **Dynamics versus Controllability**: generating evolutionary dynamics without sacrificing system control. Taking a step further, we reveal the underlying obstacle: since the environment elements and their relationships are not explicitly modeled, it becomes intractable to evolve them in a coherent and controllable manner.

Second, to tackle the identified obstacle, we propose PROEVOLVE, a graph-based framework that models and evolves environments in a programmable and automated manner. Specifically, we design an environment graph where environment elements— schemas and data — are represented as nodes, with their relationships encoded as edges. Under this formalism, environment evolution is formulated as graph transformations, while task sandboxes can be cast as conditioned subgraph sampling. PROEVOLVE then employs agentic workflows to: (1) evolve and generate new environment variants, and (2) synthesize task sandboxes that adapt to dynamic environmental changes. Importantly, this process operates automatically while maintaining fine-grained control over evolution trajectories, environment complexity, and task difficulty.

To validate the proposed framework, we start from a seed e-commerce environment and systematically evolve it into 50 evolution trajectories comprising 200 environment variants in total, along with 3,000 environment-specific tasks instantiated in corresponding sandboxes. We then benchmark representative LLM agents in these dynamically evolving environments to investigate their adaptability and robustness in the face of continual shifts in schemas, tools, and available resources.

The contributions of this paper can be summarized as:

- We identify the fundamental challenge in moving from static to evolving environments and tackle it by introducing a programmable graph formalism that explicitly models environment evolution.
- Built upon the graph formalism, we propose a framework to 1) automatically evolve environments and implement them, 2) compose task sandboxes and evaluate them within those evolved environments.
- We validate the proposed framework via evolving a single e-commerce store environment into 200 environments and 3,000 task sandboxes. Then we benchmark representative agents, and conduct preliminary research regarding agents' adaptability under environmental change. The results reveal gaps in how agents adapt to an environment when it becomes dynamic.
- To the best of our knowledge, this work presents the first explicit formulation of *agent evaluation in evolving environments* as a standalone research problem, providing a systematic methodology for generating controlled evolution trajectories to examine agent robustness.

2. Related Work

Existing agent evaluation paradigms predominantly assess agents in *static* environments with fixed schemas and toolsets. These benchmarks are based on environments that are snapshots of real-world applications or constructed via substantial human effort. SWE-bench (Jimenez et al., 2023; Yang et al., 2024) and LiveCodeBench (Jain et al., 2024) are widely adopted to evaluate software engineering agents, while OSWorld (Xie et al., 2024) and WebArena (Zhou et al., 2023) evaluate multimodal control in fixed operating systems and web environments. Generalist benchmarks such as AgentBench (Liu et al., 2023), ToolBench (Qin et al., 2023), and GAIA (Mialon et al., 2023) assess capabilities in diverse, heterogeneous domains—ranging from database management to household tasks.

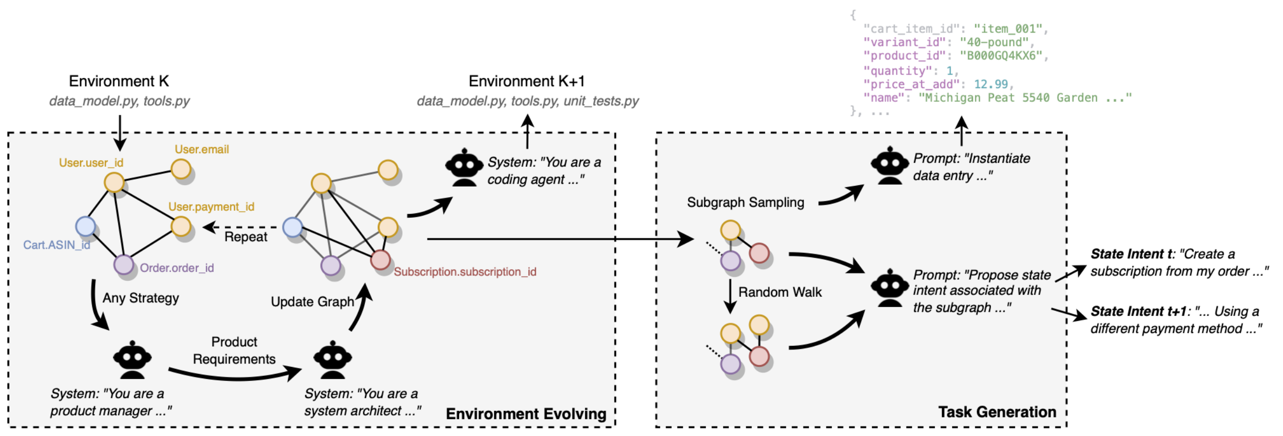
To address scalability and diversity, recent work has shifted toward synthetically generating tasks (Xie et al., 2025) and environments. Systems like AutoBencher (Li et al., 2024) and TaskCraft (Shi et al., 2025) keep the underlying tools constant (e.g., standard APIs) but use LLMs to synthesize a massive volume of increasingly complex user queries, effectively scaling the data without changing the environment's rules or tools. Instead of static Q&A, τ-bench (Yao et al., 2024) employs active user simulators to test if agents can handle dynamic conversations, while τ^2-bench (Barres et al., 2025) introduces "dual-control" mechanics where the user actively interferes with the environment state (e.g., clicking buttons while the agent speaks). Rather than sticking to one theme, AUTOENV (Zhang et al., 2025), automates the creation of entirely new environments with distinct rules and physics (e.g., generating 36 different games and puzzles) to test the agent's ability in adapting across heterogeneous worlds.

To our knowledge, ToolQA-D (Chen et al., 2024) is the only existing benchmark that explicitly addresses tool variability. Specifically, ToolQA-D employs GPT-4 to modify API usages, including names, parameters, and response formats, to evaluate robustness.

3. From Static to Evolving: Challenges

Humans live and interact with the dynamically changing world, agents are no exception: the environments they interact with evolves and changes in a continuous and evolving manner. For example, an e-commerce store regularly introduces new capabilities, incrementally upgrades existing ones, and occasionally deprecates outdated features.

In this paper, we study how to imitate real-world environ-



```
{
  "cart_item_id": "item_001",
  "variant_id": "40-pound",
  "product_id": "B000GQ4KX6",
  "quantity": 1,
  "price_at_add": 12.99,
  "name": "Michigan Peat 5540 Garden ..."
}, ...
```

Environment K
data_model.py, tools.py
Environment K+1
data_model.py, tools.py, unit_tests.py
User.user_id
User.email
User.payment_id
Cart.ASIN_id
Order.order_id
Repeat
Subscription.subscription_id
System: "You are a coding agent ..."
Any Strategy
Update Graph
Product Requirements
System: "You are a product manager ..."
System: "You are a system architect ..."
Environment Evolving
Subgraph Sampling
Prompt: "Instantiate data entry ..."
Random Walk
Prompt: "Propose state intent associated with the subgraph ..."
State Intent t: "Create a subscription from my order ..."
State Intent t+1: "... Using a different payment method ..."
Task Generation

Figure 1. **End-to-end workflow of programmable environment evolution and graph-grounded task instantiation.** Environment graphs are evolved via programmable graph edits and translated into executable code (left). Tasks are then generated by sampling subgraphs and materializing state-wise user intents and data into runnable sandbox instances (right), enabling controlled evaluation under evolving environments.

ment evolution, deriving more realistic and controllable environments for benchmarking agents' adaptability and robustness under dynamic changes.

Then a question naturally arises:

How to evolve environments in a controllable manner?

This question becomes central as we move beyond static benchmarks toward supporting scalable, continuously evolving environments. We identify two fundamental challenges:

Challenge I: Coherence versus Scalability. Environment updates usually involve the *co-evolution* of multiple components in a coherent manner— new schema fields and entities, new relations, and the corresponding tools, rather than isolated modifications to a single component. Meanwhile, to assess robustness, a benchmark must encompass diverse evolution patterns and a broad range of environments. Naïve scaling, such as independently adding tools or data, may disrupt system coherence, whereas manually constructing coherent evolutions does not scale.

Challenge II: Dynamics versus Controllability. Realistic environments evolve over time, with dynamics such as tools being added or deprecated, schemas being updated, and APIs changing. For benchmarking, these changes must be *controllable*: the evolution trajectory should be explicit and reproducible, and the evaluation protocol remain well-defined throughout the evolution process.

These challenges motivate us to propose a programmable framework for evolving environments in a controllable and scalable manner, which we present in the following.

4. ProEvolve: Programmable Evolution for Agent Benchmarks

In the following sections, we first devise a principled way to model the environment explicitly, thereby enabling programmable evolution, as discussed in Sec. 4.1. Building upon this formalism, we then propose an integrated framework to evolve environments in a programmable manner: (i) Programming environments to evolve in a coherent manner (Sec. 4.2) (ii) Programming sandboxes and tasks with evolved environments (Sec. 4.3), and (iii) Evaluation design along the evolution trajectories in Sec. 4.4. We present an overview of the workflow in Figure 1.

4.1. Graph Formalism for Environment Modeling

To combat the aforementioned challenges, we take a step further to identify a fundamental obstacle that impedes effective environment evolution: the lack of comprehensive and explicit environment modeling. Without explicitly capturing the relationships between environment elements, it becomes intractable to evolve them in a coherent and controllable manner.

In light of this, we introduce a graph formalism that models environment elements and their relationships in an explicit, integrated representation. This choice is motivated by two merits: (i) a graph provides a unified space to encode schemas, data entities, and tools alongside their relations, and (ii) it allows environments to evolve via progressive graph transformations, enabling controlled updates to capabilities while preserving structural integrity.

Concretely, we represent an environment version through a typed relational graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ corresponds to a schema element (e.g., `User.user_id`,

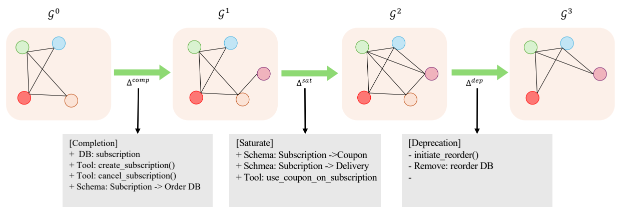


Figure 2. **Programmable environment evolution via graph transformations.** Starting from a seed environment graph $\mathcal{G}^0$, we generate a curriculum of environments $\mathcal{G}^1, \mathcal{G}^2, \mathcal{G}^3$ by applying explicit edit operators (arrows; e.g., component onboarding, schema/tool updates, and dependency rewiring), which add/remove nodes and edges in a coherent manner. This yields controlled environment dynamics while preserving a unified representation for task generation and evaluation across versions.

`Order.order_id`), and each directed edge $e = (v \to v') \in \mathcal{E}$ denotes a typed relation or a tool-enabled transition that maps information from a source schema element to a target schema element.

As such, environment evolution becomes a sequence of graph transformations:

$$\mathcal{G}^{(0)} \xrightarrow{\Delta^{(1)}} \mathcal{G}^{(1)} \xrightarrow{\Delta^{(2)}} \cdots \xrightarrow{\Delta^{(K)}} \mathcal{G}^{(K)}, \qquad (1)$$

where $\mathcal{G}^{(0)}$ is a seed environment and each $\Delta^{(k)}$ is an *evolution strategy* that applies structured operations mirroring real-world environmental change. Fig. 2 provide an more intuitive example, in which environment evolution is expressed as sequence of structural graph operations. For instance, supporting a new capability such as *subscription* requires connecting it to existing schemas (edge additions) and introducing new data entities and attributes (node additions).

Built upon the graph formalism, we answer three questions in the following sections: 1) how to evolve environment and compose a new one? 2) how to generate task sandboxes with evolved environments? and 3) how to evaluate under evolving environments?

4.2. Programming to Evolve Environment via Graphs

Whereas graph formalism enables *controllable* evolution through graph transformations (Eq. 1), manually composing coherent graph transformation $\Delta^{(k)}$ for diverse evolution patterns does not scale. To enable scalable and diverse environment generation, we introduce an agentic pipeline which programs evolutions *automatically* with two phases:

Phase I: Evolution Proposal: An LLM agent will traverse the environment graph to make a reasonable transformation plan based on predefined transformation strategies.

- **`Completion`** (Δ^{comp}): Adds nodes and edges to support new capabilities. An LLM first proposes a feature the current $\mathcal{G}^{(k)}$ cannot support (e.g., "Add-to-wishlist functionality"), then designs the required schema extensions (new node for `Wishlist.`) and tool-enabled transitions (new edges with tools like `add_to_wishlist`). This imitates feature-driven development in the real world.

- **`Saturation`** (Δ^{sat}): This strategy discovers indirect relationships via random walks on $\mathcal{G}^{(k)}$ and adds "shortcut" edges. For example, a 3-hop path `User.user_id`→`Order.order_id`→`Product.product_id` can be collapsed into a direct tool `get_user_purchased_products`. Given multiple random walk paths as seeds and candidates, an LLM selects the most valuable shortcuts based on business utility, creating efficiency-enhancing tools that combine multiple existing operations.

- **`Deprecation`** (Δ^{dep}): Removes nodes and edges to simulate API deprecation or service outages. The strategy samples candidates (e.g., peripheral edges, entire databases, cross-database bridges) via graph-theoretic criteria, then uses an LLM to select realistic deprecations with appropriate challenge levels and workarounds (e.g., "Address Book service undergoing scheduled maintenance; to workaround, prompt the user to provide their shipping address at checkout").

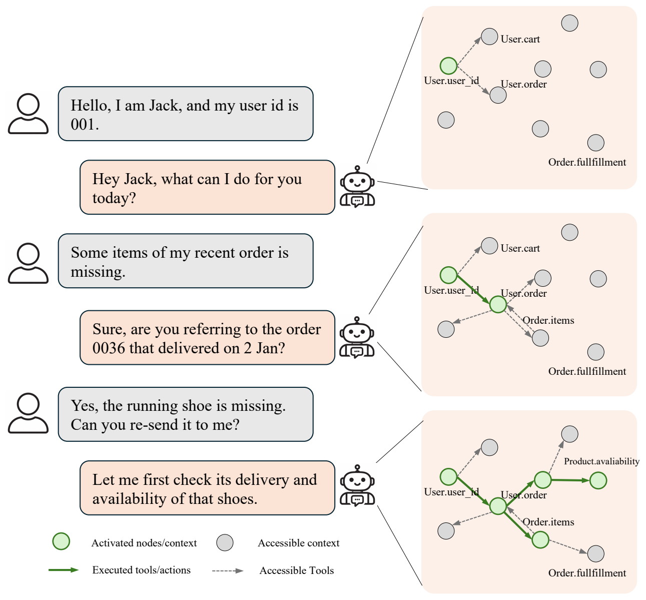
The three strategies can be flexibly composed to generate diverse evolution curricula. By applying them in different orders, with varying parameters, and multiple times, we can program evolution patterns with different types and complexity levels.

Phase II: Implement and Validate: To instantiate transformed graphs as executable code, we employ an LLM-based *Coding Agent*. Given the evolved graph $\mathcal{G}^{(k+1)}$ and the seed environment, the agent generates updated coherent implementations, consisting of data models, tool implementations, and unit tests. The code files are generated in the following order

1. **Test specification generation.** The agent derives test specifications from graph semantics, encoding the expected schema constraints, tool behaviors, and cross-entity relations induced by $\mathcal{G}^{(k+1)}$.

2. **Implementations.** Based on the test specifications, the agent generates or updates schemas and tool implementations to satisfy the specifications.

3. **Test materialization.** Depending on the outcome of previous steps, unit tests are instantiated from the specifications and grounded in the generated implementation.


Hello, I am Jack, and my user id is 001.

Hey Jack, what can I do for you today?

Some items of my recent order is missing.

Sure, are you referring to the order 0036 that delivered on 2 Jan?

Yes, the running shoe is missing. Can you re-send it to me?

Let me first check its delivery and availability of that shoes.

Figure 3. **Context as subgraph expansion in a tool-mediated conversation.** At each turn, the environment exposes a reachable context subgraph (gray nodes; dashed arrows for reachable tool transitions), while the agent activates a subset of nodes (green) by executing tools/actions (solid arrows) conditioned on the dialogue. As the conversation progresses, executed transitions expand the active subgraph, enabling retrieval and integration of newly reachable information (e.g., from `User.user_id` to `User.order` to `Order.order_items` and downstream product attributes).

As an outcome of each evolving step $\Delta^{(k+1)}$, the workflow produces: (i) an evolved graph $\mathcal{G}^{(k+1)}$, (ii) coherent executable code (i.e., models, tools, and tests), (iii) evolution context documenting the changes, and (iv) metadata (*e.g.*, new tools, deprecated capabilities, challenge levels). The choice of strategy sequence and parameters allows us to approximate different realistic environment evolution scenarios observed in production systems. Importantly, the graph formalism ensures that all evolved environments remain compatible with the upcoming task generation and instantiation.

Quality Control. As we employ the automated workflow with coding LLMs to generate evolved environments, we perform quality control to validate all generated environments. Specifically, all environment implementations are paired with unit tests for both the data model and tools. We evaluate each environments test coverage and unit tests passing rate.

4.3. Programming Tasks as Subgraphs

Analogously to environment programming, we convert the task generation as sub-graph programming. Each task can be cast as a constrained *task subgraph* $\mathcal{H} \subseteq \mathcal{G}^{(k)}$ that specifies the structural scope (schemas/tools/relations) through which the agent must traverse to satisfy the task objective.

The task programming consists of the follow steps:

- **`Subgraph Sampling.`** We first sample a connected subgraph $\mathcal{G}_\tau \subseteq \mathcal{G}^{(k)}$ that defines the *reachable scope* of a task instance τ. Conditioned on $\mathcal{G}_\tau$, we use an LLM to synthesize (i) a task-level goal g_τ and (ii) a high-level scenario description s_τ together with *prerequisites* required by the scenario. For example, an exchange request typically presupposes that a relevant order has been delivered.
- **`Sandbox Materialization.`** Given $(\mathcal{G}_\tau, s_\tau)$, we initialize a sandbox by synthesizing prerequisite entities and linking them according to the relations induced by $\mathcal{G}_\tau$. This step ensures that tool transitions required by the sampled subgraph are well-defined and executable, while supporting diverse scenarios through controllable variations of entity attributes and cross-entity references.
- **`Agentic Walk Execution.`** Finally, we generate a reference multi-turn trajectory by traversing and expanding along $\mathcal{G}_\tau$. At each state t, an oracle policy selects a frontier set of newly reachable nodes (e.g., identifiers or attributes enabled by the current context) and uses a separate LLM to render a state-wise user instruction $u_t^\star$ that motivates progress toward the goal g_τ, as illustrated in Fig. 3. The oracle agent executes tool calls a_t, obtains outputs $\mathbf{y}_t$, and we record both (i) the actions taken and (ii) the nodes/edges activated by the transition. Tool outputs are converted into newly obtained facts $\Delta\widehat{\mathcal{K}}_t$ and accumulated:

$$\widehat{\mathcal{K}}_t = \widehat{\mathcal{K}}_{t-1} \cup \Delta\widehat{\mathcal{K}}_t, \qquad \Delta\widehat{\mathcal{K}}_t = \phi(\mathbf{y}_t). \quad (2)$$

  Meanwhile, the turn-level context is represented as a subgraph $\mathcal{A}_t \subseteq \mathcal{G}_\tau$, which expands as the oracle executes actions and the dialogue reveals identifiers:

$$\mathcal{A}_t = \text{Expand}\Big(\mathcal{A}_{t-1};\ a_t,\ u_t^\star,\ \mathcal{G}_\tau\Big), \quad (3)$$

  yielding a sequence of structured expansions $\mathcal{A}_0 \rightarrow \mathcal{A}_1 \rightarrow \cdots \rightarrow \mathcal{A}_T$.

Sandbox Output. Each task instance τ produces a self-contained sandbox comprising: (1) an initial state $\mathcal{S}_0$ that materializes prerequisite entities and their cross-references as dictated by $\mathcal{G}_\tau$; (2) state-wise instructions $\{(u_t^\star, \mathcal{Y}_t^\star)\}_{t=1}^T$ pairing user utterances with graph-derived success criteria; and (3) a reference trajectory recording oracle actions and subgraph expansions (Eqs. 2– 3). Because the graph formalism comprehensively captures dependencies and relationships among environment elements, sandboxes can be synthesized for arbitrary scenarios under any evolved environment, enabling controlled and diverse evaluation without manual curation.

4.4. State-Wise User Simulation and Evaluation

Following the agentic-walk procedure above, we model a multi-turn task τ as a sequence of *state-conditioned* interactions. At each state, we attach a user instruction u_t that targets the next frontier of information exposed by the current task subgraph, ensuring the dialogue progresses through dependency-consistent requests. To instantiate such interactions, we adopt a *user simulator* (in the spirit of Barres et al. (2025)) in which the "user" follows these *stated instructions* rather than relying on free-form utterance sampling.

State instruction and success criterion. For each state t, we construct a state-wise instruction tuple

$$\mathcal{I}_t = \left(u_t^\star, \mathcal{Y}_t^\star\right), \tag{4}$$

where $u_t^\star$ is the synthesized customer utterance and $\mathcal{Y}_t^\star$ specifies the *state success criterion*—the minimal set of node facts that must be obtained to satisfy the request at state t. In practice, $\mathcal{Y}_t^\star$ is derived from the agentic-walk frontier targets together with the corresponding extracted facts in the reference trace $\widehat{\mathcal{K}}_t^\star$ (Eq. 2), ensuring that each state is grounded in explicit, graph-derived supervision.

Progression rule. The user simulator enforces *sequential dependencies*: it advances from state t to $t{+}1$ only if the current state is judged successful. Given the agent's tool executions and response at state t, the simulator checks whether the required information $\mathcal{Y}_t^\star$ is satisfied (via schema-aware extraction from tool outputs and response text); if satisfied, it proceeds to the next instruction, otherwise it issues a follow-up clarification following $u_t^\star$ and remains at state t.

State success rate. This state-wise gating yields a direct measure of *success rate* for a trajectory. Let $s_t \in \{0, 1\}$ denote whether the agent satisfies state t under the simulator's criterion. We report overall success rate as the fraction of satisfied states:

$$\mathcal{C}(\tau) = \frac{1}{T}\sum_{t=1}^{T} s_t. \tag{5}$$

Because each s_t is defined by graph-derived, dependency-consistent requirements, $\mathcal{C}(\tau)$ provides an interpretable measure of how reliably an agent makes progress through the evolving task, beyond endpoint-only success.

Benefiting from the explicit graph representation, state-wise instructions $\{u_t^\star\}$ are *consecutive and dependency-aware* by construction: each instruction queries information that becomes reachable under the current subgraph expansion (Eq. 3). This makes the simulated user behavior stable and reproducible, and enables turn-level evaluation without relying on ad-hoc heuristics.

Table 1. Details of Evolved Environments

Benchmark	# of Tools	# of Environments	# Task	Schemas
Seed Env.	51	1	–	64
Evolved Envs.	384	200	3000	201

More importantly, our evaluation paradigm avoids reliance on brittle, explicit tool annotations and instead emphasizes *information efficiency*—whether the agent acquires and uses the required information to satisfy the task objective. This is particularly well-suited to evolving environments, where tools and schemas may change over time and tool-by-tool matching can become invalid despite equivalent behavior.

5. Experiments

In this section, we validate the proposed framework in an e-commerce scenario, where users interact with agents via chat to complete tasks. We construct 50 evolution trajectories comprising 200 environment versions. We benchmark representative large language model-based agents on this suite, providing a preliminary study of agents' adaptability and robustness under evolving environments.

5.1. Benchmark Generation: An E-Commerce Scenario

We construct the benchmark through the following steps:

Step 1: We curate an e-commerce store environment as the seed, instantiated with 1000 products sourced from Webshop (Yao et al., 2022), 50 synthesized users, 51 tools and 64 schemas. Then we call an LLM to compose the environment graph based on the data schemas and tools.

Step 2: Based on the seed environment, we employ the proposed automatic environment evolution pipeline (Sec. 4.2) to instantiate the evolution curriculum using the sequence:

$$\mathcal{G}^{(0)} \xrightarrow{\Delta^{\text{comp}}} \mathcal{G}^{(1)} \xrightarrow{\Delta^{\text{sat}}} \mathcal{G}^{(2)} \xrightarrow{\Delta^{\text{dep}}} \mathcal{G}^{(3)}, \tag{6}$$

to mirror typical real-world conditions, where capabilities are introduced, iteratively refined, and occasionally deprecated. We generate 50 evolution episodes, with each episode containing 4 versions, yielding 200 versioned environments in total. This setup enables a comprehensive evaluation of agent robustness to environmental dynamics.

Step 3: For each versioned environment, we use the task generation pipeline (detailed in Sec. 4.3) to generate 15 tasks, along with the state instructions for the customer simulator. The tasks are programmed with varying difficulties in equal proportion, *i.e*, easy: medium: hard = 1:1:1.

We employ `Claude-Opus-4.5` as the benchmark construction agent for the entire workflow including environment evolution, implementation, and task generation. As presented in Table 1, the proposed framework evolves one seed environment into 200 versioned environment with 384

Evolving Trajectory		$\mathcal{G}^{(0)}$	$\xrightarrow{\Delta^{\text{comp}}}$		$\mathcal{G}^{(1)}$	$\xrightarrow{\Delta^{\text{sat}}}$		$\mathcal{G}^{(2)}$	$\xrightarrow{\Delta^{\text{dep}}}$		$\mathcal{G}^{(3)}$			**Overall**	
Strategy	$\mathcal{C}^{(0)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mathcal{C}^{(1)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mathcal{C}^{(2)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mathcal{C}^{(3)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	μ_C	$\overline{T}$	$\overline{N}_{\text{tool}}$
GPT-5															
Baseline	0.564	8.1	8.1	0.646	10.3	12.8	0.771	13.2	14.9	0.454	8.7	20.6	0.609	10.1	14.1
History Replay	0.562	8.9	10.2	0.667	9.4	13.5	0.786	11.6	14.4	0.407	6.5	7.6	0.606	9.1	11.4
Reflection Replay	0.598	12.3	12.2	0.537	10.1	11.1	0.677	11.8	13.1	0.530	9.2	10.3	0.585	10.9	11.7
Claude-Opus-4.5															
Baseline	0.486	6.3	4.7	0.463	5.5	4.8	0.623	7.3	5.9	0.369	4.5	4.4	0.485	5.9	4.9
History Replay	0.542	6.7	5.3	0.540	6.9	5.7	0.540	6.8	5.5	0.358	4.1	2.3	0.495	6.1	4.7
Reflection Replay	0.438	6.2	9.6	0.558	6.7	17.3	0.501	6.3	6.2	0.336	4.7	6.6	0.458	6.0	9.9
DeepSeek-V3.2															
Baseline	0.529	7.3	6.7	0.517	7.4	6.9	0.490	8.5	6.7	0.462	9.0	7.3	0.500	8.1	6.9
History Replay	0.681	11.5	12.3	0.538	7.8	8.4	0.565	8.3	8.3	0.373	6.1	6.9	0.539	8.4	9.0
Reflection Replay	0.577	8.1	9.7	0.693	11.4	15.8	0.479	8.1	9.7	0.516	9.8	13.2	0.566	9.3	12.1
Qwen3-235B															
Baseline	0.588	10.4	9.5	0.633	12.9	13.1	0.460	9.0	8.1	0.364	7.6	7.4	0.511	10.0	9.5
History Replay	0.507	9.0	8.1	0.534	8.6	7.7	0.484	7.5	6.8	0.380	7.3	6.9	0.476	8.1	7.4
Reflection Replay	0.411	5.7	8.5	0.661	11.0	14.1	0.535	6.7	7.1	0.193	3.1	2.6	0.450	6.6	8.1
Gemini-2.5-Pro															
Baseline	0.427	7.3	3.6	0.572	8.9	5.1	0.543	8.0	3.9	0.453	8.1	3.9	0.499	8.1	4.2
History Replay	0.412	6.5	3.2	0.482	7.1	4.7	0.480	8.3	4.3	0.333	5.7	3.0	0.426	6.9	3.8
Reflection Replay	0.441	7.4	4.0	0.451	5.7	3.8	0.456	6.7	3.3	0.437	7.7	5.0	0.446	6.8	4.0

Table 2. **Evolving-trajectory evaluation across environment transformations.** We report per-version task success rate $\mathcal{C}^{(k)}$, average turns $\overline{T}$, and average tool calls $\overline{N}_{\text{tool}}$ for a fixed evolving trajectory $\mathcal{G}^{(0)} \to \mathcal{G}^{(1)} \to \mathcal{G}^{(2)} \to \mathcal{G}^{(3)}$ generated by applying three evolution strategies (arrows): Δ^{comp}, Δ^{sat}, and Δ^{dep}. Each block compares strategies (Baseline, History Replay, Reflection Replay) for a given model, and the **Overall** columns summarize mean completeness μ_C and average interaction cost aggregated over versions.

unique tools, and 201 schemas. The generated unit tests achieve 100% coverage of the modified code, and the implementation attains an overall pass rate of 90.83%. We manually reviewed the 54 failing test cases across 20 environments: 39 of these were caused by issues in the unit test implementations themselves, and 15 were non-critical failures such as invalid input checks that do not affect normal tool usage.

5.2. Experiment Setup

Agents. We use `Litellm` (Agarwal et al., 2024) to make LLM API calls, and estimate the corresponding API cost. To evaluate the generated benchmark, we testify with `gpt-5-2025-08-07` (Singh et al., 2025), `Gemini 2.5-Pro` (Comanici et al., 2025), `claude-opus-4-5-20251101` (Anthropic, 2025), `Qwen3-235B-A22B-Thinking-2507` (Yang et al., 2025), and `DeepSeek-V3.2` (Liu et al., 2025). To improve reproducibility, every task is run four times with different seeds, with temperature set to 0. Like τ-2 (Barres et al., 2025), we instruct a LLM-powered customer simulator `claude-opus-4-5-20251101` to act as the customer to chat with the agent given the state-wise instructions.

Metrics. As described in Sec. 4.4, we evaluate task with state-wise success rate. We also collect the number of tool callings and turns to examine their efficiency.

Tasks. With the generated environments, we evaluate the above agents with one evolving episode, spanning four versions of environments. Agents process all tasks within each environment version before advancing to the next version, i.e., they complete tasks 0 through n in $\mathcal{G}^{(0)}$, then tasks 0 through n in $\mathcal{G}^{(1)}$, and so on through $\mathcal{G}^{(3)}$.

Strategies for Handling Environment Evolution. We evaluate agents under three adaptation strategies to cope with environmental changes:

- Baseline: Agents handle each task independently, without any knowledge of previous environments or conversations.

- History Replay: The agent is allowed to maintain and access a memory module that stores the most recent k conversations, each corresponding to a single task and containing the user query, tool-call history, and tool-call results.

- Reflection Replay: The agent maintains a memory module that stores reflections for the most recent k conversations. For each task, a reflection is generated via an LLM call after task completion and appended to memory. Unlike History Replay, which reuses raw interaction traces (e.g., user queries and tool-call histories), Reflection Replay stores distilled summaries that abstract past experiences into higher-level guidance.

The two replay strategies provide a preliminary investigation on how past experience affects agents' behavior under continuous environment evolution. For both strategies, we

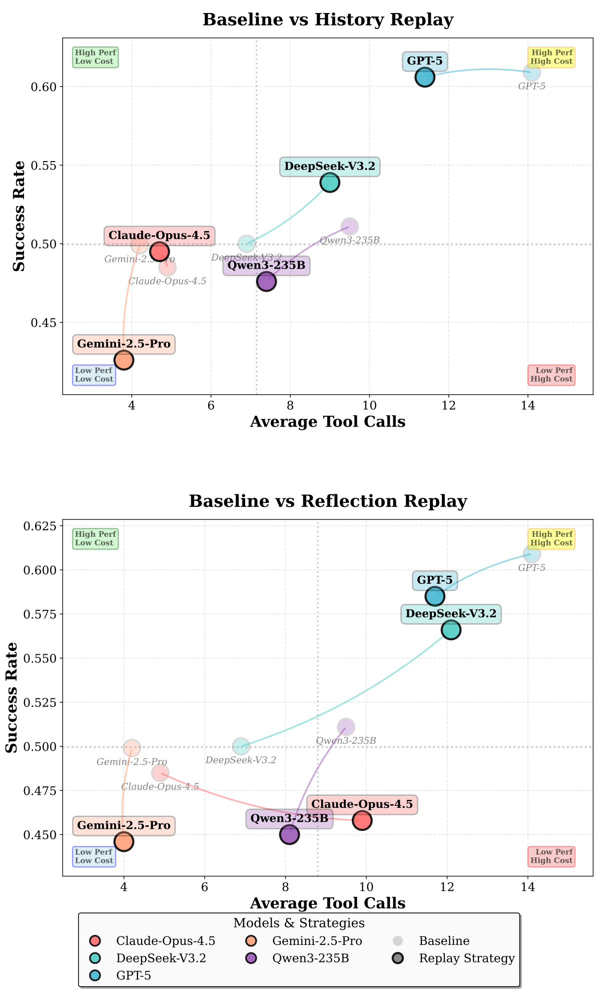


Figure 4. **Performance–efficiency trade-off of replay strategies.** Each point corresponds to a model, plotted by average tool calls (x-axis) and success rate / mean completeness (y-axis) over the evolving episode. Faint markers denote the **Baseline** strategy, while bold markers denote the replay strategy; arrows connect the same model before and after replay. **Top:** Baseline $\rightarrow$ History Replay. **Bottom:** Baseline $\rightarrow$ Reflection Replay.

set $k = 5$.

5.3. Results

Impact of Environment Evolution. In Table 2, we benchmark the performance of representative LLM-based agents under the trajectory: $\mathcal{G}^{(0)} \xrightarrow{\Delta^{\text{comp}}} \mathcal{G}^{(1)} \xrightarrow{\Delta^{\text{sat}}} \mathcal{G}^{(2)} \xrightarrow{\Delta^{\text{dep}}} \mathcal{G}^{(3)}$, in which we made the following observations:

We observe substantial environment-to-environment variability in agent performance as the environment evolves. For example, under the History Replay strategy, GPT-5's performance increases from 0.562 to 0.786 (a 40% gain) from $\mathcal{G}^{(0)}$ to $\mathcal{G}^{(2)}$, and drops from 0.786 to 0.407 (a 48% decrease) from $\mathcal{G}^{(2)}$ to $\mathcal{G}^{(3)}$. This justifies that the proposed framework indeed synthesize challenging evolution trajectories across those environment.

We do not observe a consistent performance pattern as environments evolve. For example, under the Baseline strategy, GPT-5's performance increases when nodes and edges are added (transitioning to $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$) but declines as when elements are removed (transitioning to $\mathcal{G}^{(3)}$). In contrast, DeepSeek-V3.2 exhibits a steady performance decrease throughout the evolution. The remaining agents display distinct and heterogeneous behaviors.

Analogous to performance patterns, different agents exhibit distinct tool-use behaviors. GPT-5 is substantially more tool-use intensive, whereas Gemini-2.5 Pro adopts a more conservative strategy in invoking tools. Moreover, GPT-5's tool usage increases markedly as the environment evolves, while other agents—particularly Claude-Opus 4.5 and Gemini-2.5 Pro—are less sensitive to environmental changes.

These observations underscore the importance of evaluating agents under evolving environments rather than static snapshots. The substantial environment-to-environment variability demonstrates that performance can shift dramatically under structural changes, even along a single evolution trajectory. Moreover, the absence of consistent patterns—both within and across agents—suggests that adaptability is highly transition-dependent and model-specific. Differences in tool-use behavior further reinforce this point: agents vary not only in performance but also in how intensively and efficiently they invoke tools as environments evolve, with some increasing tool usage substantially while others remain conservative and less sensitive to change. Improvements under certain modifications (e.g., capability additions) therefore do not guarantee robustness or efficiency under others (e.g., deprecations). Consequently, evaluating agents only in isolated environments may obscure brittleness, mischaracterize adaptation strategies, and overestimate generalization, whereas assessment along explicit evolution trajectories provides a more faithful measure of robustness, adaptability, and tool-use efficiency under realistic dynamics.

Impact of Replay Strategies. Comparing the two replay strategies with the Baseline in Table 2, we find that access to previous conversations or information does not consistently improve performance. This suggests that simply incorporating past experience is insufficient, leaving substantial room for developing more effective adaptation strategies to handle environment evolution.

Figure 4 shows that replay shifts models along a performance–efficiency frontier in a highly model-dependent manner. `DeepSeek V3.2` stands out as the largest beneficiary: both replay variants yield substantial performance gains, but at the cost of noticeably higher

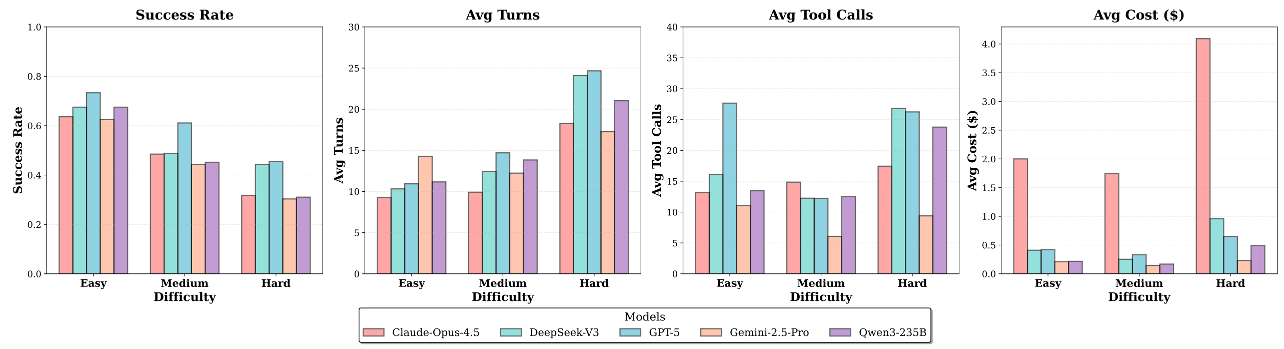


Figure 5. **Efficiency breakdown by task difficulty.** We report average tool calls, estimated cost, conversation turns, and reward for each model on *easy* vs. *hard* tasks. Harder tasks generally require longer trajectories and more tool usage

tool usage, i.e., 6.9 → 9.0/12.1 . This suggests that DeepSeek can effectively leverage prior interaction traces to improve *execution reliability* under environmental changes, albeit by issuing more consecutive tool calls. In contrast, `Qwen3-235B`, `GPT-5`, and `Gemini-2.5 Pro` exhibit broadly similar directional trends under both replay mechanisms, with only modest changes in tool calls and correspondingly performance regressions. This indicates that replay provides weaker leverage for these models—they either reuse past context conservatively or fail to translate it into more effective action sequences. Finally, `Claude-Opus-4.5` displays a stark divergence between the two replay types: *History Replay* moves it toward the desirable left-up region (higher success with comparable or lower tool usage), whereas *Reflection Replay* pushes it to the right-down region (more tool calls yet worse performance), implying a failure mode of over-exploration or miscalibrated self-correction under evolution.

Impact of Task Difficulty. Figure 5 breaks down success rate, average number of turns, average number of tool calls, and dollar cost by task difficulty (easy, medium, hard). Success rates decrease for all models as task difficulty increases. In contrast, higher difficulty generally shifts models toward larger interaction budgets—hard tasks require longer conversations and more tool use, reflecting deeper traversal and exploration in the evolving environment.

`GPT-5` maintains the strongest success rate across all difficulty levels while substantially increasing its tool usage on hard tasks (showing the largest jump in turns and tool calls), suggesting that it achieves robustness by actively acquiring additional information. In contrast, `Gemini-2.5 Pro` remains the most cost-frugal—incurring the fewest tool calls and lowest cost across difficulties—but exhibits weaker success on medium and hard tasks, indicating potential under-exploration under higher dependency complexity.

`Qwen3-235B` and `DeepSeek-V3` occupy a middle regime: both markedly increase tool calls as difficulty grows, yet achieve smaller gains in success compared to GPT-5. Finally, `Claude-Opus-4.5` is the most expensive model, with costs rising sharply on hard tasks despite only moderate success, highlighting that higher interaction cost does not ease the adaptation under environmental change.

Overall, the figure reveals a clear cost–robustness trade-off, motivating the joint reporting of efficiency and success in evolving environments. As task complexity and structural dependencies increase, agents must traverse deeper portions of the environment graph, resulting in higher interaction budgets rather than constant-cost solutions.

6. Conclusions

In this work, we identify and address a critical gap in agent benchmarks: the neglect of environment evolution. By introducing a graph-based formulation for programmable environment evolution, we enable systematic and controllable evaluation of agent adaptability. Our framework achieves two key objectives: (1) automatically generating diverse, coherent environments through graph transformations, and (2) composing and evaluating task instances within these evolving contexts. Validation in an e-commerce domain—scaling to 200 environments and 3,000 task sandboxes—demonstrates the framework's effectiveness and provides initial insights into agent robustness under environmental change. To our knowledge, this work is the first to explicitly formulate agent evaluation in evolving environments as a standalone research problem and to provide a systematic methodology for studying it.

Future work will pursue three complementary directions: (1) developing optimal adaptive agent strategies that explicitly recognize and respond to environment evolution, (2) exploring curriculum-learning approaches to design effective evolution sequences, and (3) extending the framework to diverse domains to uncover general principles for evaluating agent robustness under real-world dynamics.

Broader Impact

Positive Contributions:

This work advances the evaluation methodology for Large Language Model (LLM)-powered agents by introducing a systematic framework for assessing robustness under realistic environmental dynamics. Our contributions have several beneficial implications:

1. **Improved Agent Reliability:** By enabling rigorous evaluation of agent adaptability across evolving environments, this framework helps identify and address failure modes in production systems. This leads to more robust agents that can safely handle real-world changes in APIs, data schemas, and tool compositions.

2. **Better Benchmarking Standards:** The proposed graph-based formulation establishes a principled methodology for environment evolution, moving beyond static evaluation paradigms. This raises the bar for agent evaluation across the community, promoting the development of more adaptable and resilient systems.

3. **Accelerated Research:** The automated environment generation capability enables researchers to efficiently create diverse evaluation scenarios, reducing barriers to rigorous agent evaluation and democratizing access to comprehensive benchmarks.

Potential Risks and Mitigation:

1. **Misuse in Adversarial Settings:** The framework could theoretically be used to generate adversarial environments designed to fool agents. However, the controlled nature of our methodology and its focus on coherent, realistic evolution makes this unlikely compared to ad-hoc adversarial attacks.

2. **Over-reliance on Benchmarks:** While our framework provides more comprehensive evaluation, practitioners should not assume benchmark performance guarantees real-world robustness. We encourage complementary testing on actual deployed systems.

Conclusion:

Overall, we believe the societal impact of this work is positive. By promoting systematic, realistic evaluation of agent robustness, we contribute to building more reliable AI systems that better serve users and maintain safety as these technologies are deployed at scale.

References

Agarwal, S., Sahu, G., Puri, A., Laradji, I. H., Dvijotham, K. D., Stanley, J., Charlin, L., and Pal, C. Litllm: A toolkit for scientific literature review. *arXiv preprint arXiv:2402.01788*, 2024.

Anthropic. Introducing claude opus 4.5, 2025. URL https://www.anthropic.com/news/claude-opus-4-5.

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Chen, G., Zhang, Z., Cong, X., Guo, F., Wu, Y., Lin, Y., Feng, W., and Wang, Y. Learning evolving tools for large language models. *arXiv preprint arXiv:2410.06617*, 2024.

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Li, X. L., Kaiyom, F., Liu, E. Z., Mai, Y., Liang, P., and Hashimoto, T. Autobencher: Towards declarative benchmark construction. *arXiv preprint arXiv:2407.08351*, 2024.

Liu, A., Mei, A., Lin, B., Xue, B., Wang, B., Xu, B., Wu, B., Zhang, B., Lin, C., Dong, C., et al. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv: 2308.03688*, 2023.

Mialon, G., Fourrier, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., and Sun, M. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

Shi, D., Cao, J., Chen, Q., Sun, W., Li, W., Lu, H., Dong, F., Qin, T., Zhu, K., Liu, M., et al. Taskcraft: Automated generation of agentic tasks. *arXiv preprint arXiv:2506.10055*, 2025.

Shridhar, M., Thomason, J., Gordon, D., Bisk, Y., Han, W., Mottaghi, R., Zettlemoyer, L., and Fox, D. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. URL https://arxiv.org/abs/1912.01734.

Singh, A., Fry, A., Perelman, A., Tart, A., Ganesh, A., El-Kishky, A., McLaughlin, A., Low, A., Ostrow, A., Ananthram, A., et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Xie, J., Xu, D., Zhao, X., and Song, D. Agentsynth: Scalable task generation for generalist computer-use agents. *arXiv preprint arXiv:2506.14205*, 2025.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yang, J., Jimenez, C. E., Zhang, A. L., Lieret, K., Yang, J., Wu, X., Press, O., Muennighoff, N., Synnaeve, G., Narasimhan, K. R., et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

Yao, S., Chen, H., Yang, J., and Narasimhan, K. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Zhang, J., Peng, Y., Kong, F., Yang, C., Wu, Y., Yu, Z., Xiang, J., Ruan, J., Wang, J., Song, M., et al. Autoenv: Automated environments for measuring cross-environment agent learning. *arXiv preprint arXiv:2511.19304*, 2025.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A. Illustrative Example of Environment Evolution

For intuitiveness, we show proposed generation plans (*i.e.*, evolution contexts) as example evolution directions. To illustrate the general capability of our framework and the evolution coherence to real-world scenarios, we show examples in both the ecommerce domain where we experiment, and the airline domain using seed environment from τ^2-bench (Barres et al., 2025).

A.1. Completion strategy under ecommerce domain

V1 Evolution Context **Strategy:** `completion`

Task: Competitor Price Monitoring with Auto-Adjust Cart Alerts
User Story. *"As a shopper, I want to set target prices for products in my cart and across competitor websites, and be notified when the best price becomes available—either via a price drop on this website or a better deal at other Website."*

Why Not Supported.

1. **No `PriceHistory` entity:** only current pricing is stored (`Product.price`), with no historical tracking.
2. **No `PriceAlert` entity:** users cannot define target prices, alert conditions, or notification preferences.
3. **No `CompetitorPrice` entity:** competitor pricing is neither modeled nor comparable.
4. **No monitoring or scheduling tools:** existing tools are transactional, with no background tracking.
5. **No general notification mechanism:** notifications are task-specific (e.g., returns), not reusable.
6. **No user price preference storage:** price sensitivity and alert defaults are not persisted.
7. **No price analytics APIs:** no comparison against historical, competitor, or threshold-based signals.

Required Capabilities Missing.
Tools:

- `create_price_alert(user_id, product_id, target_price, alert_type, include_competitors?, competitor_list?)`
- `get_price_alerts(user_id, status?, product_id?)`
- `delete_price_alert(user_id, alert_id)`
- `get_price_history(product_id, days?, granularity?)`
- `get_competitor_prices(product_id)`
- `get_price_insights(product_id)`
- `trigger_price_notification(alert_id, trigger_type, current_price, trigger_source?)`

Entities:

- `PriceAlert`: alert_id, user_id, product_id, target_price, alert_type, status, include_competitors, competitor_list, created_at, triggered_at, notification_method
- `PriceHistory`: record_id, product_id, price, recorded_at, source, currency
- `CompetitorPrice`: competitor_price_id, product_id, competitor_name, competitor_url, competitor_price, last_updated, in_stock, shipping_cost
- `UserPricePreferences`: user_id, default_alert_threshold_percentage, preferred_notification_method, notification_frequency, tracked_competitors

A.2. Completion strategy under airline domain

V1 Evolution Context **Strategy:** `completion`

Task: Redeem Miles for Flight Upgrade
User Story. *"As a frequent flyer with accumulated miles, I want to redeem my miles to upgrade an existing economy reservation to business class, so I can enjoy a more comfortable flight without paying the full cash difference."*

Why Not Supported.

1. **No mileage balance tracking:** the `User` entity has no field for accumulated miles (e.g., `miles_balance`); `User.membership` only tracks tier.
2. **No mileage transaction history:** there is no entity recording how miles are earned, redeemed, transferred, or expired.
3. **No mileage earning rules:** the system lacks configuration for earning miles based on distance, cabin class, fare type, or tier multipliers.
4. **No redemption rate configuration:** there is no data defining miles required for cabin upgrades across routes or distances.
5. **No upgrade availability check with miles:** existing APIs (e.g., `update_reservation_flights`) only support cash-based changes.
6. **No miles redemption execution:** no tool exists to deduct miles and apply an upgrade to an existing reservation.
7. **No miles expiration tracking:** the system cannot track earning dates or expiration policies for loyalty miles.

Required Capabilities Missing.
Tools:

- `get_user_miles_balance(user_id)`
- `get_upgrade_miles_cost(reservation_id, target_cabin)`
- `check_upgrade_availability(reservation_id, target_cabin)`
- `redeem_miles_for_upgrade(user_id, reservation_id, target_cabin)`

Entities:

- `MilesAccount`: user_id, current_balance, lifetime_miles, pending_miles, miles_expiring_soon, expiration_date
- `MilesTransaction`: transaction_id, user_id, amount, transaction_type, source, reference_id, created_at, expiration_date
- `MilesRedemptionRate`: rate_id, redemption_type, origin_region, destination_region, from_cabin, to_cabin, miles_required, cash_copay

A.3. Saturation strategy under ecommerce domain

V2 Evolution Context **Strategy:** `saturation`

Saturation Evolution – New Shortcut Tools
Implement the following shortcut tools that provide direct access to data that was previously only accessible through multi-hop traversals.

New Tools to Implement

1. get_order_refund_summary
Type: `READ`
Description: Retrieves refund information for all returns and exchanges associated with a specific order, including request IDs and refund amounts.
Inputs (1 node(s)): `Order.order_id` (Order) [PK]
Outputs (2 node(s)): `ExchangeReturnRequest.request_id` (ExchangeReturnRequest) [PK]; `ExchangeReturnRequest.refund_amount` (ExchangeReturnRequest)
Relationship: references (one-to-many)
Discovery Path: `Order.exchange_return_request_ids -> ExchangeReturnRequest.request_id -> ExchangeReturnRequest.refund_amount`
Rationale: This is a highly valuable shortcut that eliminates the need to fetch exchange/return request IDs first and then individually query each request for refund details. It provides a complete financial summary of an order's returns/exchanges in a single call, which is essential for customer service and financial reconciliation.
Use Cases: 1. Customer service representative needs to quickly see all refunds associated with an order when handling a customer inquiry 2. Accounting department needs to reconcile order financials including all partial refunds from returns/exchanges 3. Customer viewing order history wants to see total refunded amount without navigating to individual return requests

2. get_subscription_order_swap_history
Type: `READ`
Description: Retrieves the complete product swap history for a subscription order, showing all product changes made to the subscription that generated this order.
Inputs (1 node(s)): `Order.parent_subscription_id` (Order) [FK]: Parent subscription ID if this is a subscription order
Outputs (2 node(s)): `SubscriptionSwapHistory.swap_id` (SubscriptionSwapHistory) [PK]: Unique swap event identifier; `SubscriptionSwapHistory.subscription_id` (SubscriptionSwapHistory) [FK]: Parent subscription reference
Relationship: references (one-to-many)
Discovery Path: `Order.parent_subscription_id -> Subscription.subscription_id -> SubscriptionSwapHistory.subscription_id -> SubscriptionSwapHistory.swap_id`
Rationale: This tool is valuable for understanding the context of subscription orders—showing what product swaps occurred before this order was generated. It's particularly useful for customer service scenarios where a customer receives an unexpected product and needs to understand the swap history. It eliminates multiple queries through subscription and swap history tables.
Use Cases: 1. Customer receives a subscription order with an unexpected product and customer service needs to trace back what swaps occurred 2. User reviews past subscription order and wants to understand why they received a different product than originally subscribed 3. Analytics team analyzing subscription swap patterns and their impact on order fulfillment

A.4. Saturation strategy under airline domain

V2 Evolution Context **Strategy:** `saturation`

Saturation Evolution – New Shortcut Tools
Implement the following shortcut tools that provide direct access to data that was previously only accessible through multi-hop traversals.

New Tools to Implement

1. get_reservation_flight_statuses
Type: `READ`
Description: Retrieves the current status of all flights in a reservation, allowing passengers to quickly check if any flights are delayed, cancelled, or on-time.
Inputs (1 node(s)): `Reservation.reservation_id` (Reservation) [PK]: Unique reservation identifier (e.g., 'ZFA04Y').
Outputs (2 node(s)): `ReservationFlight.flight_number` (ReservationFlight) [FK]: Flight number reference; `FlightInstance.status` (FlightInstance): Flight status (available, on time, flying, landed, cancelled, delayed).
Relationship: contains (one-to-many)
Discovery Path: `Reservation.flights -> ReservationFlight.flight_number -> Flight.flight_number -> FlightInstance.status`
Rationale: This is one of the most frequent customer queries—checking the status of their booked flights. Currently requires chaining through Reservation → ReservationFlight → Flight → FlightInstance. This shortcut eliminates 3 separate lookups and provides immediate value for customer service and self-service scenarios.
Use Cases: Customer checking if their upcoming flights are on time; Customer service agent quickly assessing disruption impact on a booking; Automated notification system checking for status changes on reservations; Mobile app displaying real-time flight status dashboard for a trip.

2. get_segment_flight_pricing
Type: `READ`
Description: Retrieves the price breakdown for all flights within a specific itinerary segment, essential for multi-city booking pricing and fare adjustments.
Inputs (1 node(s)): `ItinerarySegment.segment_id` (ItinerarySegment) [PK]: Unique identifier for the itinerary segment (e.g., 'SEG001').
Outputs (2 node(s)): `SegmentFlight.segment_flight_id` (SegmentFlight) [PK]: Unique identifier for the segment-flight association; `SegmentFlight.price` (SegmentFlight): Price for this flight in dollars.
Relationship: contains (one-to-many)
Discovery Path: `ItinerarySegment.segment_id -> SegmentFlight.segment_id -> SegmentFlight.segment_flight_id -> SegmentFlight.price`
Rationale: Multi-city reservations are complex and pricing queries per segment are frequent during booking modifications or fare recalculations. This shortcut from ItinerarySegment directly to SegmentFlight pricing eliminates intermediate lookups and is critical for the update_segment_cabin tool to calculate price differences.
Use Cases: Calculating upgrade costs for a specific leg of a multi-city trip; Displaying price breakdown by segment during booking review; Computing refund amounts when changing flights within a segment; Agent comparing alternative routing prices for a segment.

A.5. Deprecation strategy under ecommerce domain

V3 Evolution Context **Strategy:** `deprecation`

System Deprecation Notice
The following changes have been made to the system. Agents must adapt accordingly.

Deprecation Details
Type: `DATABASE`
Description: Service down: Cart (5 nodes, 7 edges)
Reason: Cart service undergoing scheduled maintenance for database migration to improve scalability and session handling. The cart system is being moved to a new distributed architecture to handle peak traffic better.
Impact: Users cannot view, modify, or create shopping carts. Cart-to-checkout flow is unavailable. Cart item counts and saved items cannot be retrieved. Affects 5 data fields and 5 cart-related tools.
Challenge Level: `MEDIUM`

Workaround. Agent can still help users browse products, check order history for past purchases, look up product details, and inform users when cart service will be restored. Users can note product IDs to add later, or agent could help with wishlists if available.

Removed Connections.
(1) `Product.asin -> Cart.items` — Tools affected: `add_to_cart(user_id: User.user_id, product_id: Product.asin, variant_id?, quantity?) -> Cart`;
(2) `Cart.cart_id -> Cart.user_id`;
(3) `Cart.cart_id -> Cart.items`;
(4) `Cart.user_id -> User.user_id` — Tools affected: `get_cart(user_id: User.user_id) -> Cart`, `add_to_cart(user_id: User.user_id, product_id: Product.asin, variant_id?, quantity?) -> Cart`, `clear_cart(user_id: User.user_id) -> Cart`;
(5) `Cart.items -> CartItem.cart_item_id` — Tools affected: `remove_from_cart(user_id: User.user_id, cart_item_id: CartItem.cart_item_id) -> Cart`, `update_cart_item(user_id: User.user_id, cart_item_id: CartItem.cart_item_id, quantity: int) -> Cart`;
(6) `User.user_id -> Cart.items` — Tools affected: `get_cart(user_id: User.user_id) -> Cart`, `add_to_cart(user_id: User.user_id, product_id: Product.asin, variant_id?, quantity?) -> Cart`, `clear_cart(user_id: User.user_id) -> Cart`;
(7) `User.cart_id -> Cart.cart_id`.

Removed Data Points.
`Cart.cart_id` (Cart); `Cart.user_id` (Cart); `Cart.items` (Cart); `Cart.created_at` (Cart); `Cart.updated_at` (Cart).

Deprecated Tools.
`add_to_cart(user_id: User.user_id, product_id: Product.asin, variant_id?, quantity?) -> Cart`; `update_cart_item(user_id: User.user_id, cart_item_id: CartItem.cart_item_id, quantity: int) -> Cart`; `remove_from_cart(user_id: User.user_id, cart_item_id: CartItem.cart_item_id) -> Cart`; `get_cart(user_id: User.user_id) -> Cart`; `clear_cart(user_id: User.user_id) -> Cart`.

Agent Requirements.
(1) Detect when deprecated services/tools are called; (2) Find alternative paths when available; (3) Gracefully handle cases with no alternatives; (4) Provide meaningful error messages.

A.6. Deprecation strategy under airline domain

V3 Evolution Context **Strategy:** `deprecation`

System Deprecation Notice
The following changes have been made to the system. Agents must adapt accordingly.

Deprecation Details
Type: `TOOL`
Description: Tool deprecation: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]` (8 edges affected)
Reason: The direct flight search API is being consolidated into the unified flight search website. As part of our Q4 API modernization initiative, specialized search endpoints are being retired in favor of the comprehensive `search_oneway_flight` and `search_roundtrip_flight` APIs which support filtering by number of stops. This reduces API surface area and maintenance overhead.
Impact: Agents can no longer directly search for non-stop flights between two cities. Flight discovery for direct routes requires using general search APIs and filtering results, or searching specific flight numbers if known.
Challenge Level: `MEDIUM`

Workaround. Use `search_oneway_flight` to find flights between origin and destination, then filter results to identify direct (non-stop) flights by examining the flight data structure for connection indicators or flight duration.

Removed Connections.
(1) `Input.user_input -> Flight.flight_number` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`, `search_onestop_flight(origin: str, destination: str, date: str) -> List[Tuple[DirectFlight, DirectFlight]]`; (2) `Flight.flight_number -> Flight.origin` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`; (3) `Flight.flight_number -> Flight.destination` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`; (4) `Flight.flight_number -> Flight.scheduled_departure_time_est` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`; (5) `Flight.flight_number -> Flight.scheduled_arrival_time_est` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`; (6) `Flight.flight_number -> Flight.dates` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`, `get_flight_status(flight_number: str, date: str) -> str`; (7) `FlightInstance.flight_number -> FlightInstance.available_seats` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`, `get_flight_availability_details`; (8) `FlightInstance.flight_number -> FlightInstance.prices` — Tools affected: `search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`.

Deprecated Tools.
`search_direct_flight(origin: str, destination: str, date: str) -> List[DirectFlight]`.

Agent Requirements. (1) Detect when deprecated services/tools are called; (2) Find alternative paths when available; (3) Gracefully handle cases with no alternatives; (4) Provide meaningful error messages.

B. Illustrative Example of Graph-Grounded Task Generation

To explicitly illustrate the task generation pipeline described in Section 4.3, we present a simplified, representative walkthrough of generating a 'Product Exchange' task within the e-commerce environment.

Step 1: Subgraph Sampling The framework first samples a connected subgraph $\mathcal{G}_{\mathcal{T}} \subseteq \mathcal{G}$ that defines the structural scope of the task. This subgraph ensures that a valid path exists between the user's starting state (Authentication) and the desired outcome (Exchange).

- Sampled Nodes ($\mathcal{V}_\tau$): `User` $\rightarrow$ `Order` $\rightarrow$ `Product` $\rightarrow$ `ExchangeRequest`.
- Sampled Edges/Tools ($\mathcal{E}_\tau$):
  - `authenticate_user()` (connects Session to User)
  - `get_order_by_id()` (connects User to Order History)
  - `initiate_exchange()` (connects Order to ExchangeRequest)
- Goal Synthesis (g_τ): The LLM formulates a goal realizable within $\mathcal{G}_\tau$: *"Exchange the delivered grey Burkini for the correct purple variant."*
- Scenario Description (s_τ): *You are Susan Morales. You ordered a purple Meijunter Burkini (size M) but received grey. You need to verify the order details and initiate an exchange."*

Step 2: Sandbox Materialization The framework instantiates the specific entities required by $\mathcal{G}_\tau$ to ensure the task is executable:

- A `User` entity is created: `Susan Morales` (ID: `user_089`).
- An `Order` entity is injected (`ORD-2024-089-001`) with a specific anomaly: the `items` list contains the `grey` variant, while the user's intent is `purple`.
- The `Product` database is populated with `B0861GWJV2` (Meijunter Burkini) to ensure variant data exists.

Step 3: Agentic Walk Execution To generate the multi-turn dialogue, a simulator executes a reference trajectory by expanding along $\mathcal{G}_\tau$. At each turn t, the simulator identifies the *frontier set* (newly reachable nodes) and generates a state-wise customer instruction u_t^*.

- Turn 1 (Authentication):
  - Frontier: The simulator identifies the `User` node is reachable via authentication.
  - Generated Utterance (u_1^*): *"Hi, I need to check on an order. My email is susan.morales.user_089@example.com."*
  - Action (a_1): Agent calls `authenticate_user(email="...")`.
- Turn 2 (Order Verification):
  - Frontier: With the user authenticated, specific `Order` nodes become reachable.
  - Generated Utterance (u_2^*): *"I ordered a purple Burkini but received grey. The order number is ORD-2024-089-001. Can you check the details?"*
  - Action (a_2) Agent calls `get_order_by_id(order_id="ORD-2024-089-001")`.
- Turn 3 (Exchange Initiation):
  - Frontier: With the order details exposed (confirming the grey variant was shipped), the `ExchangeRequest` node becomes reachable via `initiate_exchange()`.
  - Generated Utterance (u_3^*): *"Yes, that confirms it was the wrong color. Please start an exchange for the purple size M."*
  - Action (a_3): Agent calls `initiate_exchange(order_id=..., new_variant="purple_tag m")`.

C. Prompts used in environment evolution workflow

C.1. Task Proposer in Completion Strategy

System

You are a Product Manager for a `{domain.name}` website. Your job is to identify new features and capabilities that would enhance the website but are NOT currently supported.

## Domain Context: `{domain.description}`

## You Have Deep Knowledge Of:
- `{domain.name.title()}` best practices
- Customer needs and pain points in domain.name
- Modern `{domain.name}` features (`{common_features}`)
- Backend data modeling and API design
- Industry-specific regulations and requirements

When proposing tasks, be specific and realistic. Consider the user journey and what data and tools would be needed to support the feature in a `{domain.name}` context.

Prompt

## Task Proposal Request

Analyze the current domain.name system and propose a NEW feature/task that is NOT currently supported. The new feature should extend the system without significantly changing existing functionality.

### Domain Context **{domain.name.title()}**:
{domain.description}

### Current System State

#### Databases (Data Entities):
{', '.join(databases)}

#### Available Tools (APIs):
{', '.join(tools)}

#### Current Graph Structure:
"' json
{graph_json}

### Your Task Propose a feature in the domain of "{selected_domain}" that the current system CANNOT support.

Use seed value {seed} to ensure your proposal is unique and creative. {exclusion_text}

#### Output Format Respond with the task in EXACTLY this format (wrapped in <task_proposal> tags):

<task_proposal>
### Task: [Concise Task Name] User Story: "[First-person user story describing the need]"

Why Not Supported:

[Specific reason 1 - what data/entity is missing] [Specific reason 2 - what tool/API is missing] [Continue with specific gaps...]

Required Capabilities Missing:

[tool_name_1()] tool - [what it does] [tool_name_2()] tool - [what it does] [EntityName] entity with fields: [field1, field2, ...] [Continue with specific requirements...] <task_proposal>

#### Guidelines - Be specific about what's missing - Reference actual databases and tools from the current system when explaining gaps - Propose realistic domain.name features that would add business value - Ensure the task is achievable with reasonable additions to the system. - Consider domain.name-specific requirements and best practices

C.2. Tool Designer in Saturation Strategy

System

You are an expert API designer for {domain.name} systems. Your task is to design meaningful, well-named tools (API endpoints) that provide shortcuts for accessing data across entity relationships.

## Domain Context: {domain.description}

## You Understand:
- {domain.name.title()} domain concepts ({entities_formatted.lower()}, etc.)
- RESTful API naming conventions
- The value of convenience methods that combine multiple operations
- Clear, descriptive naming that indicates what the tool does
- {domain.name.title()}-specific terminology and conventions

## Your Tool Names Should Be:
- Snake_case format
- Action-oriented (get_, list_, find_, search_, add_, update_, etc.)
- Clear about what data is being accessed or modified
- Concise but descriptive
- Using {domain.name}-appropriate terminology

Prompt

## Tool Design Task

### Domain Context {domain.name.title()}: {domain.description}

### System Context

#### Databases/Entities: {', '.join(databases)}

#### Existing Tools (for context, you may create similar tools with different functionality): {existing_tools_display}

### Task

You are given {len(path_descriptions)} candidate paths discovered through graph traversal. Your task is to:

1. **SELECT** the {num_tools} MOST VALUABLE paths that would benefit from having shortcut tools
2. **DESIGN** a tool for each selected path, choosing which nodes serve as INPUTS and which as OUTPUTS

### Multi-Input/Multi-Output Tools

You can design tools with flexible input/output mappings:
- **Single Input → Single Output**: Simple lookup (e.g., get product by user_id)
- **Multiple Inputs → Single Output**: Filtered lookup (e.g., get order by user_id AND date)
- **Single Input → Multiple Outputs**: Fetch related data (e.g., get user's orders AND cart items)
- **Multiple Inputs → Multiple Outputs**: Complex queries (e.g., get products and reviews by user AND category)

For each path, you can select ANY nodes from the path as inputs or outputs - they don't have to be just the endpoints!

### Selection Criteria (use your judgment):
- **Business Value**: How useful would this shortcut be in real {domain.name} scenarios?
- **Time Savings**: How much complexity does the shortcut eliminate?
- **Natural Fit**: Does this relationship make intuitive sense to users/developers?
- **Diversity**: Select paths that provide different types of functionality
- **Practicality**: Would developers actually use this tool frequently?

### All Candidate Paths

"'json {json.dumps(path_descriptions, indent=2)} "'

#### Output Format Select exactly {num_tools} paths and provide tool proposals in this exact JSON format (wrapped in <tool_proposals> tags):

<tool_proposals> [{{ "path_id": 0, "tool_name": "snake_case_tool_name", "tool_type": "READ or WRITE", "description": "Clear description of what the tool does", "input_node_ids": [1, 5], "output_node_ids": [12, 15, 18], "relationship_type": "references|belongs_to|contains|aggregates|links_to", "cardinality": "one-to-one|one-to-many|many-to-one|many-to-many", "rationale": "Why you selected this path and why this shortcut is valuable", "use_cases": ["Use case 1", "Use case 2"] }}] </tool_proposals>

#### Guidelines - Input/Output Selection:
- 'input_node_ids': List of node_ids from the path that serve as inputs (parameters to the tool)
- 'output_node_ids': List of node_ids from the path that serve as outputs (returned data)
- You can select ANY nodes from the path - not just start/end
- At least one input and one output are required
- Tool Names: Use snake_case, be descriptive but concise
- READ tools: get_, list_, find_, search_, lookup_
- WRITE tools: add_, update_, link_, associate_, set_

Be Specific. The tool should clearly indicate:
- What entity/data is being accessed
- The relationship between inputs and outputs
- Whether it's a read or write operation
- Use Cases: Provide realistic {domain.name} scenarios

#### Relationship Types:
'references': Direct FK relationship
'belongs_to': Child-to-parent relationship
'contains': Parent-to-children relationship
'aggregates': Computed/derived relationship
'links_to': Cross-entity shortcut

Important: You MUST select exactly {num_tools} paths. Choose wisely based on value, not just arbitrarily.

C.3. Deprecation Strategy

System

You are an expert system architect responsible for managing API deprecations and service maintenance in a {domain.name} website.

## Domain Context: {domain.description} ### Core Functional Areas: {core_areas_formatted}

Your responsibilities: 1. Evaluate proposed deprecations for realism and business sense 2. Assess the impact on downstream systems and users 3. Determine appropriate challenge levels for system resilience testing 4. Suggest workarounds when available

You understand: - Real-world reasons for API deprecation (security, performance, consolidation, sunset) - Service outage scenarios (maintenance, failures, migrations) - How agents/systems should gracefully handle unavailable services - The importance of testing system robustness - {domain.name.title()}-specific operational concerns and compliance requirements

Your deprecation decisions should be: - Realistic (something that would actually happen in a {domain.name} production system) - Challenging but fair (agents should be able to adapt) - Well-documented (clear reasons and workarounds) - Domain-appropriate (consider {domain.name}-specific regulations and practices)

Prompt

## Deprecation Selection Task

### Context You are reviewing deprecation candidates for a {domain.name} system. Sampling mode used: **{mode}**

### Graph Overview - Databases: {', '.join(graph_databases)} - Total nodes: {graph_num_nodes} - Total edges: {graph_num_edges}

### Candidates

“‘json {json.dumps(candidates, indent=2)} “‘

### Your Task Select ONE candidate that best satisfies: - Realism: The deprecation should be something that could actually happen in production - Challenge: It should create a meaningful challenge for an agent to handle (prefer medium to hard difficulty) - Interestingness: The deprecation should test the agent's ability to adapt

### Output Format Provide your selection in this JSON format wrapped in <deprecation_decision> tags:

<deprecation_decision> {{ "candidate_id": 0, "deprecation_reason": "Realistic reason why this would be deprecated", "impact_summary": "Brief description of what functionality is affected", "challenge_level": "easy|medium|hard|extreme", "workaround_hint": "Brief hint about how an agent might work around this" }} </deprecation_decision>

#### Challenge Level Guidelines - easy: Clear alternatives exist, minimal adaptation needed - medium: Alternatives exist but require finding new paths (PREFERRED) - hard: Limited alternatives, significant changes needed - extreme: Critical functionality lost, graceful failure required

Choose wisely - select the candidate that creates the most interesting and realistic challenge.

C.4. Graph Evolver for Attribute Graph Update

System

You are a Software Architect designing data models and APIs for a {domain.name} website.

## Domain Context: {domain.description}

Given a task requirement and the current system state (represented as an attribute graph), your job is to: 1. Design new data entities (databases) with their attributes 2. Design new tools (APIs) that operate on these entities 3. Define relationships between new and existing entities

## Follow These Conventions: - Database names are PascalCase (e.g., {', '.join(list(domain.example_entities.keys())[:3])}) - Attribute names are snake_case (e.g., {list(domain.example_entities.keys())[0].lower()}_id, created_at) - Tool names are snake_case verbs (e.g., create_{list(domain.example_entities.keys())[0].lower()}, get_{list(domain.example_entities.keys())[1].lower()}_status) - Primary keys typically end with _id - Foreign keys reference other entities' primary keys - Include created_at/updated_at timestamps for mutable entities

## Relationship Types: references, belongs_to, contains, has_attribute, identifies, used_for, updates, explains

## Cardinalities: one-to-one, one-to-many, many-to-one, many-to-many

Prompt

## Graph Evolution Design Task

### Domain Context {domain.name.title()}: {domain.description}

Given the following task requirement and current system state, design the data model and API changes needed.

### Task Requirement

{task_full_text}

### Current System State

“‘json {graph_json} “‘

### Your Task Design the additions needed to support this task: - New Databases (Entities): What new data entities are needed? - New Nodes (Attributes): What attributes do these entities need? What new attributes for existing entities? - New Edges (Relationships): How do new entities connect to existing ones? - New Tools: What new APIs are needed?

#### Output Format Respond with your design in EXACTLY this JSON format (wrapped in <graph_evolve_design> tags). Include fields like ‘allowed_values‘ that are presented in the original json when applicable:

<graph_evolve_design> {{ "rationale": "Brief explanation of the design decisions", "new_databases": ["DatabaseName1", "DatabaseName2"], "new_nodes": [{{ "database": "DatabaseName", "attribute": "attribute_name", "type": "str|int|float|bool|List[str]|Dict[str, Any]|Optional[str]|etc", "description": "What this attribute represents", "is_primary_key": true|false, "is_foreign_key": true|false, "modifiable": true|false, }}], "new_edges": [{{ "source_database": "SourceDB", "source_attribute": "source_attr", "target_database": "TargetDB", "target_attribute": "target_attr", "relationship_type": "references|belongs_to|contains|has_attribute", "cardinality": "one-to-one|one-to-many|many-to-one|many-to-many", "description": "What this relationship represents", "tools": ["tool_name_1(argument_1: node_name_1, argument_2: node_name_2) -> return_data_class", ...] }}], "new_tools": ["tool_name_1(argument_1: node_name_1, argument_2: node_name_2) -> return_data_class", ...] }} </graph_evolve_design>

### Design Guidelines - Every new database MUST have a primary key (usually ‘[entity]_id‘) - Foreign keys should reference existing primary keys - Tools should be listed on the edges they operate on - Include both READ and WRITE tools as appropriate - Connect new entities to existing entities where logical - Include standard fields like ‘created_at‘, ‘updated_at‘, ‘status‘ where appropriate - Use consistent naming conventions: - Database names: PascalCase - Attribute names: snake_case - Tool names: snake_case verbs (e.g., create_order, get_order_status)

C.5. Coding Agent in Environment Implementation

System

You are the Coding Agent in a {domain.name} environment evolution pipeline. Your role is to generate updated Python code (data models and tool implementations) based on evolved graph representations.

## Pipeline Context

The environment evolution pipeline works as follows: 1. A Base Environment is represented as an attributed graph (nodes = data entities, edges = relationships + tools) 2. Graph Evolution Strategies transform the base graph into a new graph representation: - **Completion**: Adds nodes and edges based on task requirements from a Task Proposal Agent - **Saturation**: Adds edges via random walk sampling to discover new tool capabilities - **Deprecation**: Removes edges and nodes via graph pruning to deprecate features 3. You (the Coding Agent) receive the new graph representation and generate: - ‘data_model.py‘: Updated Pydantic models reflecting schema changes - ‘tools_implementation.py‘: Updated tool implementations reflecting edge changes

## Your Responsibilities

1. **Analyze Graph Differences**: Compare the base graph with the new graph to identify: - Added nodes → Create new Pydantic models or add fields to existing models - Removed nodes → Remove or deprecate corresponding models/fields - Added edges → Implement new tools with @is_tool decorators - Modified edges → Update tool signatures, parameters, or behavior - Removed edges → Remove or deprecate corresponding tools

2. **Maintain Code Consistency**: Use the base_data.py and base_tool.py as templates to ensure: - Consistent coding patterns and style - Proper Pydantic model inheritance and validation - Correct @is_tool(ToolType.READ/WRITE) decorators - Comprehensive docstrings with Args, Returns, Raises sections - Appropriate error handling with ValueError for invalid inputs

3. **Handle Evolution Strategies Appropriately**: - Completion: Focus on adding new functionality while preserving existing code - Modification: Update existing tools/models without breaking backward compatibility - Saturation: Add tools that leverage existing data relationships - Deprecation: Safely remove code while maintaining system integrity

## {domain.name.title()} Domain Scope

The environment covers these core functional areas, and will progress to support additional features: {core_areas_formatted}

### Key Entities and ID Patterns {entities_formatted}

### Database and Toolkit Classes - Database class: '{domain.database_class}' - Toolkit class: '{domain.toolkit_class}'

Always generate complete, syntactically correct Python code that can be directly saved to files.

Prompt

### Evolution Context

The following context describes why this evolution is happening and should guide your implementation:

{evolution_context}

"""

prompt += f""" ### Environment files

<attribute_graph> {graph_rep} </attribute_graph>

<data_model> {data_model} </data_model>

<tools_implementation> {tools_implementation} </tools_implementation>

<updated_attribute_graph> {updated_graph_rep} </updated_attribute_graph>

<updated_data_model> {updated_data_model} </updated_data_model>

### Instructions

1. Analyze the differences between the original 'attribute_graph' and 'updated_attribute_graph' 2. Identify new edges that represent new tools to be implemented 3. For each new tool mentioned in the edges' "tools" field: - Create a new method with the @is_tool decorator - Determine if it's a READ or WRITE operation based on the relationship type - Implement the tool logic following existing patterns in the codebase - Add proper docstrings with Args, Returns, and Raises sections - Handle error cases appropriately 4. Update imports to include any new models from the updated data_model 5. Update the '{domain.database_class}' or relevant database class if new collections are needed 6. Ensure the tools properly interact with the new data structures 7. Follow the existing code style and patterns 8. Add any necessary helper methods

### Key Considerations for {domain.name.title()} Domain

- READ tools should use @is_tool(ToolType.READ) - WRITE tools should use @is_tool(ToolType.WRITE) - Tools should validate inputs and raise ValueError for invalid data - Tools should properly update related entities (e.g., updating user references when creating new entities) - Use the ‘{domain.toolkit_class}‘ class for all tool implementations

Please update the ‘tools_implementation.py‘ file, wrapping the complete updated code within <updated_tools_implementation> and </updated_tools_implementation> tags. Output the complete file, not just the changes. """

C.6. Test Generation in Environment Implementation

System

You are the Test Generation Agent in a {domain.name} environment evolution pipeline. Your role is to generate comprehensive pytest unit tests for newly created or modified data models and tools.

## Pipeline Context

After the Coding Agent generates updated data models and tool implementations based on evolved graph representations, you generate tests to verify the correctness of: 1. New Pydantic data models (field validation, serialization, edge cases) 2. New tool implementations (happy paths, error handling, edge cases) 3. Integration between models and tools

## Your Responsibilities

1. **Generate Test Specifications**: Analyze the graph changes to determine what needs testing: - New data models $\rightarrow$ Test instantiation, field validation, serialization - New tools $\rightarrow$ Test input/output contracts, error handling, edge cases - New relationships $\rightarrow$ Test data integrity across entities

2. **Write Comprehensive Tests**: For each component, generate tests covering: - **Happy Path**: Valid inputs produce expected outputs - **Edge Cases**: Boundary values, empty inputs, maximum lengths - **Error Handling**: Invalid inputs, missing data, not found scenarios - **Type Validation**: Correct types are enforced

3. **Follow Testing Best Practices**: - Use pytest framework with clear test function names (test_<span><span style="color: rgb(150, 34, 73); font-weight: bold;"><function></span><span style="color: black; font-weight: normal;">_) - Use unittest.mock to mock database operations and external dependencies - Create realistic test fixtures with valid {domain.name} data - Each test should be independent and not rely on other tests - Use descriptive assertion messages

4. **Mock Database Operations**: - Mock the database class (‘{domain.database_class}‘) to avoid actual database calls - Set up mock return values that match expected data structures - Verify that database methods are called with correct arguments

## {domain.name.title()} Domain Context

### Key Entities and ID Patterns {entities_formatted}

### Example Test Data Generate test data that reflects realistic {domain.name} scenarios: {test_data_formatted}

### Database and Toolkit Classes - Database class: ‘{domain.database_class}‘ - Toolkit class: ‘{domain.toolkit_class}‘

Always generate complete, syntactically correct pytest code that can be directly executed.

Prompt

### Evolution Context

The following context describes why this evolution is happening and should guide your implementation:

{evolution_context}

"""

prompt += f""" ### Updated Graph Representation

{updated_graph_rep}

### Graph Changes Summary

New Databases/Entities: {graph_diff.get('new_databases', [])}

Added Nodes (new data fields): {_format_nodes_for_prompt(graph_diff.get('added_nodes', []))}

Removed Nodes: {_format_nodes_for_prompt(graph_diff.get('removed_nodes', []))}

Added Edges (new relationships/tools): {_format_edges_for_prompt(graph_diff.get('added_edges', []))}

Removed Edges: {_format_edges_for_prompt(graph_diff.get('removed_edges', []))}

New Tools to Implement: {graph_diff.get('new_tools', [])}

### Instructions

Based on the graph changes above, generate a structured test specification that defines WHAT should be tested. Do NOT write actual test code yet - just specify the test requirements.

For each component, provide:

#### 1. Data Model Test Specifications For each new or modified data model: - Model name and purpose - Required fields and their expected types - Field validation rules (min/max, patterns, allowed values) - Optional vs required fields - Edge cases to test (empty values, null, boundary values, invalid types) - Serialization/deserialization requirements

#### 2. Tool Test Specifications For each new or modified tool: - Tool name and purpose - Input parameters with types and constraints - Expected output structure for success cases - Error conditions and expected exceptions - Edge cases to test: - Empty/null inputs - Non-existent references (e.g., entity not found) - Invalid parameter types - Boundary values - Side effects to verify (e.g., database updates, related entity changes)

#### 3. Integration Test Specifications For relationships between entities: - Data integrity constraints to verify - Cross-entity operations to test - Cascading effects to validate

### {domain.name.title()} Domain Test Data Examples

Use these patterns for realistic test data: {chr(10).join(f"- {k}: {repr(v)}" for k, v in domain.example_test_data.items())}

Output the test specifications in a structured format wrapped in <test_specifications> and </test_specifications> tags. Use clear sections and bullet points for each test requirement.

D. Example of Reflection Replay Strategy.

```
# Previous Task Reflections

You have completed similar tasks before. Here are reflections from those experiences:

   **What went well:**
   The assistant successfully navigated the core e-commerce workflow, from product search
   to checkout. Key strengths included systematically using search tools to find relevant
   products, clearly presenting product details (price, availability, ratings), and
   efficiently guiding the user through adding items to the cart. The assistant correctly
   applied the user's saved shipping address and payment method at checkout, confirmed the
   order summary, and provided a clear final confirmation with the order ID and
   total\u2014all of which directly supported the user's stated goal.

   **What could be improved:**
```

```
The task required 56 steps over nearly 5 minutes, indicating significant inefficiency.
Much of the time was spent on overly granular product searches and repetitive
verification steps. For instance, after the user expressed interest in a specific
4-shelf organizer, the assistant conducted multiple broad searches instead of
immediately focusing on that variant. Furthermore, the assistant occasionally asked for
redundant confirmations (e.g., re-verifying the shipping address already provided in
the tool output) and included unnecessary detail in responses, which extended the
conversation without adding value.

**Key learnings:** The primary insight is that strict, linear step-by-step execution
can create friction in a fluid conversational task. While the assistant completed the
required actions, it did not optimize for speed or conversational flow. Future
performance can be enhanced by interpreting user intent more holistically\u2014for
example, when a user asks to \"review the cart,\" the assistant can immediately proceed
to checkout if the cart is already populated, rather than waiting for explicit
confirmation at each micro-step. Additionally, tool calls should be more targeted; once
a product is identified, subsequent calls should focus on that specific item's
availability and variants.

**Recommendations:** For similar tasks, adopt a more streamlined approach: (1) After
the initial search, quickly narrow options based on user cues (e.g., \"4-shelf
version\") and check stock for that specific item. (2) Consolidate
confirmations\u2014assume the user wants to proceed if they add an item to the cart and
then ask to review it. (3) Use tool outputs more efficiently; if a tool returns the
cart total and items, immediately present the summary and move to checkout without
intermediate verification steps. (4) Aim to complete the core workflow in fewer, more
decisive steps, reducing the step count by at least 30-40% while maintaining clarity.
```