THE WORLD WON'T STAY STILL: PROGRAMMABLE EVOLUTION FOR AGENT BENCHMARKS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-powered tool-calling agents fulfill user requests by interacting with environments and invoking tools over multi-turn processes. Yet most existing benchmarks evaluate these systems under static environment interfaces with fixed schemas and toolsets, making it difficult to assess how agents behave as environments evolve — when capabilities are added, reorganized, or deprecated across successive environment versions. In this paper, we study *structured environment evolution* as a benchmark-construction problem for tool-calling agents. 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 schemas — and their dependencies. Under this formalism, adding, removing, or modifying capabilities is expressed as graph transformations that coherently propagate updates across tools, schemas, and data access. Building on this, PROEVOLVE supports (1) automatic generation of evolved executable environments through explicit graph transformations, and (2) graph-grounded construction of task sandboxes via subgraph sampling and instantiation. We validate PROEVOLVE in two tool-calling domains, e-commerce and airline booking, in terms of quality, implementation validity, and failure modes. Finally, we use the generated benchmark as a downstream diagnostic to study how representative agents behave under structured environment evolution.

1 INTRODUCTION

Tool-calling agents operate over structured environments in which schemas, data entities, and tool interfaces jointly determine what information can be accessed and what actions can be executed (Yao et al., 2023; Barres et al., 2025; Shridhar et al., 2020; Yang et al., 2018). Despite recent progress, existing approaches typically evaluate agents under *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 simplifies benchmark design, but it limits our ability to assess how agents behave when capabilities are added, reorganized, or deprecated across environment versions.

Recent efforts have expanded the scale and diversity of agent benchmarks by increasing the number of tools or tasks (Li et al., 2024; Shi et al., 2025; Yao et al., 2024; Chen et al., 2024), or by introducing curated sets of distinct environments (Zhang et al., 2025). These approaches improve coverage, but they typically treat environments as fixed snapshots or independently generated variants, rather than as successive versions linked by structured transitions. As a result, they capture environment variation, but not structured environment evolution. In tool-calling environments, however, tools, data entities, and schemas are tightly coupled: adding, reorganizing, or deprecating a capability often requires coordinated updates across multiple components. For example, adding an order-cancellation capability to an order-placement system may reuse and extend existing order schemas, data entities, and tool interfaces, rather than introducing an isolated subsystem. Thus, what distinguishes environment *evolution* from environment *variation* is the version linkage: each version is derived from its predecessor, unmodified structure is preserved, and intended changes are propagated coherently across coupled components.

In this paper, we study structured environment evolution as a benchmark-construction problem for tool-calling agents. We focus on environments where schemas, data entities, and tools are tightly

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

coupled, such that modifying one component often requires coordinated updates to others. We argue that environment evolution is distinct from task scaling, independent environment variation, and surface-level tool perturbation: an evolved environment should be derived from a previous version, preserve unmodified structure, and propagate intended changes coherently across schemas, tools, data access, executable behavior, and generated tasks. This leads to two desiderata: (1) scalable coherence — generating diverse environment versions without breaking dependencies, and (2) controlled versioned dynamics — making each transition explicit, reproducible, and evaluable.

To address this problem, we propose PROEVOLVE, a graph-based framework for programming structured evolution in tool-calling environments. PROEVOLVE represents each environment as a typed relational graph over schemas, data entities, and tool-enabled transitions. Under this formalism, semantic evolution intents such as capability addition, access-path reorganization, and deprecation are implemented as explicit graph transformations. The transformed graph is then instantiated into executable environments and graph-grounded task sandboxes, with validation of implementation correctness, semantic intent alignment, diversity, and generation failure modes. We evaluate PROEVOLVE in two structured tool-calling domains, e-commerce and airline booking, and use downstream agent results as diagnostic evidence that the generated environment changes are meaningful.

Concretely, PROEVOLVE generates 150 evolved-version candidates in e-commerce and 15 in airline booking, with 3,000 and 300 task sandboxes across their respective version-linked trajectories. Our validation pipeline combines graph-validity checks, compilation, graph-grounded unit tests, severity-based defect filtering, and oracle replay, complemented by assessments of semantic fidelity, task quality, and diversity. Version linkage enables three complementary analyses that are difficult to conduct with static or independently generated benchmarks: applying individual primitives to a shared seed while holding tasks fixed, tracking agent behavior along cumulative lineages, and varying the prior-version context supplied to agents. Our single-edit comparisons show that deprecation reduces success across all models in the matched panel despite verified alternative paths, with interaction patterns consistent with repeated recovery attempts in some models and early termination in others. Context-prior analyses further show that distilled experience does not uniformly improve task completion and can be associated with failures to recognize available tools. Together, these findings demonstrate the diagnostic value of programmable, version-linked environments for studying agent behavior under structured environment change.

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 large 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 whether 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 to adapt 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 interfaces, including names, parameters, and response formats, to evaluate robustness.

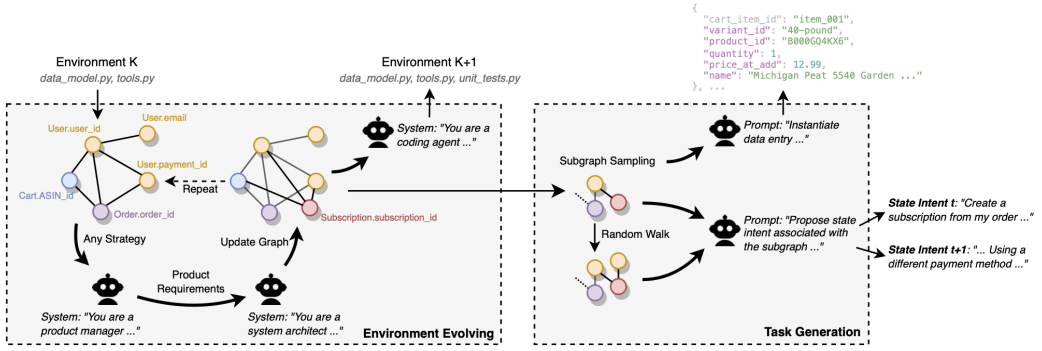

Figure 1: **End-to-end workflow of PROEVOLVE.** Starting from a seed tool-calling environment, PROEVOLVE represents schemas, data entities, and tools as an environment graph, applies semantic evolution intents as graph transformations, and instantiates the transformed graph into executable environment versions. For each version, task sandboxes are generated by sampling task subgraphs, materializing version-specific prerequisites, and producing state-wise user instructions and success criteria, enabling controlled evaluation under structured environment evolution.

3 FROM STATIC TO EVOLVING: PROBLEM SETUP AND CHALLENGES

Existing benchmarks typically evaluate tool-calling agents in static environments, where tools, schemas, and reachable relations remain fixed throughout evaluation. Yet real-world systems are rarely static: capabilities are introduced, interfaces are updated, and outdated functionality may be deprecated over time. Motivated by this gap, we study evolving environments, where these interfaces change across versions through explicit, dependency-aware updates. This shift from fixed snapshots to structured versioned change is the setting addressed in this section.

3.1 DEFINITION: STRUCTURED ENVIRONMENT EVOLUTION

We now formalize the notion of *structured environment evolution* that this paper studies, distinguishing it from weaker forms of environment modification prevalent in existing benchmarks.

Definition 3.1 (Structured Environment Evolution)**.** *Let $\mathcal{G}^{(0)}$ be a seed environment. A* structured environment evolution *is a sequence $\mathcal{G}^{(0)} \xrightarrow{\Delta^{(1)}} \mathcal{G}^{(1)} \xrightarrow{\Delta^{(2)}} \cdots \xrightarrow{\Delta^{(K)}} \mathcal{G}^{(K)}$ satisfying:*

1. ***Sequential Dependence.*** *Each $\Delta^{(k+1)}$ is conditioned on current $\mathcal{G}^{(k)}$—not independently sampled.*
2. ***Structural Memory.*** *$\mathcal{G}^{(k+1)}$ preserves all components of $\mathcal{G}^{(k)}$ not explicitly removed by $\Delta^{(k+1)}$, maintaining continuity across versions.*
3. ***Coherent Propagation.*** *$\Delta^{(k+1)}$ simultaneously updates all coupled components (schemas, tools, data access), maintaining functional consistency.*

Unlike task scaling, independent environment generation, or surface-level interface perturbation, structured evolution is version-linked: each transition preserves unmodified structure and coherently propagates intended changes across the environment and its task sandboxes.

This leads to the central question of the paper:

> *How to evolve environments in a controllable manner for agent benchmarking?*

Answering this question requires two desiderata:

Desideratum I: Scalable Coherence. Environment evolution should scale beyond manually curated updates while preserving coherent dependencies among schemas, data entities, tools, executable behavior, and task sandboxes. Naively adding tools or entities may increase benchmark size, but can break reachability, create invalid data dependencies, or produce tasks that are no longer grounded in the corresponding environment version. Thus, scalable evolution requires generating diverse environment versions without sacrificing structural and functional consistency.

Desideratum II: Controlled Versioned Dynamics. Environment changes should be explicit, reproducible, and evaluable across versions. Each transition should specify what is added, reorganized, or deprecated; how the change propagates through coupled components; and how tasks remain valid under the new version. Without such control, benchmark construction reduces to unconstrained environment sampling rather than structured evolution.

These challenges motivate us to propose a programmable framework for evolving environments in a controllable and scalable manner, as described in the following sections.

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) designing evaluation along evolution trajectories in Sec. 4.4. We present an overview of the framework in Figure 1.

4.1 GRAPH FORMALISM FOR ENVIRONMENT MODELING

To operationalize structured environment evolution, we need an explicit representation of the components that co-evolve in a tool-calling environment. We therefore model schemas, data entities, tools, and their dependency relations in a unified graph representation.

To this end, we introduce a graph formalism that models environment elements and their relationships in an explicit, integrated representation. This choice is motivated by two merits: 1) it makes reachable information explicit by encoding schemas, data entities, tools, and their relations in a shared structure; 2) it makes version-linked evolution programmable: graph transformations can add, iterate, or remove capabilities while preserving unmodified structure and propagating changes via coupled components.

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`, `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:

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

where $\mathcal{G}^{(0)}$ is a seed environment and each $\Delta^{(k)}$ applies a structured operation to the current version. Each transformation is version-linked: it is conditioned on $G^{(k-1)}$, preserves components not explicitly modified, and updates affected relations so the resulting environment remains executable and evaluable. Fig. 2 illustrates this process. Adding a new capability is not treated as an isolated tool insertion: the graph edit introduces the required entities and attributes, connects them to existing schemas via dependency edges, and exposes the corresponding tool-enabled transitions. This makes the intended change explicit before it is implemented as code.

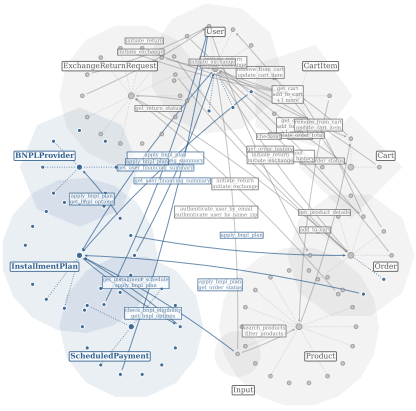

Figure 2: **Programmed Environment Evolution with two versions** The evolution is programmed via graph transformations, expressed as editions on nodes (schemas, e.g., `InstallmentPlan`) and edges (tool-enabled transitions, e.g., `get_installment_schedule`) that derives evolutions with existing structure.

The following sections instantiate this formalism along three steps: programming graph transformations to produce executable environment versions (Sec. 4.2), generating version-linked task sandboxes from each evolved graph (Sec. 4.3), and evaluating agents through graph-grounded, information-centric criteria (Sec. 4.4).

4.2 Programming Environment Evolution via Graphs

Whereas the graph formalism enables *controllable* evolution through graph transformations (Eq. 1), manually composing coherent graph transformations $\Delta^{(k)}$ for diverse evolution patterns does not scale. To enable scalable and diverse environment generation, we introduce an agentic pipeline that *automatically* programs evolutions in 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}) introduces new capabilities by adding the required schema elements, data entities, and tool-enabled transitions. For example, adding buy-now-pay-later support requires new financing entities, eligibility relations, and checkout-related tools.
- `Saturation` (Δ^{sat}) reorganizes access paths by adding shortcut tool-enabled transitions over useful multi-hop relations. For example, a multi-step path from user identity to purchased products can be compressed into a direct purchase-history tool.
- `Deprecation` (Δ^{dep}) removes or disables nodes and edges to model controlled capability removal or access loss, while preserving valid alternative paths when required.

These primitives are not intended to exhaustively cover all possible environment changes. Instead, they instantiate common structured changes in tool-calling environments: capability addition, access-path reorganization, and capability removal. Additional primitives can be incorporated as long as they can be expressed as semantic intents and graph operations under the same validation pipeline, including both further topology edits and attribute-level edits, e.g., signature-preserving tool-behavior drift expressed as changes to a tool edge's preconditions or output semantics.

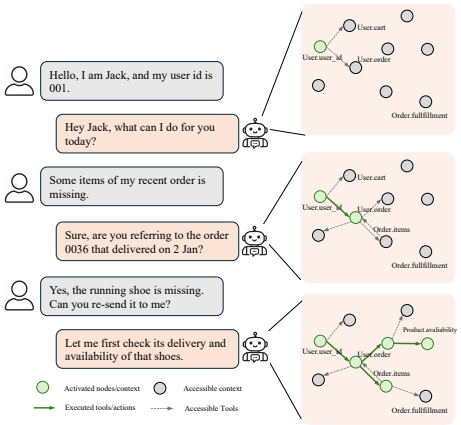

Figure 3: **Context as subgraph expansion in a tool-mediated conversation.** At each turn, the environment exposes a reachable context subgraph (gray nodes; dashed arrows for reachable tool transitions), while the agent activates a subset of nodes (green) by executing tools/actions (solid arrows) conditioned 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).

Phase II: Implement and Validate:

Given the transformed graph $\mathcal{G}^{(k)}$, ProEvolve instantiates an executable environment version by updating schemas, data models, tool implementations, and graph-grounded tests. The graph serves as the structural specification for code generation, ensuring that implementation changes are derived from explicit dependencies rather than disconnected edits.

Each generated version is validated along three axes. **Graph validity** checks whether the graph edits preserve schema consistency, reachability, and dependency constraints. **Implementation validity** checks whether the generated code compiles and whether graph-grounded unit tests detect a blocking implementation defect. **Semantic-intent alignment** checks whether the implemented change realizes the intended evolution $\Delta^{(k)}$, rather than merely passing tests. Versions that fail critical checks are filtered before being used for task generation or downstream evaluation.

This graph-first design addresses the two desiderata in Section 3. First, representing each evolution step as explicit graph edits preserves scalable coherence: new capabilities, schema changes, and tool-enabled transitions are introduced through a unified dependency structure rather than as disconnected modifications. Second, graph-grounded specifications support quality assurance before a version is advanced to the next stage: each version must pass hard, executable gates on graph validity, code compilation, and the absence of blocking defects detected by graph-grounded unit tests, while semantic alignment with the intended evolution is further assessed by rubric-based analysis (Sec. 5.2). This helps contain error accumulation across successive versions.

4.3 PROGRAMMING TASKS AS SUBGRAPHS

Given an evolved environment version $\mathcal{G}^{(k)}$, task generation must remain grounded in the capabilities and constraints of that version. We therefore formulate each task as a constrained subgraph $\mathcal{G}_\tau \subseteq \mathcal{G}^{(k)}$, which specifies the schemas, data dependencies, and tool-enabled transitions needed to satisfy the task objective. Since $\mathcal{G}_\tau$ is sampled from the current graph, each task reflects the tools, reachable information, and constraints available in that version, making task sandboxes version-linked rather than drawn from a environment-agnostic generation.

The task programming consists of the following steps:

- **`Subgraph Sampling.`** We sample a connected task subgraph $\mathcal{G}_\tau \subseteq \mathcal{G}^{(k)}$ and synthesize a task-level goal g_τ and scenario description s_τ conditioned on the sampled structure.
- **`Sandbox Materialization.`** Given $(\mathcal{G}_\tau, s_\tau)$, we instantiate prerequisite entities, attributes, and cross-references according to the relations in $\mathcal{G}_\tau$, ensuring that required tool transitions are executable under the current environment version. When a scenario involves multiple prerequisites, we compose them by jointly instantiating the prerequisite nodes and edges induced by $\mathcal{G}_\tau$ and enforcing shared identifiers and cross-entity references.
- **`Agentic Walk Execution.`** We then perform an oracle traversal over $\mathcal{G}_\tau$ to produce state-wise user instructions, graph-derived success criteria, and a reference trace of tool actions and activated graph elements. Tool outputs are converted into new obtained facts and accumulated as:

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

  Meanwhile, the active task context is represented as a subgraph $\mathcal{A}_t \subseteq \mathcal{G}_\tau$ that expands with executed actions and revealed information:

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

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

Sandbox output. Each task instance τ produces a version-specific sandbox containing: (1) an initial state $\mathcal{S}_0$ with materialized prerequisite entities and cross-references; (2) state-wise user instructions $\{u_t^\star\}_{t=1}^T$ paired with graph-derived success criteria $\{\mathcal{Y}_t^\star\}_{t=1}^T$; and (3) a reference graph trajectory recording oracle actions, activated nodes and edges, and context expansions. Since all components are derived from $\mathcal{G}_\tau \subseteq \mathcal{G}^{(k)}$, each sandbox remains grounded in the capabilities, constraints, and reachable information of its corresponding environment version.

4.4 STATE-WISE USER SIMULATION AND EVALUATION

Under structured evolution, tool names, schemas, and access paths may change across versions, so exact tool-matching becomes brittle. We therefore evaluate task progress at the level of graph-derived information: whether the agent obtains and uses the facts required by each state of tasks.

State instruction and criterion. For each state t, we construct an instruction tuple $\mathcal{I}_t \;=\; \big(u_t^\star,\; \mathcal{Y}_t^\star\big)$, where $u_t^\star$ is the synthesized user instruction and $\mathcal{Y}_t^\star$ denotes the minimal set of graph-grounded facts required to satisfy the state. In practice, $\mathcal{Y}_t^\star$ is derived from the frontier targets and extracted facts in the reference trajectory, so the criterion is grounded in the task subgraph $\mathcal{G}_\tau \subseteq \mathcal{G}^{(k)}$.

Progression rule. The user simulator follows the state-wise instructions and advances from state t to $t+1$ only if the agent satisfies $\mathcal{Y}_t^\star$, judged by schema-aware extraction from tool outputs and response text. Otherwise, the simulator remains at the current state and issues a follow-up clarification.

State success rate. Let $s_t \in \{0, 1\}$ denote if the agent satisfies state t. The state-wise success rate is: $\mathcal{C}(\tau) = \frac{1}{T}\sum_{t=1}^{T} s_t$. Because each $Y_t^\star$ and instruction $u_t^\star$ are derived from the task subgraph expansion (Eq. 3), $\mathcal{C}(\tau)$ measures dependency-consistent progress through the evolved environment. This yields stable user simulation and supports an information-centric, tool-agnostic evaluation: rather than matching fixed tool sequences or explicit tool annotations, we assess whether the agent acquires and uses the required information. This is especially suitable for structured evolution, where tools, schemas, or access paths may change while equivalent task progress remains valid.

5 EXPERIMENTS

We evaluate ProEvolve primarily as a benchmark-construction framework. Our experiments are organized around three questions:

Q1. Validity: Are evolved environments executable and semantically aligned?

Q2. Quality: Are environments and task sandboxes diverse, version-linked, and nontrivial?

Q3. Utility: Do evolved environments induce meaningful agent-behavior differences?

5.1 BENCHMARK CONSTRUCTION SETUP

Domains. We construct benchmarks in two structured tool-calling domains. E-commerce contains 1,000 products sourced from WebShop (Yao et al., 2022), 50 synthesized users, 51 executable tools, and 64 schema nodes. Airline booking provides a second evaluation domain adapted from τ^2-bench (Barres et al., 2025). For each seed environment, we construct an environment graph from the available schemas, data entities, and tools.

Environment Evolution. Starting from each seed graph, we instantiate evolution trajectories using the primitive sequence, i.e., $\mathcal{G}^{(0)} \xrightarrow{\Delta^{\text{comp}}} \mathcal{G}^{(1)} \xrightarrow{\Delta^{\text{sat}}} \mathcal{G}^{(2)} \xrightarrow{\Delta^{\text{dep}}} \mathcal{G}^{(3)}$ corresponding to capability addition, access-path reorganization, and controlled capability removal. We generate 50 e-commerce trajectories and 5 airline trajectories, containing 150 and 15 evolved-version candidates, respectively, or 200 and 20 trajectory-version instances when the shared seed is counted in each four-stage trajectory.

Table 1: **Benchmark scale and validation summary.**

	E-com.	Airline
1. Benchmark scale		
Evolution trajectories	50	5
Evolved-version candidates	150	15
Task sandboxes	3,000	300
2. Executable checks		
Unit-test pass rate	90.8%	93.1%
Defect-free versions	96.0%	86.7%
Oracle-replay solvable	100%	100%
3. Rubric judges (1–5)		
Environment edits		
Plausibility	4.76	4.78
Graph fidelity	4.63	4.71
Code fidelity	4.54	4.60
Sandboxes		
Scenario coherence	4.75	4.81
Transition feasibility	4.50	4.06
Prerequisite sufficiency	4.62	4.88
4. Environment diversity		
Pairwise tool Jaccard ↑	0.976	
Unique-tool ratio ↑	0.645	
Tool-frequency entropy ↑	8.07	

Task Generation. For each environment version, we generate task sandboxes using the subgraph-based pipeline in Sec. 4.3. Each task is sampled from the corresponding evolved graph and materialized with version-specific entities, prerequisites, state-wise instructions, and graph-derived success criteria. We balance task difficulty with an equal ratio of easy, medium, and hard tasks.

Downstream Evaluation. Agents interact with a state-wise user simulator under the protocol of Sec. 4.4. We report state-wise success $\mathcal{C}$, average turns $\bar{T}$, and average tool calls $\bar{N}_{\text{tool}}$. We evaluate GPT-5, Claude-Opus-4.5, DeepSeek-V3.2, Qwen3-235B, Gemini-2.5-Pro, GPT-6 Sol, Claude-Opus-5.5, and Kimi K3.

5.2 GENERATION QUALITY OF EVOLVED ENVIRONMENTS AND TASKS

We answer Q1 with executable checks and hard gates, and rubric-based judge analysis, and Q2 with difficulty separation and environment diversity; Table 1 summarizes the results.

Executable Checks and Hard Gates. Every candidate version is compiled and tested with graph-grounded unit tests covering modified code. Only genuine implementation defects block a candidate from seeding the next evolution stage; test-side errors and corner cases do not (Appendix A.1). The blocking-defect pass rates are 96.0% in e-commerce and 86.7% in airline. Such defects account for only 0.7% of pooled e-commerce test executions but can be consequential, e.g., a missing import that breaks dependent tests. Filtered env-versions are regenerated in place before later stages or task construction. A sandbox candidate is released only if its reference oracle satisfies every state-

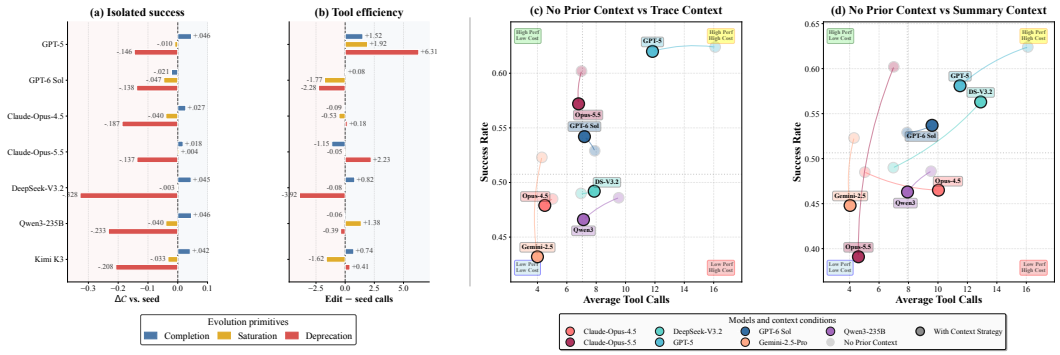

Figure 4: **Deprecation consistently degrades agents; prior-version context rarely helps.** (a) Success change from the seed when each primitive is applied independently on matched tasks. (b) Tool-call change (edit − seed). (c, d) Context-prior probes over $\mathcal{G}^{(1)}$–$\mathcal{G}^{(3)}$; arrows point from no prior context (faded) to Trace or Summary Context for the same model.

wise criterion on replay; candidates that fail this gate are discarded and resampled. All released sandboxes—3,000 in e-commerce and 300 in airline—pass oracle replay. This is a generation-time solvability gate, not an agent rollout success rate. These gates are decided by execution, independent of any LLM judgment.

Rubric-based Judge Analysis. As a secondary check on properties that executable gates cannot verify, an LLM judge (Claude Sonnet 4.5) scores each evolution edit on plausibility, graph fidelity, and code fidelity, and each sandbox on scenario coherence, transition feasibility, and prerequisite sufficiency (1–5 scale; rubrics in Appendix A.2). All mean scores lie between 4.06 and 4.88 across both domains (Table 1), and no evaluation scores ≤ 2. The Saturation graph-completeness score is 4.1, reflecting its shortcut serialization of a multi-hop path as a single endpoint-to-endpoint tool edge whose implementation is tested. We complement the sandbox scores with an independent human audit. The annotators' ratings average 4.33 for prerequisite sufficiency and 4.25 for state-transition feasibility, with quadratic-weighted $\kappa = 0.79$ and no rating ≤ 2 (Appendix A.5).

Diversity and the role of explicit structure. We compare against a free-form baseline that shares the seed, coding agent, and compute budget but receives no graph edits. ProEvolve produces more diverse evolutions: pairwise tool Jaccard distance is 0.976 vs. 0.867, and the unique-tool ratio is 0.645 vs. 0.394. It achieves this while adding roughly half as many tools per step (10.8 vs. 21.0), which indicates structural variety rather than volume (Appendix A.3).

Difficulty control. Graph-grounded difficulty levels also separate agent behavior: across models, success falls and interaction cost rises from easy to hard tasks (e.g., GPT-5: 0.73 → 0.45). Appendix B.1 reports the full success, turn, tool-call, and cost breakdown.

5.3 Diagnostic Utility: What Structured Evolution Reveals About Agents

Version linkage supports three diagnostic views that independently generated benchmarks cannot provide. First, a primitive can be applied to a shared seed while tasks, initial states, and success criteria are held fixed, which isolates its effect. Second, agents can be followed along a cumulative lineage. Third, the environment can be held fixed while only the prior-version experience available to the agent is varied. We organize the results by finding rather than by protocol, starting with the most controlled evidence. Full protocols, model coverage, and results are provided in Appendix B.

Table 2: **Behavior along cumulative evolution.** Each cell reports $\bar{\mathcal{C}}/\bar{T}/\bar{N}_{\text{tool}}$ on tasks sampled from that version's own graph. The columns characterize trajectory-conditioned behavior.

Model	$\mathcal{G}^{(0)}$ (Seed)	$\mathcal{G}^{(1)}$ (+Comp)	$\mathcal{G}^{(2)}$ (+Sat)	$\mathcal{G}^{(3)}$ (+Dep)
GPT-5	0.564 / 8.1 / 8.1	0.646 / 10.3 / 12.8	0.771 / 13.2 / 14.9	0.454 / 8.7 / 20.6
Claude-Opus-4.5	0.486 / 6.3 / 4.7	0.463 / 5.5 / 4.8	0.623 / 7.3 / 5.9	0.369 / 4.5 / 4.4
DeepSeek-V3.2	0.529 / 7.3 / 6.7	0.517 / 7.4 / 6.9	0.490 / 8.5 / 6.7	0.462 / 9.0 / 7.3
Qwen3-235B	0.588 / 10.4 / 9.5	0.633 / 12.9 / 13.1	0.460 / 9.0 / 8.1	0.364 / 7.6 / 7.4
Gemini-2.5-Pro	0.427 / 7.3 / 3.6	0.572 / 8.9 / 5.1	0.543 / 8.0 / 3.9	0.453 / 8.1 / 3.9
GPT-6 Sol	0.546 / 6.9 / 8.1	0.514 / 7.5 / 9.1	0.563 / 7.1 / 6.3	0.511 / 9.3 / 8.2
Claude-Opus-5.5	0.579 / 7.1 / 6.3	0.515 / 6.2 / 5.8	0.778 / 9.5 / 9.1	0.512 / 6.6 / 6.1
Kimi K3	0.541 / 7.9 / 7.2	0.407 / 5.2 / 4.0	0.426 / 5.1 / 3.9	0.510 / 9.1 / 10.2

Deprecation degrades agents on the matched workload. Applying each primitive independently to the seed on a common-support workload, where every task remains oracle-solvable through a verified alternative path (Fig. 4(a)), deprecation reduces success by 0.14–0.33 across all seven models. Completion and Saturation produce success shifts within $|\Delta\mathcal{C}| \leq 0.05$ on the same workload.

Tool use under deprecation differs across models. Holding the workload fixed exposes distinct interaction patterns (Fig. 4(b)). DeepSeek-V3.2 issues 3.9 fewer tool calls despite an available alternative path, whereas GPT-5 issues 6.3 more. The remaining models likewise differ in whether tool use increases or decreases after deprecation.

Lineage-conditioned behavior differs from isolated effects. Along the cumulative lineage (Table 2), success at $\mathcal{G}^{(3)}$ falls below the seed for seven of eight models, consistent with the isolated deprecation effect. Other stages do not transfer: the Saturation-stage peaks of GPT-5 (0.77) and both Claude-Opus models vanish when Saturation is applied alone ($|\Delta\mathcal{C}| \leq 0.04$). Because lineage results mix accumulated history with version-specific tasks, cumulative comparisons alone can suggest primitive effects that a matched design does not support; version linkage makes both views available.

Prior-version experience can become stale. Varying only the context from earlier versions over $\mathcal{G}^{(1)}$–$\mathcal{G}^{(3)}$ (Fig. 4(c, d)), Trace Context improves success by at most 0.013 across seven models, while Summary Context is riskier and model-dependent: Claude-Opus-4.5 doubles its tool calls without gain and Claude-Opus-5.5 loses 0.21. Failure decomposition shows that under Summary Context the agent more often claims that still-existing tools are unavailable (Appendix B.6); distilled experience carries outdated beliefs about tool availability into the current version.

6 Conclusions

We study structured environment evolution as a benchmark-construction problem for tool-calling agents. Our framework, ProEvolve, makes version-linked, dependency-aware changes programmable through explicit graph transformations, executable environment construction, and graph-grounded task generation. Experiments in e-commerce and airline booking demonstrate the feasibility of this approach and assess the validity, semantic fidelity, and diversity of the generated environments.

Beyond benchmark construction, version-linked diagnostics show that deprecation reduces success across models benchmarked despite valid alternative paths, while cumulative trajectories reveal model-dependent patterns in performance and tool use. Prior-version experience does not consistently improve task completion, highlighting the challenge of reusing experience as environments evolve. These findings illustrate the value of moving agent evaluation beyond fixed snapshots. Future work will extend the evolution primitives to policy changes, permission constraints, and tool-behavior drift.

AI USE STATEMENT

We used AI coding assistants to help configure the experimental infrastructure and implement experiment pipelines. We accessed language models through APIs for benchmark-data generation and model inference, as described in the paper. We also used AI tools to improve the language and clarity of the manuscript. The authors reviewed all AI-assisted code, generated data, experimental outputs, and manuscript edits, and take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

The environment representation and evolution primitives are specified in Sec. 3.1 and Sec. 4; benchmark construction and validation are described in Sec. 5.1; and the downstream evaluation setup is documented in Appendix B.0. The appendix also provides prompts and rubrics, per-stage validation results, model configurations, and statistical procedures. The generated environments, task sandboxes, initial states, success criteria, oracle traces, audit records, and evaluation code needed to reproduce the reported analyses are available in our anonymous repository: `https://anonymous.4open.science/r/ProEvolve-5F2F`.

ETHICS STATEMENT

This work studies the evaluation of tool-calling agents in controlled, evolving environments. Our experiments do not involve human subjects or private user data, and the generated environments and task instances are intended solely for research and evaluation. The purpose is to improve the robustness, reliability, and transparency of agent evaluation.

REFERENCES

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

Anthropic. Claude Opus 5.5. Online, 2026. URL https://www.anthropic.com/claude-opus-5-5.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

BerriAI. LiteLLM: Call 100+ LLM apis using the openai format. GitHub repository, 2023. URL https://github.com/BerriAI/litellm.

Guoxin Chen, Zhong Zhang, Xin Cong, Fangda Guo, Yesai Wu, Yankai Lin, Wenzheng Feng, and Yasheng Wang. Learning evolving tools for large language models. *arXiv preprint arXiv:2410.06617*, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, 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.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Xiang Lisa Li, Farzaan Kaiyom, Evan Zheran Liu, Yifan Mai, Percy Liang, and Tatsunori Hashimoto. Autobencher: Towards declarative benchmark construction. *arXiv preprint arXiv:2407.08351*, 2024.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv: 2308.03688*, 2023.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

OpenAI. Introducing GPT-6 Sol and Luna. Online, 2026. URL https://openai.com/index/introducing-gpt-6-sol-and-luna/.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

Dingfeng Shi, Jingyi Cao, Qianben Chen, Weichen Sun, Weizhen Li, Hongxuan Lu, Fangchen Dong, Tianrui Qin, King Zhu, Minghao Liu, et al. Taskcraft: Automated generation of agentic tasks. *arXiv preprint arXiv:2506.10055*, 2025.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 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.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Jingxu Xie, Dylan Xu, Xuandong Zhao, and Dawn Song. Agentsynth: Scalable task generation for generalist computer-use agents. *arXiv preprint arXiv:2506.14205*, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 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.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

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

Jiayi Zhang, Yiran Peng, Fanqi Kong, Cheng Yang, Yifan Wu, Zhaoyang Yu, Jinyu Xiang, Jianhao Ruan, Jinlin Wang, Maojia Song, et al. Autoenv: Automated environments for measuring cross-environment agent learning. *arXiv preprint arXiv:2511.19304*, 2025.

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

Appendix Table of Contents

Appendix A > Generation Quality Analysis
Appendix B > Full Agent Evaluation Results
Appendix C > Limitations
Appendix D > Version-Linked Task Sandbox Generation
Appendix E >Licenses for Existing Assets
Appendix F > Illustrative Examples of Structured Environment Evolution
Appendix G > Prompts and Implementation Details

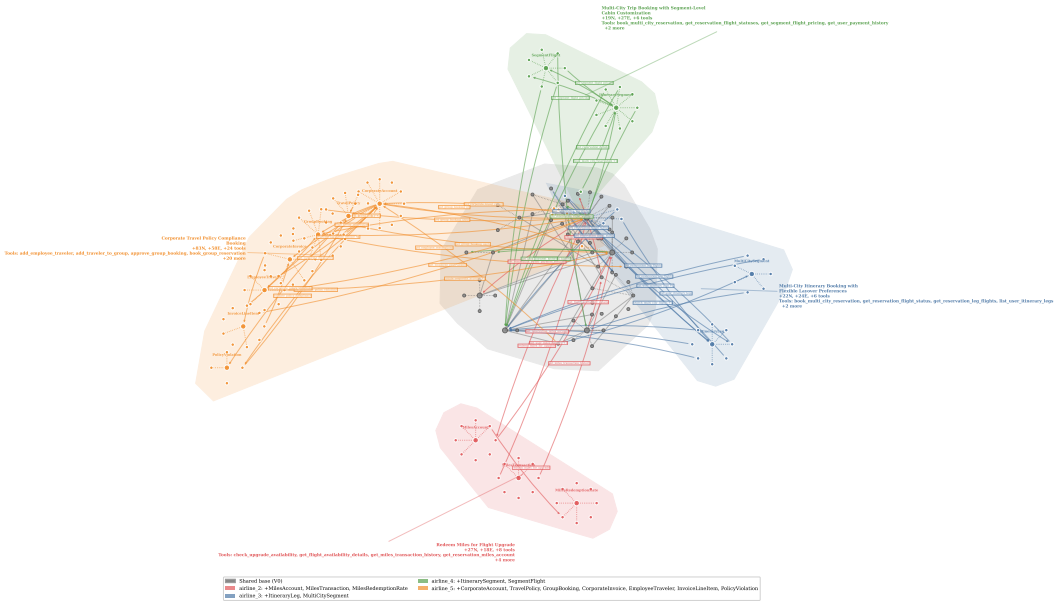

Figure 5: Layered topology of evolved airline environments. Grey nodes and edges represent the shared seed environment (V0: 10 databases, 56 nodes, 68 edges, 21 tools). Each colored region represents an independent evolution trajectory extending the base airline schema with new capabilities. Dotted edges denote intra-database connections; solid directed edges denote cross-database tool invocations. Node labels indicate newly added databases; edge labels show tool names introduced by each trajectory.

A Generation Quality Analysis

A.1 Implementation Validity and Test-Failure Breakdown

We categorize every failing unit test by the severity of its underlying cause, reflecting whether the failure indicates a genuine defect in the tool implementation or only a test-side artifact. The taxonomy has three levels:

- **(1) No impact.** The tool implementation is correct; the test itself carries a bug. Typical signals include string-label or enum casing mismatches, hardcoded entity identifiers absent from the fixture, pydantic validation errors in fixture setup, and attribute access on `MagicMock` / `FixtureFunctionDefinition` objects.
- **(2) Corner case.** The tool works on common inputs but fails on an edge case of limited practical impact—an optional field not populated, a `ValueError` on an uncommon argument combination, or an assertion on a seldom-used return shape.

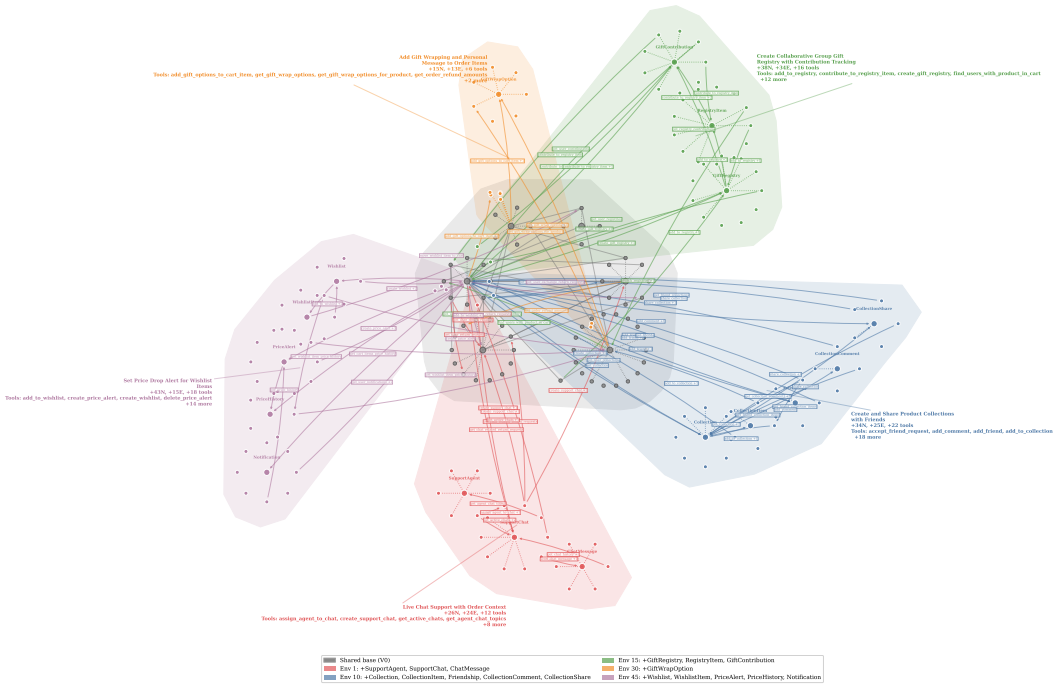

Figure 6: Layered topology of evolved e-commerce environments. Grey nodes and edges represent the shared seed environment (V0: 7 databases, 64 schema nodes, 63 graph edges, and 51 executable tools). For readability, the topology visualizes the 23 cross-database tool edges and omits 28 intra-database tools. Each colored region corresponds to a distinct evolution trajectory that adds new databases, attributes, and tool-mediated relationships to the base schema. Dotted edges denote intra-database attribute connections; solid directed edges denote cross-database tool invocations. Annotations on nodes indicate newly introduced databases; annotations on edges show the corresponding tool names.

- **(3) Partial impact (tool defect).** The tool implementation is genuinely broken. Typical signals are collection-time `ImportError` / `NameError`, `NameError` inside `tools_implementation.py` or `data_model.py`, missing tool methods, or module-level attribute omissions.

We say that an initial candidate *passes the blocking-defect gate* iff the test suite detects no level-3 implementation failure. Candidates that fail this gate are excluded from the released benchmark and regenerated in place.

Criticality distribution. The mean e-commerce unit-test pass rate across env-versions is 90.8%. In the pooled audit of 7,315 test executions, 36% of failures are level 1, 58% are level 2, and only 5.9% are level 3; level-3 failures correspond to roughly 0.7% of the pooled test executions. Airline failures are likewise dominated by the two non-blocking categories.

Representative examples. *(Level 1)* In `environment_48/V2`, the test fixture constructs `DeliverySlot(slot_type="morning")`, but the model's `Literal` only admits {`standard`, `express`, `weekend`, `holiday`}. The tool itself is correct; the fixture uses an out-of-vocabulary label.

(Level 2) `airline_1/V3` triggers an assertion on a seldom-used response field where the tool returns `None` instead of a default. Typical user flows are unaffected.

(Level 3) `environment_15/V1` calls `timedelta(days=1)` inside `tools_implementation.py` without importing `datetime.timedelta`, producing a

Table 3: Initial-candidate blocking-defect pass rates and failure criticality distributions. Category columns report shares among failed tests within each row. An env-version passes the blocking gate iff it contains zero level-3 implementation defects detected by the test suite. Rejected candidates are regenerated in place before release. Each row reports the first attempt at each (trajectory, stage) slot; later stages are built only on admitted predecessors.

Version	Pass rate	Cat-1 share	Cat-2 share	Cat-3 share
E-commerce initial batch				
V1 (completion)	92%	25.5%	64.5%	10.0%
V2 (saturation)	100%	51.0%	49.0%	0.0%
V3 (deprecation)	96%	49.2%	49.7%	1.0%
All	**96.0%**	36.0%	58.0%	5.9%
Airline initial batch				
V1 (completion)	100%	85.4%	14.6%	0.0%
V2 (saturation)	100%	78.9%	21.1%	0.0%
V3 (deprecation)	60%	0.0%	60.0%	40.0%
All	**86.7%**	77.8%	19.4%	2.8%

collection-time `NameError` that breaks downstream tests. This env-version is filtered from the pipeline and never serves as a base for V2 generation.

A.2 Semantic-Intent Alignment Rubric and Scores

We evaluate each individual evolution modification using an LLM-as-a-Judge (Claude Sonnet 4.5) that scores three dimensions, each decomposed into three sub-criteria on a 1–5 scale. Score 5 indicates *meets*; 3–4 indicates *partially meets*; ≤ 2 indicates *does not meet*.

Table 4: Scoring rubric for semantic fidelity.

Sub-criterion	What it checks
Plausibility. Whether the evolution is a plausible real-world change.	
Realism	The evolution reflects a change that could plausibly happen in production (e.g., feature addition, API consolidation, deprecation).
Coherence	The proposal has a clear user story and internally consistent entities/tools.
Complexity	The scope is well-calibrated for a single evolution step—non-trivial yet bounded.
Graph Fidelity. Alignment between implemented code and the specified graph edit.	
Completeness	All nodes/edges added or removed by the graph edit appear in the implementation.
Correctness	Field types, cardinalities, and tool placements match the graph specification.
Consistency	The new elements follow the existing graph's conventions (naming, FK patterns, edge structure).
Code Fidelity. Correctness of tool signatures and cross-entity relations.	
Model completeness	All required fields are present on the generated pydantic models.
Tool completeness	All tool methods specified by the graph are implemented.
Signature correctness	Tool parameter and return types match the graph edge definitions.

Per-sub-criterion results. Table 5 reports mean scores and the fraction of evaluations at each tier. Across all nine sub-criteria in both domains, **0% of evaluations score "does not meet"**. A score of 5 counts as "meet," while scores of 3–4 indicate that the core functionality is retained with partial differences in realization. The rubric separates functional coherence from exact edit fidelity.

Completeness by primitive. Table 6 reports graph and tool completeness for each evolution primitive. Saturation serializes a selected multi-hop path as an endpoint-to-endpoint shortcut edge and implements the corresponding shortcut tool.

Partial-compliance categories. Partial ratings primarily reflect differences in tool coverage, signatures, optional fields, or graph annotation detail. The released audit provides the item-level assignments and matched evidence spans.

Table 5: Per-sub-criterion LLM-as-Judge results on sampled trajectories in e-commerce and airline. Results aggregate 30 e-commerce and 15 airline evaluations.

	E-commerce			Airline		
Sub-criterion	**Mean**	**Meet**	**Partial**	**Mean**	**Meet**	**Partial**
Realism	4.87	90%	10%	5.00	100%	0%
Coherence	4.80	83%	17%	4.93	93%	7%
Complexity	4.60	60%	40%	4.40	40%	60%
Graph completeness	4.40	47%	53%	4.40	47%	53%
Graph correctness	4.77	80%	20%	4.80	80%	20%
Graph consistency	4.73	80%	20%	4.93	93%	7%
Model completeness	4.57	70%	30%	4.13	33%	67%
Tool completeness	4.50	50%	50%	4.73	73%	27%
Signature correctness	4.57	67%	33%	4.93	93%	7%

Table 6: **Completeness by evolution primitive** (e-commerce, LLM-as-a-Judge, 1–5). A score of 5 denotes exact realization, 4 a minor omission, and 3 retention of core functionality; each primitive contributes 10 evaluations.

Primitive	Graph completeness	Tool completeness
Completion (Δ^{comp})	4.7	4.5
Saturation (Δ^{sat})	4.1	4.3
Deprecation (Δ^{dep})	4.4	4.7

Per-strategy means. Mean scores exceed 4.3 for all three primitives. For e-commerce, Completion/Saturation/Deprecation yield Plausibility means of 4.67/4.80/4.80, Graph Fidelity means of 4.78/4.53/4.58, and Code Fidelity means of 4.33/4.67/4.63, respectively. Airline follows the same pattern.

A.3 Environment Diversity Metrics

Baseline protocol. We compare ProEvolve against a *free-form LLM evolution* baseline that evolves from the same seed environment but without graph-guided transformations. Specifically, the baseline uses the same coding agent with identical seed environment and matched compute budget, but receives no explicit graph edits—it is prompted only with a free-form description of the intended change (e.g., 'add a capability'), and generates code directly. Both pipelines run 50 evolution trajectories and produce $V^{(0)} \to V^{(1)}$ pairs; all diversity metrics below are computed over these 50 pairs per pipeline. This comparison measures the value of explicit transformation guidance over unconstrained code evolution.

Metrics. Let T_i be the set of tools in trajectory i. We report pairwise tool Jaccard distance, unique tool ratio, and tool-frequency entropy. Analogous metrics are computed on entity sets.

Observation. ProEvolve produces materially more diverse trajectories on every metric despite generating roughly half as many new tools per step (10.8 vs. 21.0). Rather than repeatedly reusing a small set of common tool patterns, the graph-guided pipeline induces structurally diverse environment variants.

A.4 Task Quality Analysis

We evaluate the intrinsic quality of generated task sandboxes using an LLM-as-a-Judge (Claude Sonnet 4.5) that scores each task along four dimensions on a 1–5 scale. We sample 200 task sandboxes balanced across difficulty levels (easy, medium, hard) and evolution primitives (seed, completion, saturation, deprecation). Table 8 defines the scoring rubrics; Table 9 reports aggregate results.

Table 7: Diversity comparison between graph-guided PROEVOLVE and a free-form LLM evolution baseline over 50 $V^{(0)} \to V^{(1)}$ pairs per pipeline. The baseline receives the same e-commerce seed, coding agent, and compute budget but no explicit graph edits. Deltas are computed from the unrounded columns.

Metric	PROEVOLVE	Free-form LLM	Δ
Tool diversity			
Pairwise Jaccard distance ↑	**0.976**	0.867	+0.109
Unique tool ratio ↑	**0.645**	0.394	+0.251
Tool-frequency entropy (bits) ↑	**8.07**	7.66	+0.41
Avg. new tools per trajectory	10.8	21.0	—
Entity diversity			
Pairwise Jaccard distance ↑	**0.979**	0.805	+0.174
Unique entity ratio ↑	**0.606**	0.336	+0.270
Entity-frequency entropy (bits) ↑	**6.18**	6.10	+0.08

Table 8: Scoring rubric for task quality evaluation. Score 5 = fully meets; 3–4 = partially meets; ≤2 = does not meet.

Dimension	What it checks
Scenario Coherence. Does the sequence of states form a realistic, logical customer journey?	
Narrative arc	States follow a natural progression; no random or disconnected actions.
Transition logic	Each state follows plausibly from its predecessor without forced jumps.
Difficulty Calibration. Does the task complexity match the claimed difficulty level?	
State count	Easy: 5–6 states; medium: 6–8; hard: 8–10.
Operation mix	Easy: mostly reads; medium: reads + simple writes; hard: multi-entity read/write chains.
State Transition Feasibility. Is each transition (State $N \to N{+}1$) executable given available information?	
Data handoff	Each state's outputs provide sufficient inputs for the next state.
No impossible jumps	The customer is never required to possess information not yet revealed.
Init Environment Sufficiency. Are prerequisite entities properly materialized?	
Entity coverage	All non-pickable entities (orders, carts, reservations) referenced by the task are initialized.
Cross-references	Shared identifiers and FK links across entities are consistent.

Judging conventions. Authentication after browsing is acceptable (browsing does not require `user_id`). User accounts and product catalogs pre-exist in the base database and need not be explicitly initialized. Empty `init_actions` is acceptable for tasks that create entities at runtime (e.g., search → cart → checkout).

Table 9: Task quality scores (1–5 scale) judged by LLM-as-a-Judge across sampled sandboxes.

Dimension	E-com.	Airline
Scenario Coherence	4.75	4.81
Difficulty Calibration	5.00	5.00
State Transition Feasibility	4.50	4.06
Prerequisite Sufficiency	4.62	4.88

Across all four dimensions, zero *LLM-judge* evaluations score "Fails" (≤ 2), indicating that graph-grounded task generation produces coherent and well-calibrated sandboxes. Executability is established separately by oracle replay (Sec. 5.2).

A.5 HUMAN AND SIMULATOR VALIDATION

This subsection separates deterministic checks from rubric judgments, reports an independent human audit of sandbox quality, and evaluates simulator decision consistency with two additional models.

What is established deterministically. Table 10 maps each validity aspect to its deterministic gate and to the rubric sub-criterion that complements it. Structural validity, reachability and dependency closure, syntax and import correctness, tool executability, modified-code coverage, defect filtering, and sandbox solvability are all decided programmatically. Test specifications are generated from the graph diff, but their compilation, execution, coverage, and pass/fail outcomes are deterministic. No claim in this column rests on an LLM preference judgment.

Table 10: **Deterministic validation gates vs. complementary rubric-based analysis.** Every aspect in the middle column is decided programmatically; the right column lists the rubric sub-criterion that provides a separate, stricter view of the same aspect. No claim in the middle column depends on an LLM preference judgment.

Evaluated aspect	Deterministic gate	Complementary rubric
Schema / structural consistency	Programmatic graph-validity checks	Graph correctness, graph consistency
Reachability & dependency closure	Programmatic reachability and closure checks before code generation	Proposal-to-graph alignment
Syntax & import correctness	Python compilation and symbol resolution	—
Tool executability	Executed graph-grounded pytest tests with bounded repair	Graph-to-code alignment
Modified-code coverage	Coverage measurement over changed code	—
Critical implementation defects	Severity-based filtering before the next evolution step	—
Sandbox solvability	Executable oracle traversal after deterministic sandbox materialization	Scenario coherence, transition feasibility

Annotators and sampling. Two non-author annotators rated the same 20 randomly sampled e-commerce sandboxes on prerequisite sufficiency and state-transition feasibility. The second annotator was blinded to all previous scores.

Human audit results. Mean prerequisite-sufficiency and state-transition-feasibility scores are 4.33 and 4.25, respectively (Table 11). The two annotators achieve quadratic-weighted $\kappa = 0.79$, 60% exact agreement, and 95% within-one-point agreement. No individual rating is ≤ 2.

Table 11: **Human audit of randomly sampled e-commerce sandboxes.** Values average two non-author annotators' ratings on the same items.

Dimension	Mean score
Prerequisite sufficiency	4.33
State-transition feasibility	4.25

Sandbox materialization and oracle replay. The deterministic prerequisite resolver is a fixed step of sandbox materialization. It initializes missing required entities, enforces user–entity references, and replaces invalid arguments with values already present in the sandbox database without changing the task goal or success criteria. Oracle replay then executes the reference tool sequence against the materialized initial state and requires every state-wise criterion to be satisfied. Candidates that fail replay are discarded and resampled. All released sandboxes—3,000 in e-commerce and 300 in airline—pass oracle replay and contain nonempty oracle traces, tool results, and states. This is a generation-time solvability gate, not an agent pass rate.

Simulator decision consistency. The user simulator is Claude-Opus-4.5 throughout. We replayed a shared set of simulator decisions with Qwen3-235B-2507 and DeepSeek-V3.2; Table 12 reports agreement with 95% confidence intervals. The consequential decisions — whether to terminate and whether a state is satisfied — are highly consistent across models. Raw agreement on the `[STATE:N]` marker is lower (0.409 / 0.437) because the two comparison models emit the marker far less often; the marker is not used by grounded state-wise scoring, and restricted to turns where both models emit it, state agreement is 0.853 / 0.846.

Table 12: **Simulator decision agreement.** The same 504 turns from 40 task clusters were replayed with Qwen3-235B-2507 and DeepSeek-V3.2 and compared against the Claude-Opus-4.5 simulator; brackets give 95% confidence intervals. State agreement is restricted to turns where both models emit a `[STATE:N]` marker; that marker is not used by grounded state-wise scoring.

Decision	Opus vs. Qwen	Opus vs. DeepSeek
Termination agreement	.948 [.917, .976]	.911 [.873, .949]
Success-judgment agreement	.912 [.788, 1.000]	.979 [.936, 1.000]
State agreement (both emit marker)	.853 [.798, .904]	.846 [.785, .903]

B Full Agent Evaluation Results

This section provides detailed agent-evaluation results supporting the compact diagnostics reported in Sec. 5.3. The main text orders the evidence by experimental control: matched single-edit comparisons, trajectory-conditioned aggregates, and context-prior probes. This appendix provides the corresponding protocols and detailed results.

B.0 Evaluation Setup and Model Coverage

Models. We evaluate `gpt-5-2025-08-07` Singh et al. (2025), `claude-opus-4-5-20251101` Anthropic (2025), `DeepSeek-V3.2` Liu et al. (2025), `Qwen3-235B-A22B-Thinking-2507` Yang et al. (2025), `Gemini-2.5-Pro` Comanici et al. (2025), `gpt-6-sol` OpenAI (2026), `claude-opus-5-5` Anthropic (2026), and `Kimi-K3` Team et al. (2026). All calls use LiteLLM BerriAI (2023).

Decoding and repeated runs. All model calls use temperature 0, with no top-p or top-k truncation. This disables deliberate sampling but does not make hosted inference bitwise deterministic: otherwise identical API requests can still produce different responses, which in turn lead to different agent–simulator trajectories. Repeated rollouts quantify this end-to-end run variation.

Simulator. Adapting the state-wise simulator pattern from τ^2-bench Barres et al. (2025), an LLM user simulator issues state-wise instructions and advances only when the current state's criterion is satisfied. The simulator is `claude-opus-4-5-20251101` unless otherwise noted. Its decision consistency is reported in Appendix A.5.

B.1 Difficulty Separation

Figure 7 breaks down state-wise success, turns, tool calls, and estimated API cost by graph-grounded task difficulty for the five models included in this analysis. Success decreases from easy to hard tasks across models, while harder tasks generally require longer and more tool-intensive interactions.

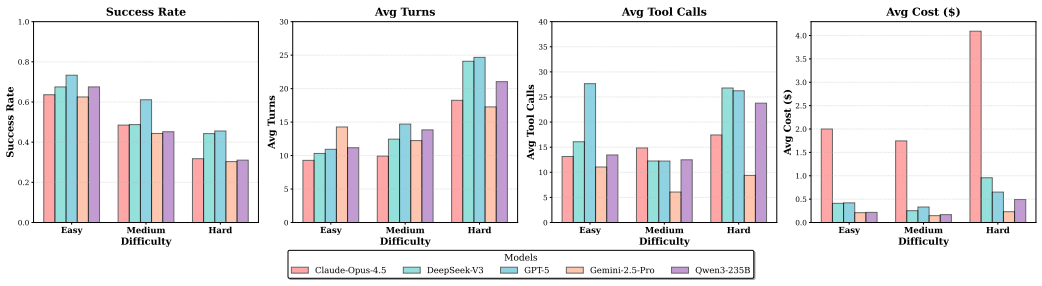

Figure 7: **Agent behavior by task difficulty.** State-wise success, average turns, average tool calls, and estimated API cost are aggregated by graph-grounded difficulty level. Harder tasks generally reduce success and require longer, more tool-intensive interactions.

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
Trace Context	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
Summary Context	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
Trace Context	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
Summary Context	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
Trace Context	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
Summary Context	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
Trace Context	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
Summary Context	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
Trace Context	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
Summary Context	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 13: **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 adaptation strategies (Baseline, Trace Context, Summary Context) for a given model, and the **Overall** columns summarize mean state-wise success $\mu_{\mathcal{C}}$ and average interaction cost aggregated over versions. Context-prior conclusions in the main text average only $\mathcal{G}^{(1)}$–$\mathcal{G}^{(3)}$, where prior versions exist; the Overall column here additionally includes $\mathcal{G}^{(0)}$.

B.2 Detailed Results: E-Commerce

Table 13 reports the full e-commerce breakdown behind the stage-wise aggregate results in the main paper: five models, three context conditions, and four environment versions along the trajectory

$$G^{(0)} \xrightarrow{\Delta_{\text{comp}}} G^{(1)} \xrightarrow{\Delta_{\text{sat}}} G^{(2)} \xrightarrow{\Delta_{\text{dep}}} G^{(3)}.$$

We report state-wise success $\mathcal{C}^{(k)}$, average turns $\bar{T}$, and average tool calls $\bar{N}_{\text{tool}}$ for each version. The versions are *cumulative*, with task sandboxes sampled for each version; these results characterize trajectory-conditioned behavior.

The results show substantial variation across both environment versions and agents. For example, GPT-5 with trace context increases from 0.562 at $\mathcal{G}^{(0)}$ to 0.786 at $\mathcal{G}^{(2)}$, but drops to 0.407 at $\mathcal{G}^{(3)}$. Other agents exhibit different trajectories and tool-use patterns.

B.3 Matched Single-Edit Protocol

This subsection documents the matched single-edit protocol.

Table 14: **Matched single-edit comparison.** Each primitive is applied *independently* and evaluated on the same common-support workload, with task instructions, initial state, entity IDs, required facts, and success criteria held fixed. Cells report $\mathcal{C}$ / $\bar{T}$ / $\bar{N}_{\text{tool}}$ as mean $\pm$ standard deviation across repeated rollouts; parenthesised values are success changes from the corresponding baseline arm, computed on unrounded values. Deprecation lowers success for every model despite a verified alternative path throughout the matched workload.

Model	Seed	Δ^{comp}	Δ^{sat}	Δ^{dep}
GPT-5	.657±.024 13.31±.45 14.23±.84	.703±.014 (+.05) 14.84±.39 15.75±1.02	.647±.039 (−.01) 14.23±.73 16.15±.92	.511±.030 (−.15) 18.77±.22 20.54±.74
Claude-Opus-4.5	.629±.032 14.68±1.19 12.28±.71	.656±.016 (+.03) 15.54±2.36 12.19±1.44	.589±.009 (−.04) 14.27±.70 11.75±.44	.442±.041 (−.19) 12.15±.29 12.46±1.00
DeepSeek-V3.2	.660±.023 13.67±.59 15.54±.95	.705±.015 (+.04) 13.87±.04 16.36±.82	.657±.031 (−.00) 13.82±.98 15.46±1.21	.332±.025 (−.33) 11.54±.38 11.62±1.45
Qwen3-235B	.697±.036 13.85±1.07 19.62±1.38	.743±.005 (+.05) 13.92±.78 19.56±.87	.657±.036 (−.04) 14.64±1.08 21.00±1.92	.464±.026 (−.23) 11.72±1.30 19.23±1.30

Table 15: **Matched single-edit results for GPT-6 Sol, Claude-Opus-5.5, and Kimi K3.** $\Delta\bar{T}$ and $\Delta\bar{N}_{\text{tool}}$ are edited minus baseline, and brackets give task-bootstrap 95% confidence intervals.

Model	Primitive	$\mathcal{C}$: seed → edit	$\Delta\mathcal{C}$	$\Delta\bar{T}$	$\Delta\bar{N}_{\text{tool}}$
GPT-6 Sol	Completion	.682 → .661	−.021 [−.075, .027]	+.05 [−1.10, 1.23]	+.08 [−1.41, 1.62]
	Saturation	.685 → .638	−.047 [−.088, −.006]	+.44 [−.74, 1.72]	−1.77 [−3.85, .08]
	Deprecation	.652 → .514	−.138 [−.202, −.069]	−2.56 [−4.59, −.28]	−2.28 [−4.44, .15]
Claude-Opus-5.5	Completion	.604 → .622	+.018 [−.033, .074]	−1.95 [−5.26, .72]	−1.15 [−4.03, 1.31]
	Saturation	.627 → .631	+.004 [−.051, .063]	−.10 [−1.38, 1.23]	−.05 [−1.41, 1.41]
	Deprecation	.651 → .514	−.137 [−.207, −.072]	+.10 [−1.08, 1.36]	+2.23 [.26, 4.05]
Kimi K3	Completion	.675 → .717	+.042 [.002, .089]	+.82 [−.49, 2.36]	+.74 [−.90, 2.36]
	Saturation	.709 → .676	−.033 [−.061, −.001]	−.77 [−2.41, .62]	−1.62 [−3.56, .28]
	Deprecation	.701 → .492	−.208 [−.277, −.140]	−1.69 [−3.46, .05]	+.41 [−1.72, 2.95]

Matched construction. Each primitive is applied independently to $\mathcal{G}^{(0)}$. Within each baseline–edit pair, we hold fixed the task instruction, initial database state, entity IDs, simulator disclosures, success criteria, and evaluation settings.

Common support. The workload contains 15 sandboxes whose success criteria remain reachable and oracle-solvable under all three edits.

Alternative-path verification under deprecation. For the deprecation variant, each required state remains accessible through a non-deprecated path used successfully by the reference oracle.

Aggregation. Each task is executed in three separate rollouts under the same inputs and decoding configuration. For each rollout, metrics are first aggregated over the complete matched workload; tables then report the mean and standard deviation across the three workload-level aggregates. Deltas use unrounded values, and per-task results are included in the released logs.

B.4 Cross-Domain Results: Airline

We use airline booking to examine whether the generated environment trajectories are executable and evaluable in another structured tool-calling domain.

Table 16 shows that the airline environments are executable and induce nontrivial evaluation signals across versions. As in e-commerce, versions are cumulative and task sandboxes are sampled independently per version, so these results are trajectory-conditioned. The trends are model- and probe-dependent: for Opus-4.5, Summary Context substantially reduces overall success and increases tool usage, whereas Trace Context is more stable; for DeepSeek-V3.2, Trace and Summary Context

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	$\mathcal{C}^{(0)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mathcal{C}^{(1)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mathcal{C}^{(2)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mathcal{C}^{(3)}$	$\overline{T}$	$\overline{N}_{\text{tool}}$	$\mu_{\mathcal{C}}$	Cost
Claude-Opus-4.5														
Baseline	0.58	11	12	0.49	7	9	0.54	9	10	0.27	9	12	0.47	$0.68
Trace Context	0.50	9	18	0.35	7	10	0.58	8	12	0.43	10	11	0.47	$0.70
Summary Context	0.60	11	22	0.29	7	14	0.25	6	14	0.20	9	16	0.34	$1.02
DeepSeek-V3.2														
Baseline	0.35	13	16	0.34	12	10	0.53	14	22	0.49	14	13	0.43	$0.21
Trace Context	0.60	16	18	0.33	8	9	0.56	9	11	0.33	9	8	0.46	$0.14
Summary Context	0.60	11	12	0.40	10	16	0.39	11	15	0.47	12	13	0.46	$0.17
Qwen3-235B														
Baseline	0.33	9	11	0.20	9	7	0.33	10	9	0.31	11	13	0.29	$0.07
Trace Context	0.43	9	8	0.30	12	10	0.37	9	11	0.35	12	15	0.36	$0.08
Summary Context	0.23	9	11	0.41	12	11	0.23	10	12	0.47	11	14	0.34	$0.09

Table 16: **Airline domain: evolving-trajectory evaluation.** We report per-version state-wise 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: Δ^{comp}, Δ^{sat}, and Δ^{dep}. The **Overall** columns report mean success $\mu_{\mathcal{C}}$ and estimated API cost per task.

yield similar aggregate success but different interaction budgets; Qwen3-235B remains lower overall but varies across versions.

B.5 Context Probe Details

Purpose. We use context conditions to diagnose how prior-version experience affects agents as tools, schemas, and access paths change.

Context conditions. We compare three conditions. **No Prior Context** solves each task independently without prior-version experience. **Trace Context** provides the agent with previous interaction traces from earlier environment versions. **Summary Context** provides distilled natural-language summaries of prior experience instead of raw traces.

Implementation. The following protocol is shared by all seven models in the context-prior analysis. For each environment version, context information is constructed only from previous versions in the same evolution trajectory. Trace Context retains the last five prior task traces; when the accumulated message history exceeds this window, older traces are dropped. Summary Context uses the evaluated agent itself to generate a two-to-four-paragraph reflection from the completed conversation, reward, and termination status, and retains the five most recent reflections. The context is prepended to the agent prompt before the current task begins. At $\mathcal{G}^{(0)}$ no prior version exists, so the three conditions reduce to independent seed-version control runs and are excluded from context-prior aggregates. We keep the task instruction, simulator protocol, and evaluation metric unchanged across context conditions.

Interpretation. Context probes are useful as a diagnostic because prior-version experience is not guaranteed to remain valid under structured evolution. Trace context may help when changes preserve useful access paths, but it may also become stale after deprecation or access-path reorganization. Summary context can reduce prompt length, but distilled summaries may overgeneralize previous tool availability or schema assumptions.

B.6 Failure-Mode Analysis

We complement the aggregate context results with a failure-mode decomposition for Claude-Opus-4.5 under the same No Prior, Trace, and Summary Context conditions described in Appendix B.5. For each included failed trajectory, we inspect the first unmet state and assign one agent-side failure label. Figure 8 reports percentages among failures assigned one of the four labels below.

Failure taxonomy. We distinguish four agent-side failure patterns:

Table 17: Agent-side failure-mode taxonomy.

Failure Mode	Definition
Missed tool discovery	The agent does not know a relevant tool exists. It tells the user "I can't do that" or "I don't have access" when an appropriate tool is available. Failure at the discovery/awareness level.
Invalid parameters	The agent identifies and calls the correct tool but passes wrong arguments—wrong entity IDs, malformed field values, or incorrect types. The tool errors out or returns incorrect results.
Missing invocation	The agent understands what needs to happen and knows the tool exists, but does not proactively call it. It waits passively, provides text explanations, or lets the user drive the next step. Failure in initiative/sequencing, not knowledge.
Unrecovered error	A tool call returns an explicit error (e.g., "product not found"), and the agent fails to adapt—it retries with the same bad parameters, gives up entirely, or tells the user to try later instead of attempting an alternative approach.

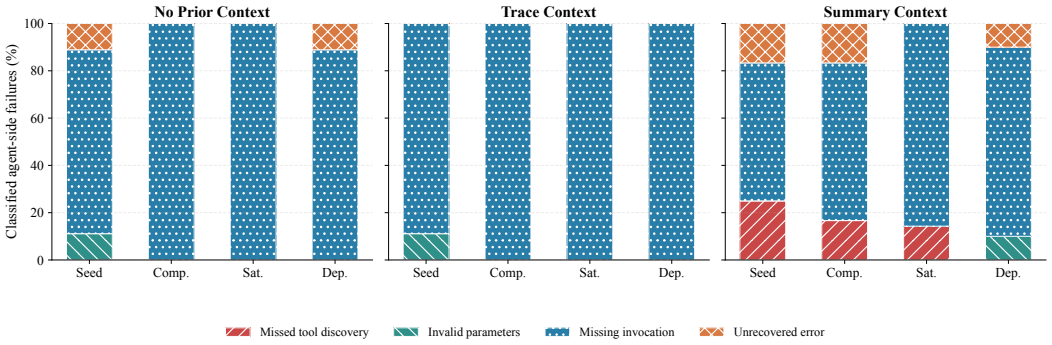

Figure 8: Distribution of classified agent-side failures across environment versions and context conditions (agent: Claude-Opus-4.5). Each bar normalizes over trajectories assigned one of the four categories in Table 17. Seed-version bars are no-context controls and are excluded from context-prior conclusions. Horizontal-axis labels Seed/Comp./Sat./Dep. denote the cumulative versions $\mathcal{G}^{(0)}$–$\mathcal{G}^{(3)}$.

Results across context conditions. Figure 8 compares the classified failure composition across the four cumulative versions under the three context conditions.

- **Missing invocation is the dominant classified mode.** It is the largest category at every version under No Prior Context and Trace Context, and remains the largest category under Summary Context.
- **Missed tool discovery appears under Summary Context.** This category appears at the earlier cumulative versions under Summary Context but not under No Prior or Trace Context. The association is consistent with summaries carrying stale beliefs about tool availability.
- **Invalid parameters and unrecovered errors are secondary modes.** They occur less often than missing invocation.

Implications. Within the classified subset, failures are concentrated in tool use and initiative rather than argument construction. Trace Context retains a composition close to No Prior Context, whereas Summary Context is associated with additional missed-discovery cases.

C LIMITATIONS

Diagnostic scope. PROEVOLVE is designed as a diagnostic instrument that complements human-designed tests, real usage logs, and domain-specific evaluations. **Domain scope.** The framework targets structured tool-calling environments with explicit schemas, entities, and tool-mediated transitions. Extending it to less structured or higher-stakes settings calls for domain-specific representations, expertise, and validation.

D VERSION-LINKED TASK SANDBOX 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.

Because the sampled task subgraph is drawn from the current environment version $\mathcal{G}^{(k)}$, the resulting task only references tools, entities, and reachable relations available in that version.

Step 1: Subgraph Sampling The framework first samples a connected subgraph $\mathcal{G}_\tau \subseteq \mathcal{G}$ 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.

Prerequisite composition. When a task requires multiple prerequisites, we jointly instantiate the prerequisite nodes and edges induced by the sampled subgraph. Shared identifiers and cross-entity references are enforced so that all required tool transitions are executable under the corresponding environment version. For example, an exchange task requires a valid user, a delivered order, a product variant mismatch, and an exchange-capable order state; these entities are materialized together rather than independently. A deterministic prerequisite resolver performs this composition as a fixed materialization step. It may initialize a missing required entity, enforce a required user–entity reference, or select a valid argument already present in the sandbox database, but it does not modify the task goal or success criteria. The materialized candidate is retained only if its reference oracle satisfies every state-wise criterion; otherwise, the sandbox is resampled.

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")`.

E LICENSES FOR EXISTING ASSETS

We list the licenses for all external assets used in this work:

- **WebShop** Yao et al. (2022): Released under the MIT License. We use the product catalog (1,000 products) as seed data for the e-commerce environment.
- **τ^2-bench** Barres et al. (2025): Released under the MIT License. We adapt the airline booking domain schema and seed environment specification as the starting point for our airline evolution trajectories.
- **LiteLLM** BerriAI (2023): Released under the MIT License. Used as the unified interface for making LLM API calls during evaluation.

All commercial LLM APIs (GPT-5, Claude-Opus-4.5, Gemini-2.5-Pro, DeepSeek-V3.2, Qwen3-235B, GPT-6 Sol, Claude-Opus-5.5, and Kimi K3) are accessed through their respective providers' standard API agreements. Our generated benchmark assets (environment versions, task sandboxes, and evaluation code) will be released under the MIT License upon acceptance.

F ILLUSTRATIVE EXAMPLES OF STRUCTURED ENVIRONMENT EVOLUTION

We provide representative evolution contexts in e-commerce and airline booking to illustrate how semantic evolution intents are translated into graph edits and executable environment changes. These examples cover the three primitives used in the main paper: Completion, Saturation, and Deprecation.

Table 18: Representative examples of structured environment evolution.

Domain	Primitive	Example capability/change
E-commerce	Completion	Competitor price monitoring / alerts
Airline	Completion	Redeem miles for flight upgrade
E-commerce	Saturation	Order refund summary shortcut
Airline	Saturation	Reservation flight status shortcut
E-commerce	Deprecation	Cart service unavailable
Airline	Deprecation	Direct-flight search deprecated

F.1 COMPLETION STRATEGY UNDER E-COMMERCE 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

F.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

F.3 SATURATION STRATEGY UNDER E-COMMERCE 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

F.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.

F.5 DEPRECATION STRATEGY UNDER E-COMMERCE 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.

F.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.

G PROMPTS AND IMPLEMENTATION DETAILS

This appendix provides the prompts and implementation details used in the two-phase environment evolution pipeline. Phase I proposes semantic evolution intents and translates them into graph edits. Phase II instantiates the transformed graph into executable code and validates the resulting environment version.

G.1 PHASE I: EVOLUTION PROPOSAL PROMPTS

G.1.1 COMPLETION: TASK PROPOSER

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

G.1.2 SATURATION: TOOL DESIGNER

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.

G.1.3 DEPRECATION: DEPRECATION SELECTOR

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.

G.1.4 GRAPH EVOLVER FOR ATTRIBUTE GRAPH UPDATES

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)

G.2 Phase II: Graph-to-Code Instantiation

G.2.1 Coding Agent

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}
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.

G.2.2 TEST SPECIFICATION GENERATOR

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 → Test instantiation, field validation, serialization - New tools → Test input/output contracts, error handling, edge cases - New relationships → 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_<function>_) - 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}
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.

G.2.3 Validation and Filtering

Validation and filtering pipeline. Each generated env-version passes through a cascading validation pipeline before it is admitted as a base for subsequent evolution steps. The pipeline has three stages, ordered by cost and specificity. *(i) Graph-validity checks* inspect the transformed graph $\mathcal{G}^{(k+1)}$ for schema consistency, reachability from `Input` nodes to newly added tools, and dependency closure (every referenced node and edge is defined). Graph edits that violate these constraints are rejected before any code is generated. *(ii) Syntax and import checks* compile every generated `data_model.py` and `tools_implementation.py` under the target Python runtime, then confirm that every symbol imported by the generated `test_tools.py` resolves against the implementation module. This catches truncated LLM outputs, missing class definitions, and unresolved references at the earliest possible point. *(iii) Unit-test execution* runs the graph-grounded tests with `pytest` to check that the new and modified tools behave as specified. These three stages are applied to every generated version. The LLM-as-a-Judge analysis in Appendix A.2 is a sampled semantic audit rather than a per-version admission gate.

Retry behavior. Failures at stages (ii) and (iii) trigger a bounded repair loop with up to $R{=}3$ retries. On each retry the coding agent is re-prompted with the failing code and the captured error (SyntaxError trace, ImportError, pytest failure output), and produces a targeted patch rather than regenerating the full file. This matches the stage of failure: syntax/import errors typically resolve in a single retry because the error message localizes the defect, whereas test-execution failures may need two to three rounds to converge. After R retries we stop to bound cost; the filtering rule below then decides whether the candidate can seed the next evolution stage.

Filtering rule for critical defects. We treat an env-version as having a *critical tool-code defect* iff at least one failure reaches severity level 3 in the taxonomy of Appendix A.1, i.e., a genuine bug in `tools_implementation.py` or `data_model.py` (e.g., collection-time `ImportError`/`NameError`, missing tool method, undefined symbol inside tool code). Such env-versions are *filtered*: they are not admitted as a base $\mathcal{G}^{(k)}$ for the next evolution step, which prevents defects from propagating and compounding across the trajectory. Filtered env-versions are regenerated in place so that benchmark construction retains the target number of released environment trajectories. Env-versions with only level-1 or level-2 failures (test-side artifacts or minor corner cases) do not meet the blocking-defect criterion and may be retained.

Initial-candidate defect rates. The overall filtering rate is 4.0% in e-commerce and 13.3% in airline. Inspecting the filtered cases, the defects cluster into three recurring patterns: (a) missing standard-library imports used in the new code (e.g., `timedelta` without `from datetime import timedelta`); (b) classes referenced in tests that were placed in `tools_implementation.py` instead of being exported from `data_model.py`; and (c) data-model entities that the graph edit introduced but the coding agent failed to serialize (e.g., `FlightInstance`, `ServiceUnavailableError`). No V2 (saturation) candidate in the initial batch triggers the blocking-defect gate; we report this as an observation of the audited batch rather than as a primitive-specific defect rate.

G.3 Runtime Configuration and Model Selection

All components of the ProEvolve pipeline—benchmark generation, task generation, and evaluation—use the following unified configuration.

Generation backbone. Both Phase I (evolution proposal: task proposer, graph evolver, saturation tool designer, deprecation selector) and Phase II (coding agent, test specification generator) use `claude-opus-4-5-20251101`.

Task generation. The subgraph-based task pipeline (Sec. 4.3)—including scenario synthesis, sandbox materialization, agentic walk execution, and state-wise instruction generation—also uses `claude-opus-4-5-20251101`.

User simulator. During downstream agent evaluation (Sec. 4.4), the state-wise user simulator that issues instructions and judges state satisfaction is powered by `claude-opus-4-5-20251101`. It adapts the state-wise simulator pattern from τ^2-bench Barres et al. (2025): the simulator issues the instruction for the current state and advances only after that state's criterion is satisfied.

LLM-as-a-Judge. Semantic fidelity evaluation (Appendix A.2) and task quality assessment (Appendix A.4) use `claude-sonnet-4-5-20250929`.

Model identifiers. The LiteLLM/Bedrock identifiers are `bedrock/us.openai.gpt-6-sol`, `bedrock/us.anthropic.claude-opus-5-5`, and `bedrock/us.moonshotai.kimi-k3`.