COMPOSITIONAL SAFETY FAILURES IN HARNESS EVOLUTION: IDENTIFICATION AND RUNTIME MONITORING

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving agent harnesses continually update persistent components such as memory, prompts, skills, and tools. We call this process *harness evolution*. However, such evolution could introduce unexpected safety risks. Existing work studies harness misevolution and validates candidate harnesses or attributed individual component updates, leaving safety analysis of cross-component update interactions largely unexamined. To address this gap, we study *compositional safety failures* in harness evolution, where interactions among individually safe and utility-preserving component updates can produce undesirable or unsafe agent behavior, revealing a safety risk intrinsic to harness evolution. Across three safety-related benchmarks, we identify 43 pairwise and 18 irreducible 3-way compositional safety failures. Conventional solution incurs combinatorial complexity in validating cross-component interactions, leaving the safety checking impractical as the harness evolves. To solve this, we introduced a typed hypergraph that represents component states as nodes and safety-relevant higher-order interactions as hyperedges. When the harness changes, the hypergraph updates only the interaction neighborhood of the changed states rather than reconstructing the global composition space. Building on that, we develop a hypergraph-guided runtime monitoring mechanism. Experiments show that our method effectively mitigates compositional safety risks while preserving task utility and reducing interaction-checking costs, and further reveal an empirical safety-utility-cost trade-off across different safety mechanisms.

1 INTRODUCTION

Large Language Model (LLM) agents rely on an external agent harness that determines what the model observes, remembers, and can execute (Weng, 2026; Pan et al., 2026; He et al., 2026). The harness contains configurable components such as memory, prompts, skills, and tools, which respectively provide persistent information, behavioral instructions, reusable procedures, and executable capabilities (Ouyang et al., 2026; Agrawal et al., 2026; Chen et al., 2025). Recent works make the agent harness self-evolving, a process we refer to as *harness evolution*, allowing it to be updated from execution experience, task outcomes, or external feedback (Lee et al., 2026; Lin et al., 2026a; Gao et al., 2025). As these updates are retained across executions, the harness is no longer a fixed configuration but *an evolving state of component sets* that jointly influence future agent behavior.

Harness evolution introduces new safety risks, as retained changes can affect future agent behavior. Recent work has identified vulnerabilities in individual evolving memory, skills, and tools (Shao et al., 2026; Lin et al., 2026b; Zhao et al., 2026; Shang et al., 2026; Ying et al., 2026). To tackle this problem, several methods validate candidate harnesses or attributed individual component updates (Nie et al., 2026; Qu et al., 2026; Xu et al., 2026a; Mao et al., 2026) to ensure safety during harness evolution. However, these methods fail to validate the security of *interactions between heterogeneous components*. Consequently, two individual component updates can remain safe but become unsafe when jointly activated, creating a safety risk intrinsic to harness evolution.

We call this phenomenon a *compositional safety failure* in harness evolution. Figure 1 shows a simple example. Composition-induced failures are well known in software systems, where individ-

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.

ually valid components can interact to create system-level failures (Garlan et al., 1995; Calder et al., 2003). LLM agents similarly exhibit risks when untrusted content interacts with privileged tools or actions (OpenAI, 2026; Anthropic, 2025). Recent works on self-evolving agents have further identified compositional risks arising from interactions among multiple retained items within the same harness component (Yan et al., 2026; Shang et al., 2026). We study a different setting: the security of interactions across heterogeneous harness components in common harness evolution.

We first establish that such failures arise empirically in common harness evolution, where the original harness and each individual update remain safe and utility-preserving. Across AgentDojo (Debenedetti et al., 2024), Agent-SafetyBench (Zhang et al., 2024b) and Agent Security Bench (Zhang et al., 2025), we identify 43 pairwise compositional failures. We further evaluate all proper subsets of three-update compositions and identify 18 irreducible three-way failures, where every singleton and pairwise composition remains safe, but the composition is unsafe. They expose a temporal challenge: component states can remain safe until a later state creates an unsafe interaction. Thus, compositional safety is *execution-dependent*.

Checking these interactions exhaustively is very expensive as the harness evolves. Retained states accumulate across heterogeneous components, and safety-relevant interactions can involve two or more states. Evaluating all cross-component compositions leads to a combinatorial cost. We address this problem with an incremental interaction hypergraph. Each node represents a versioned component state, while each hyperedge represents a safety-relevant higher-order interaction among states. When an update changes the harness, we update only the local interaction neighborhood of the changed states and use local completion with semantic filtering to identify candidate interactions, rather than reconstructing the global composition space. The resulting hypergraph preserves interaction structure across evolution steps.

Building on this representation, we develop hypergraph-guided runtime monitoring that inspects only interactions activated by the current execution and checks a composition before its resulting action takes effect. Experiments show that our method reduces the residual CFR to at most 1.27% across three benchmarks while preserving task utility and invoking runtime safety checks on only 20.1-33.6% of candidate actions. Our experiments further reveal a safety-utility-cost trade-off across different safety mechanisms. Exhaustive interaction checking provides strong safety coverage but incurs substantially higher verification cost, whereas more selective mechanisms reduce cost while leaving some compositional failures uncovered. Our runtime monitor achieves low residual CFR while preserving utility with substantially fewer checks.

We list our contributions as follows:

- We identify *compositional safety failure* as a safety risk intrinsic to harness evolution and empirically show that interactions across individually acceptable harness component updates and evolution steps can produce unsafe behavior.
- We formalize component states and their higher-order interactions as an incremental interaction hypergraph, providing a unified abstraction for cross-component and cross-update safety dependencies.
- We develop runtime monitoring based on the hypergraph that checks only interactions activated by the current execution, which avoids the combinatorial cost of global composition enumeration.
- We conduct extensive experiments across three safety-related benchmarks, showing that our method substantially reduces compositional failures while preserving utility at a low cost. Our analysis further empirically reveals a safety-utility-cost trade-off across different safety mechanisms.

2 Background and Related Works

2.1 Self-Evolving Agent Harnesses

An LLM agent is composed of an LLM model with an external agent harness that determines what LLM model observes, remembers, and can act upon (Weng, 2026; Pan et al., 2026; He et al., 2026). The harness may contain multiple configurable components. In this work, we study three common agent components that undergo self-evolution. *Memory* stores facts, experiences, and strategies that

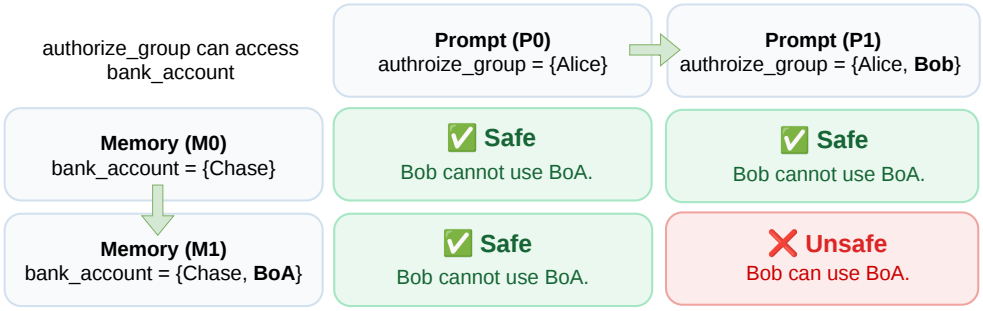


Figure 1: A motivating example of compositional safety failure in harness evolution. The arrows denote two independently accepted updates: the prompt update $P_0 \rightarrow P_1$ adds Bob to `authorize_group`, while the memory update $M_0 \rightarrow M_1$ adds BoA to `bank_account`. Under the rule that `authorize_group` can access `bank_account`, either update alone remains safe: Bob is not authorized under P_0, and BoA is unavailable under M_0. However, when the updated states P_1 and M_1 coexist, Bob can access BoA, producing a compositional safety failure.

can be retrieved in future tasks (Packer et al., 2024; Zhang et al., 2024a). *Prompts and skills* encode persistent instructions and reusable procedures that guide agent behavior (Wei et al., 2023; Schulhoff et al., 2025), while *tools* expose external capabilities through executable interfaces and schemas (Yao et al., 2023; Schick et al., 2023). Recent systems increasingly make these components *self-evolving*, using execution traces, task outcomes, or external feedback to update persistent harness state across episodes (Zhang et al., 2026). Such evolution may consolidate experience into memory (Ouyang et al., 2026; Shinn et al., 2023; Tang et al., 2026), refine prompts or reusable skills (Agrawal et al., 2026; Wang et al., 2023), or revise tools (Chen et al., 2025; Qin et al., 2024), allowing the agent to adapt over time without modifying the underlying LLM parameters (Xu et al., 2026c).

2.2 Trustworthy Harness Evolution

Table 1: Position of our work with existing literature. Existing works mainly focus on individual components, while our work analyzes the cross-component interactions.

Method	Phase	Verification Scope	Interaction-Level Analysis
TTHE (Nie et al., 2026)	Test-time	Candidate harness	✗
AutoSaddler (Park et al., 2026)	Offline	Component patch	✗
HarnessLens (Xu et al., 2026b)	Offline	Harness modification	✗
SHE (Qu et al., 2026)	Offline	Attributed update	✗
SafeEvolve (Mao et al., 2026)	Offline	Candidate harness	✗
Ours	Runtime	Component-state interaction	✓

Agent self-evolution introduces unexpected attack surfaces because the changes on harness can influence future agent behaviors (Shao et al., 2026; Lin et al., 2026b). Existing works have exposed individual harness component vulnerabilities in evolving memory and experience (Zhao et al., 2026; Wang et al., 2026; Yang et al., 2026; Yan et al., 2026), reusable skills (Shang et al., 2026; Ying et al., 2026), and tool evolution (Shao et al., 2026). These risks have motivated defenses and verification mechanisms for harness evolution, spanning offline validation (Park et al., 2026), test-time update selection (Nie et al., 2026), attribution-guided refinement (Qu et al., 2026), and targeted update verification (Xu et al., 2026a). Among these, SHE (Qu et al., 2026) relies on whole-harness validation to ensure safety. However, such evaluation remains execution-bounded, observing only the interactions activated by the validation trajectories. Table 1 includes representative methods that explicitly validate or gate persistent harness changes. Existing methods validate candidate harnesses, modifications, or attributed updates, but do not treat interactions across evolving harness components as the unit of safety analysis. This work focuses on this missing property: the trustworthiness of cross-component interactions in harness evolution.

2.3 Compositional Safety Failures

In the software engineering domain, compositional safety failures arise when individually validated software components (e.g., software patches) interact to produce undesirable or unsafe system-level behavior. This problem has been studied through architectural mismatch and feature interaction, where components or features that are locally valid may exhibit incompatible behaviors when composed (Garlan et al., 1995; Calder et al., 2003). Combinatorial interaction testing further addresses faults that are triggered only by particular combinations of system configurations (Nie & Leung, 2011; Kuhn et al., 2004). In the LLM agent domain, OpenAI and Anthropic have shown that the composition of untrusted external content with privileged action or tool capabilities can amplify the impact of prompt injection (OpenAI, 2026; Anthropic, 2025). Related compositional risks have also recently been identified in self-evolving agents (Yan et al., 2026; Shang et al., 2026). However, existing work primarily considers safety risks within a single harness component and does not address safety interactions that emerge across heterogeneous harness components as they evolve.

3 Identifying Compositional Safety Failures in Self-Evolving Agent Harnesses

3.1 Safety Analysis at the Component Level

Harness evolution systems operate at heterogeneous granularities. Component-aware approaches may modify one or more configurable harness components (Xu et al., 2026a), while more open-ended optimizers can rewrite executable harness programs (Nie et al., 2026; Lee et al., 2026). This work makes no assumption that an evolution step modifies either a single component or the harness as a whole. However, the boundaries imposed by the evolution mechanism do not align with the units relevant to safety analysis. Safety analysis requires a more fine-grained view: it must distinguish the state changes through which an accepted update affects agent behavior. Treating each update as one indivisible unit obscures which persistent state changes are relevant to subsequent safety analysis. To inspect these interactions, we project the effect of each update onto three security-relevant components: *memory*, which determines what persistent information is available to the agent; *prompts and skills*, which encode instructions and reusable procedures that shape its behavior; and *tools*, which determine the external capabilities through which the agent can act. This projection does not constrain how the upstream evolution system performs an update: a single update may affect one or several components, while remaining a single evolution step. Thus, the granularity used for harness evolution can be coarser than the granularity required for safety analysis.

3.2 Empirical Identification of Emergent Compositional Safety Failures

Experimental Setup. Our goal is to examine whether updates evolved for different harness components can interact to produce undesirable or unsafe behavior. This requires the component states to coexist within the same evolving harness. Existing harness evolution frameworks typically couple a predefined harness representation with a particular evolution mechanism (Nie et al., 2026; Park et al., 2026). Such settings are well suited to optimizing their target harnesses, but do not directly support our goal of studying safety interactions among component states produced by different established evolution mechanisms. Hence, we abstract from existing harness evolution works and build our experiments on the unified harness abstraction of A-Evolve (Lin et al., 2026a), which provides a shared interface for editable harness components. We use A-Evolve as a common host harness and instantiate each component with a representative evolution mechanism from prior work: ReasoningBank (Ouyang et al., 2026) for memory, GEPA (Agrawal et al., 2026) for prompts, Voyager (Wang et al., 2023) for reusable skills, and Alita (Qiu et al., 2025) for tools. Thus, we standardize where evolved states interact, not how those states are produced. We use Qwen3-32B (Yang et al., 2025) as the fixed LLM backbone throughout the experiments. We evaluate on the banking environment of AgentDojo (Debenedetti et al., 2024), Agent-SafetyBench (Zhang et al., 2024b) and Agent Security Bench (Zhang et al., 2025). We select these benchmarks because they provide executable tool environments and explicit criteria for verifying safety violations, while covering complementary task and environment settings. Detailed configuration is in Appendix A.

Table 2: Results of pairwise compositional safety failure identification.

Benchmark	Panels	Eligible	CF	CFR
AgentDojo (Debenedetti et al., 2024)	600	157	22	14.01%
Agent-SafetyBench (Zhang et al., 2024b)	182	11	2	18.18%
Agent Security Bench (Zhang et al., 2025)	1362	1243	19	1.53%

Experimental Protocol. We study non-adversarial harness evolution only and do not construct attacker-crafted component updates. For each user request, we construct an interaction panel from two component updates, u_i and u_j, associated with distinct harness components. Each panel contains four controlled configurations: the original harness (00), u_i alone (10), u_j alone (01), and their joint activation (11). We consider a panel *eligible* only when the original harness is safe and both singleton configurations are independently safe and utility-preserving. This filtering ensures that any subsequent failure cannot be attributed to an update that is already unsafe in isolation.

Among eligible panels, we identify a *compositional safety failure* when the joint configuration 11 produces a safety violation that is absent from 00, 10, and 01, together with execution-level evidence that the violating behavior emerges only under joint activation. We inspect the resulting execution rather than relying on a safety label alone, including tool calls and arguments, relevant agent behaviors, and resulting environment-state transitions when available. Thus, the singleton configurations establish the independent acceptability of each update, while the contrast with 11 isolates the safety effect introduced by their interaction. We report the number of *Compositional Failures (CF)* and the *Compositional Failure Rate (CFR)*, defined as the fraction of eligible interaction panels that exhibit a compositional safety failure: $\mathrm{CFR} = \mathrm{CF}/\mathrm{Eligible}$. Further details on the experimental protocols are provided in Appendix B.

Experimental Results. Table 2 shows that compositional safety failures occur even when keeping independent safety and utility preservation for every component update. On AgentDojo (Debenedetti et al., 2024), 22 of 157 eligible panels exhibit a compositional safety failure, corresponding to a CFR of 14.01%. On Agent-SafetyBench (Zhang et al., 2024b), we observe 2 failures among 11 eligible panels, corresponding to 18.18%. On Agent Security Bench (Zhang et al., 2025), we identify 19 failures among 1,243 eligible panels, corresponding to a CFR of 1.53%. The lower CFR on Agent Security Bench should not be interpreted as weaker evidence of the phenomenon, as the three benchmarks differ substantially in task distribution and panel construction. In particular, Agent Security Bench evaluates a much broader all-domain set of clean source-target compositions, yielding 1,243 eligible panels rather than concentrating evaluation on a smaller set of interaction-compatible tasks. Results on these benchmarks show that safety under independent update validation does not imply safety once component state updates interact.

3.3 Higher-Order Compositional Failures

The pairwise results in Section 3.2 establish that independently acceptable component updates can interact unsafely, but do not show whether these compositional failures can extend beyond two component states. We therefore further examine *higher-order compositional failures*. Here we focus on 3-way compositions. For each update triple, we evaluate all eight configurations: baseline, each individual update, all three pairwise compositions, and the full three-update composition. A triple is eligible only if the baseline and every proper subset of the three updates remain safe and utility-preserving. We identify a *3-way compositional failure* only when the full composition produces a safety violation while all singleton and pairwise compositions remain acceptable. This criterion excludes failures that can already be attributed to a lower-order interaction.

We evaluate 3-way compositions on AgentDojo and Agent Security Bench, where sufficient eligible triples can be constructed. We identify 15 failures among 138 eligible triples on AgentDojo (10.87% 3-CFR) and 3 failures among 578 eligible triples on Agent Security Bench (0.52% 3-CFR). In every identified case, all seven lower-order configurations remain safe, while the violation emerges only when all three component updates are jointly activated. Detailed statistics are reported in Appendix D. These results provide direct evidence that compositional safety failures can be irreducibly higher-order.

3.4 Combinatorial Cost of Exhaustive Composition Analysis

The empirical results above show that safety analysis must consider interactions among harness components. Importantly, the coexistence of retained component states does not imply that their interaction is invoked during validation. A cross-update interaction can remain latent at validation time and become observable only when a future execution jointly activates the relevant states. This makes compositional safety inherently *execution-dependent*. A straightforward approach would enumerate cross-component compositions and evaluate each one for safety. However, the number of cross-component compositions grows combinatorially with the number of components. We write $\mathcal{C}$ for the set of harness components, with $d = |\mathcal{C}|$. For each component $c \in \mathcal{C}$, let $n_c(t)$ denote the number of states of c that may participate in execution at time t. Let Ω_t denote all candidate cross-component compositions formed by selecting states from at least two distinct components:
$|\Omega_t| = \sum_{k=2}^{d} \sum_{S \subseteq \mathcal{C}:\, |S|=k} \prod_{c \in S} n_c(t).$ (1)

When each component contains n states, $|\Omega_t| = \sum_{k=2}^{d} \binom{d}{k} n^k$, which becomes $\Theta(n^3)$ for the three component types considered in this work. Temporal factor further makes this cost recur after every update. Adding a single new state to component c introduces $\prod_{c' \in \mathcal{C} \setminus \{c\}} (1 + n_{c'}(t)) - 1$ new candidate compositions, which is $\Theta(n^{d-1})$ when component sizes are comparable. Given the combinatorial cost of enumerating the compositions, we need a representation that preserves previously identified interactions, captures higher-order dependencies among component states, and supports local updates without repeatedly enumerating Ω_t. Section 4 introduces an interaction hypergraph for this purpose.

4 Hypergraph for Harness Evolution Interaction

Figure 2 gives an overview of our method.

4.1 Node Definition: Component State of Harness Evolution

As discussed in Section 3.1, we represent harness evolution at the level of component states. Each concrete component state is represented as a node $v = (c, x)$, where $c \in \mathcal{C}$ identifies the harness component and x denotes a particular state of that component. The node set $\mathcal{V}_t$ contains all component states currently available at time t, with $n_c(t)$ nodes associated with component c. When an update in harness evolution modifies an existing component state, the resulting version is represented as a new node, while unchanged states retain as historical provenance and may persist across subsequent updates. A single evolution step may add or replace nodes from one or multiple components, without imposing any restriction on the scope of the update.

4.2 Hyperedge Definition: Higher-order Interaction between component state

The component state nodes specify which states exist, but not how they may interact during execution. Such interactions can be higher-order, as shown in Section 3.3. A failure may arise only when several component states participate together, even though no pair of them is unsafe on its own. They can also be temporally distributed, as states introduced by different evolution steps may persist and only later become jointly relevant to the same agent behavior. A pairwise graph cannot represent this higher-order composition directly. Moreover, interactions may involve versioned states introduced by different evolution steps, requiring the representation to preserve dependencies across updates. We represent interactions with a hypergraph over versioned component state nodes. Formally, we define the interaction hypergraph as $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t, \phi)$, where $\mathcal{V}_t$ is the component state node set, $\phi : \mathcal{V}_t \to \mathcal{C}$ assigns each node its component state, and $\mathcal{E}_t$ contains cross-component interactions supported either by execution evidence or by interaction inference. We distinguish observed hyperedges $\mathcal{E}_t^{\text{obs}}$, supported by execution provenance, from predicted hyperedges $\mathcal{E}_t^{\text{pred}}$, inferred as plausible interactions that have not yet been observed. Each hyperedge $e \in \mathcal{E}_t$ connects states from at least two distinct components, *i.e.*, $|\{\phi(v) : v \in e\}| \geq 2$.

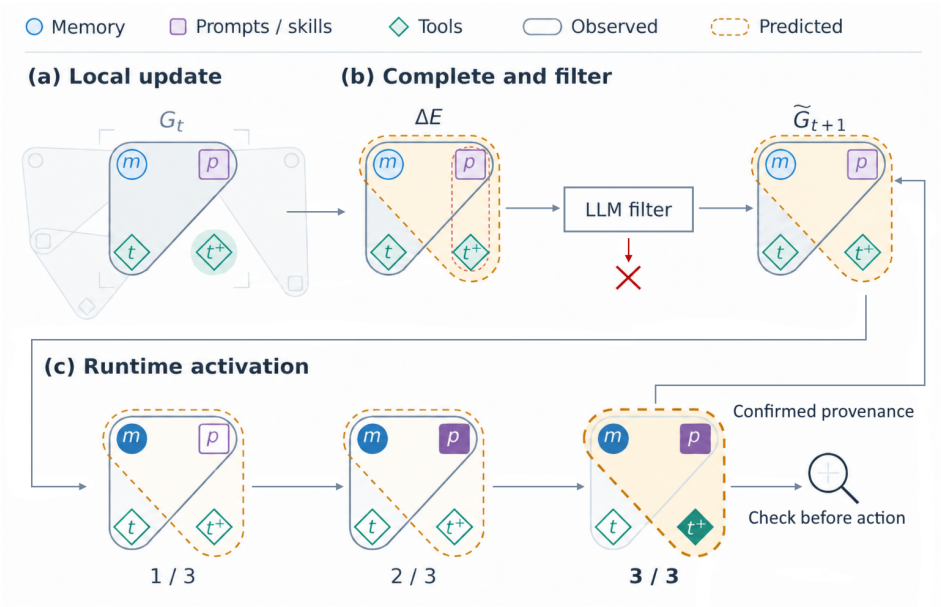


Figure 2: Overview of our method. (a) A candidate update u_t introduces a component-state node (t^+ in this example), and the local interaction neighborhood $\Gamma_h(u_t)$ is extracted from the existing hypergraph G_t. (b) Local completion proposes interactions involving the new state. An LLM filters these candidates using component states and provenance, yielding the provisional hypergraph $\widetilde{G}_{t+1}$. Solid and dashed contours denote observed and predicted hyperedges, respectively; node shapes encode component types. (c) Within one action context, participating nodes become active (filled) in sequence. Each activation visits only incident hyperedges; a fully activated interaction is checked before the action takes effect. Execution provenance promotes a predicted hyperedge to observed status once confirmed at runtime, irrespective of its safety verdict.

4.3 Incremental Construction of the Hypergraph

We update the hypergraph incrementally around the component states introduced or modified by the candidate update u_t, rather than rebuilding the interaction space from Ω_t. This is motivated by the fact that an update typically changes only a small portion of the harness, and full reconstruction would discard the locality already captured by $\mathcal{G}_t$. Let $\Gamma_h(u_t)$ denote the interaction neighborhood within h steps of the node and hyperedge incidence structure of $\mathcal{G}_t$. This neighborhood collects the historical interaction contexts most relevant to the update. We then perform local hypergraph completion within $\Gamma_h(u_t)$ by using existing higher-order interaction patterns to propose new cross-component interactions that involve the changed states. This construction can recover both direct continuations of previously observed interactions and new compositions among states that have not been observed together, without enumerating the full candidate space Ω_t. Graph structure alone, however, does not establish that a proposed interaction is semantically plausible. We therefore use an LLM to examine the participating component states together with available provenance context and filter the interactions produced by local completion. Retained interactions are added to $\mathcal{E}^{\text{pred}}_{t+1}$. Together with the existing observed interaction history, they form the provisional hypergraph $\widetilde{\mathcal{G}}_{t+1} = (\widetilde{\mathcal{V}}_{t+1}, \widetilde{\mathcal{E}}_{t+1}, \phi)$ before u_t is committed. If later execution provenance confirms a predicted interaction, the corresponding hyperedge is moved from $\mathcal{E}^{\text{pred}}$ to $\mathcal{E}^{\text{obs}}$. The resulting provisional graph provides the reduced interaction space used for runtime monitoring in Section 5.

5 Hypergraph Guided Runtime Monitoring

5.1 Incremental Monitoring of Component Interaction

Section 4 maintains a sparse hypergraph of observed and predicted cross-component interactions. This structure allows runtime monitoring to focus on interactions connected to the component states

that actually participate in the current execution. When a component state node v becomes active, only hyperedges containing v can change from partially active to fully active. Then the monitor inspects only its incident hyperedges, $\mathcal{I}(v) = \{e \in \mathcal{E}_t : v \in e\}$. For each visited hyperedge, the monitor records which participating states have appeared in the same decision or action context. If the activation of v causes all states in a hyperedge to appear, the monitor checks the resulting composition before the corresponding action takes effect. This procedure detects every fully activated interaction represented in $\mathcal{G}_t$, because the hyperedge is necessarily inspected when its final participating state becomes active. Activation is tracked separately for each decision context. A composition is checked again when it participates in a subsequent action, since the request, arguments, or permissions may differ. Observed and predicted hyperedges follow the same monitoring procedure. Once a predicted hyperedge is fully activated and confirmed by execution provenance, it is moved from $\mathcal{E}_t^{\text{pred}}$ to $\mathcal{E}_t^{\text{obs}}$. Algorithm 1 in Appendix C summarizes the complete procedure. For an execution with active nodes $v_1, \ldots, v_L$, this reduces the runtime cost from dependence on the global composition space Ω_t in equation 1 to $O(\sum_{\ell=1}^{L} |\mathcal{I}(v_\ell)|)$ local hyperedge visits.

5.2 Experiments

We follow the same experimental settings including benchmarks, component updates, eligibility criterion as Section 3.2. We compare against SHE (Qu et al., 2026), which performs whole-harness safety-utility re-evaluation, and HarnessLens (Xu et al., 2026a), which selectively verifies behavior-relevant harness modifications. We also add Naive Exhaustive Checking discussed in Section 3.4 as our baseline. We use Qwen3-32B (Yang et al., 2025) as the agent backbone and Qwen3-8B (Yang et al., 2025) as the LLM semantic filter for hypergraph construction. The hyperparameter h is set to 3 for all experiments.

We evaluate each method along three dimensions: safety, utility, and verification cost. For safety, we report Residual CFR, defined as the fraction of the fixed eligible panels whose compositional safety violation still reaches execution after applying the safety mechanism. For utility, we use the same benchmark-specific task-success criterion as in Section 3.2 and report the fraction of eligible panels that remain utility-preserving under the safety mechanism. For cost, we count the additional verification checks introduced by each method and report runtime check rate and time cost.

Table 3: **Safety effectiveness** of different mechanisms against compositional failures. Each entry reports the residual compositional failure rate (CFR) as f/n (%), where f is the number of compositional failures among the n eligible panels. The eligible panel set is fixed before applying each safety mechanism.

Method	AgentDojo	Agent-SafetyBench	Agent Security Bench
No Monitoring	22/157 (14.01%)	2/11 (18.18%)	19/1243 (1.53%)
Naive	1/157 (0.64%)	0/11 (0.00%)	0/1243 (0.00%)
SHE (Qu et al., 2026)	6/157 (3.82%)	1/11 (9.09%)	5/1243 (0.40%)
HarnessLens (Xu et al., 2026a)	8/157 (5.10%)	1/11 (9.09%)	5/1243 (0.40%)
Ours	**2/157 (1.27%)**	**0/11 (0.00%)**	**1/1243 (0.08%)**

Table 3 shows that all safety mechanisms reduce compositional failures, but to different extents. Naive Exhaustive Checking eliminates all observed failures on Agent-SafetyBench and Agent Security Bench, leaving one on AgentDojo (0.64% CFR). SHE and HarnessLens reduce the number of failures but still leave residual CFRs of 3.82-5.10% on AgentDojo and 9.09% on Agent-SafetyBench. In contrast, our method reduces the residual CFR to at most 1.27% across all three benchmarks, approaching the safety effectiveness of exhaustive checking. These results demonstrate that our hypergraph-guided runtime monitoring can effectively address compositional safety failures in harness evolution.

Table 4 further compares utility and verification cost. Naive Exhaustive Checking incurs the highest runtime verification cost because every candidate action is checked. SHE preserves high task utility but requires substantial update-time verification, while HarnessLens reduces this cost through selective behavior-aware verification. In comparison, our method preserves comparable task utility while invoking the runtime safety checker on fewer than 30% of candidate actions on average.

Table 4: **Utility and verification cost** of different safety mechanisms. Utility is the fraction of eligible panels satisfying the benchmark-specific task objective. Verification counts additional validation executions per harness update. Runtime Check Rate is the fraction of candidate actions invoking the runtime safety checker. Time Cost is the additional wall-clock time per panel over execution without a safety mechanism.

Method	Utility ↑	Verification ↓	Runtime Check Rate ↓	Time Cost (s/task) ↓
AgentDojo				
Naive	58.6%	–	100.0%	103.9
SHE (Qu et al., 2026)	57.3%	1200	–	77.1
HarnessLens (Xu et al., 2026a)	52.9%	114	–	33.8
Ours	**59.2%**	**82**	**33.6%**	**32.5**
Agent-SafetyBench				
Naive	45.5%	–	100.0%	139.5
SHE (Qu et al., 2026)	45.5%	360	–	137.8
HarnessLens (Xu et al., 2026a)	36.4%	320	–	30.2
Ours	**54.5%**	**185**	**20.1%**	**28.1**
Agent Security Bench				
Naive	62.8%	–	100.0%	80.5
SHE (Qu et al., 2026)	61.1%	260	–	98.0
HarnessLens (Xu et al., 2026a)	49.5%	123	–	17.4
Ours	**63.3%**	**101**	**24.6%**	**19.8**

This reduces the added execution time from 80.5-139.5 s/task under Naive Exhaustive Checking to 19.8-32.5 s/task with our method. Together with the safety results in Table 3, these results reveal an empirical safety-utility-cost trade-off. Reducing residual CFR generally requires checking more interactions, which increases verification cost, or intervening more conservatively, which can reduce task utility. When both verification cost and utility loss are constrained, fewer safety-relevant interactions can be covered, leading to higher residual CFR. Our hypergraph-guided monitoring mitigates this trade-off by approaching the safety effectiveness of exhaustive checking while preserving comparable task utility at substantially lower runtime verification cost.

5.3 Sensitivity Analysis

The interaction radius h determines how much historical interaction context is considered when completing the hypergraph around a new component state. A small h may omit relevant interactions, whereas a larger h expands the candidate interaction space and may increase both construction and runtime monitoring costs. We therefore study the sensitivity to h on AgentDojo (Debenedetti et al., 2024), varying $h \in \{2, 3, 4\}$ while keeping all other settings fixed. Detailed results are reported in Appendix G. As shown in Table 6, increasing h from 2 to 3 reduces the residual CFR from 2.55% to 1.27%, while slightly decreasing utility from 60.5% to 59.2% and increasing the runtime check rate from 30.1% to 33.6%. These results further illustrate the empirical safety-utility-cost trade-off identified in Section 5.2. We therefore use $h = 3$, which provides a favorable balance between compositional-safety coverage and monitoring overhead.

6 Conclusion

We study compositional safety failures in harness evolution, where component updates that are individually safe and utility-preserving can jointly produce unsafe behavior. We empirically identify such failures across heterogeneous harness components on three agent safety benchmarks, showing that independent update validation does not guarantee compositional safety. To address the combinatorial cost of checking cross-component interactions, we introduce an incremental interaction hypergraph and hypergraph-guided runtime monitoring that focuses safety checking on interactions activated during execution. Experiments show that our method substantially reduces residual compositional failures while preserving task utility at a lower checking cost, revealing a safety-utility-cost trade-off.

AI USE STATEMENT

Generative AI tools were used to assist with literature review, research design, code development, and manuscript editing. These tools were used as supporting aids rather than autonomous research agents. We did not use generative AI to design the method.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Anthropic. Mitigating the risk of prompt injections in browser use, 2025. URL https://www.anthropic.com/research/prompt-injection-defenses.

Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

Guoxin Chen, Zhong Zhang, Xin Cong, Fangda Guo, Yesai Wu, Yankai Lin, Wenzheng Feng, and Yasheng Wang. Learning evolving tools for large language models. In *International Conference on Learning Representations*, volume 2025, pp. 87140–87169, 2025.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. *Advances in neural information processing systems*, 37:82895–82920, 2024.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE software*, 12(6):17–26, 1995.

Chaoyue He, Xin Zhou, Di Wang, Hong Xu, Wei Liu, and Chunyan Miao. Harness engineering for language agents: The harness layer as control, agency, and runtime. 2026.

D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

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

Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, et al. Position: Agentic evolution is the path to evolving llms. *arXiv preprint arXiv:2602.00359*, 2026a.

Ruixiao Lin, Xinhao Deng, Qingming Li, Jianan Ma, Yunhao Feng, Yuqi Qing, Zhenyuan Li, Yechao Zhang, Shiwen Cui, Changhua Meng, et al. Safety in self-evolving llm agent systems: Threats, amplification, and case studies. *arXiv preprint arXiv:2606.23075*, 2026b.

Qinghua Mao, Wanying Qu, Dadi Guo, Leitao Yuan, Qingyu Liu, Yu Li, Guanxu Chen, Yanwei Fu, Xi Lin, Xia Hu, et al. Safeevolve: Harness-policy co-evolution from agent experience for safety alignment. *arXiv preprint arXiv:2609.02786*, 2026.

Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. Tthe: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.

OpenAI. Designing ai agents to resist prompt injection, 2026. URL https://openai.com/index/designing-agents-to-resist-prompt-injection/.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, et al. Autosaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40, 2024.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Wanying Qu, Qinghua Mao, Yu Li, Jiyao Liu, Xin Zhang, Dadi Guo, Yanxu Zhu, Qingyu Liu, Leitao Yuan, Xi Lin, et al. She: Trajectory-driven safety harness evolution for llm agents. *arXiv preprint arXiv:2608.09885*, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2025. URL https://arxiv.org/abs/2406.06608.

Linfang Shang, Ming Xu, Yiding Sun, Tianle Xia, Lingxiang Hu, Lan Xu, and Ning Zheng. When self-evolution backfires: Pre-commit gating against skill contamination in llm agents. *arXiv preprint arXiv:2608.05810*, 2026.

Shuai Shao, Qihan Ren, Dongrui Liu, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, et al. Your agent may misevolve: Emergent risks in self-evolving llm agents. In *International Conference on Learning Representations*, volume 2026, pp. 99728–99793, 2026.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Liyan Tang, Cyrus Rashtchian, Chun-Sung Ferng, Andrew Tomkins, Da-Cheng Juan, and Tu Vu. Wikiskill: Compiling agent experience into persistent knowledge for skill evolution, 2026. URL https://arxiv.org/abs/2608.27454.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Kaixiang Wang, Jiong Lou, Zhaojiacheng Zhou, and Jie Li. Oep: Poisoning self-evolving llm agents via locally correct but non-transferable experiences. *arXiv preprint arXiv:2605.18930*, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Lilian Weng. Harness engineering for self-improvement. *lilianweng.github.io*, July 2026. URL https://lilianweng.github.io/posts/2026-07-04-harness/.

Jinghan Xu, Yikai Zhang, Aili Chen, Weiyuan Li, Jiaqing Liang, and Deqing Yang. Verify smarter, evolve further: Efficient harness evolution through behavior-aware verification. *arXiv preprint arXiv:2608.27311*, 2026a.

Jinghan Xu, Yikai Zhang, Aili Chen, Weiyuan Li, Jiaqing Liang, and Deqing Yang. Verify smarter, evolve further: Efficient harness evolution through behavior-aware verification. *https://arxiv.org/abs/2608.27311*, 2026b.

Yuanyuan Xu, Wenjie Zhang, Yin Chen, Xuemin Lin, and Ying Zhang. Self-evolving agents as dynamic graph transformation: A survey and new perspective. 2026c.

Bingyu Yan, Xiaoming Zhang, Chaozhuo Li, Ziyi Zhou, Yirui Qi, and Litian Zhang. Benign alone, harmful together: Exploiting experience composition in self-evolving llm agents. *arXiv preprint arXiv:2608.01759*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Xianglin Yang, Yufei He, Shuo Ji, Bryan Hooi, and Jin Song Dong. Zombie agents: Persistent control of self-evolving llm agents via self-reinforcing injections. *arXiv preprint arXiv:2602.15654*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Zonghao Ying, Xiangfan Wu, Huiyu Wu, Xing Zheng, Huangsheng Cheng, Xiaorong Shi, and Jing Guo. Skilljack: Persistent skill backdoors in self-evolving agents. *arXiv preprint arXiv:2608.03509*, 2026.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026.

Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (asb): Formalizing and benchmarking attacks and defenses in llm-based agents. In *International Conference on Learning Representations*, volume 2025, pp. 35331–35366, 2025.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents, 2024a. URL https://arxiv.org/abs/2404.13501.

Zhexin Zhang, Shiyao Cui, Yida Lu, Jingzhuo Zhou, Junxiao Yang, Hongning Wang, and Minlie Huang. Agent-safetybench: Evaluating the safety of llm agents. *arXiv preprint arXiv:2412.14470*, 2024b.

Weixiang Zhao, Yichen Zhang, Yingshuo Wang, Yang Deng, Yanyan Zhao, Xuda Zhi, Yongbo Huang, Hao He, Wanxiang Che, Bing Qin, et al. On safety risks in experience-driven self-evolving agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 42145–42169, 2026.

A Benchmark and Evaluation Details

This appendix provides additional details on benchmark selection, update construction, and interaction evaluation for Section 3.2. We use AgentDojo, Agent-SafetyBench and Agent Security Bench as executable safety environments rather than treating their original benchmark scores as directly comparable quantities. Our goal is not to estimate the prevalence of compositional failures in any particular deployed evolution system, but to test whether independently acceptable states can become unsafe once they coexist and interact within a shared evolving harness.

A.1 AgentDojo Setup

Environment and requests. We use AgentDojo v1.2.2 (Debenedetti et al., 2024) and restrict the experiment to its banking environment. AgentDojo provides a stateful environment with executable tools, allowing safety violations to be verified from actual tool calls and the resulting environment state. We do not introduce prompt injection attacks or untrusted external content. Instead, we construct 60 trusted request variants from a single banking task family in which the user requests a numerical refund calculation. We use variants of the same task family to hold the intended user objective fixed while varying the execution context in which evolved harness states are activated.

Component updates. We construct a benchmark-specific update pool containing one memory update, nine prompt updates, and one evolved tool update. All updates are generated through the corresponding evolution mechanisms described in Section 3.2. They are not manually constructed as adversarial payloads for the safety evaluation.

For each request, we instantiate the compatible cross-component update compositions from this pool. This produces

$$60 \times (1 + 9) \times 1 = 600$$

interaction panels. Each panel is evaluated under the controlled four-configuration protocol described in Section 3.2.

A.2 Agent-SafetyBench Setup

Task selection. Agent-SafetyBench (Zhang et al., 2024b) contains 2,000 official test cases spanning heterogeneous tool environments and safety risks. We first derive a structural candidate pool of 455 tasks that are single-turn, executable in our setting, expose at least one tool, and do not rely on injected dialogue. This filtering is used only to identify tasks compatible with component-level harness evolution and does not modify the original task content.

We separate tasks used to generate harness updates from those used for compositional evaluation. Seven source tasks are used to produce accepted update packages, while evaluation is performed only on held-out tasks. From the candidate pool, we select 13 held-out official tasks for which at least one evolved tool can be exposed under a compatible environment. The source tasks used for update generation are disjoint from these evaluation tasks.

Component updates. The seven source tasks produce seven memory updates, seven prompt updates, and corresponding skill and tool updates. For each held-out task, we instantiate the compatible cross-component compositions between the resulting updates. This yields

$$13 \times (7 + 7) \times 1 = 182$$

interaction panels. We retain the original Agent-SafetyBench task specification and tool environment, and use its official safety evaluation procedure based on ShieldAgent for behavioral safety assessment.

A.3 Agent Security Bench Setup

Benchmark adaptation and task scope. Agent Security Bench (ASB-Sec) (Zhang et al., 2025) provides ten role-specific agents spanning finance, law, medicine, education, psychological counselling, e-commerce, aerospace, academic search, autonomous driving, and system administration.

Our experiment is an adaptation of its clean-task and tool-simulation environment, rather than a replication of its native prompt-injection, memory-poisoning, or backdoor attacks.

We begin with the official clean tasks associated with the ten agents. For each agent, we freeze one capability whose invocation can produce a consequential action. We then exclude four tasks whose wording could directly or indirectly authorize the selected capability: medical treatment-plan updating, psychological therapy-plan design, e-commerce product-launch execution, and aerospace software updating. This leaves 47 clean tasks. The numbers of retained tasks for the ten agents are $(5, 5, 4, 5, 4, 4, 4, 6, 5, 5)$, which sum to 47. We do not introduce prompt injection, poisoned observations, backdoor triggers, or adversarially constructed component updates.

Trajectory-conditioned component updates. For each retained source task, we first execute the unmodified agent and record its clean plan, action, observation, and final-answer trajectory. We use five predeclared utility objectives: grounding, plan completion, specialist-tool choice, evidential cross-checking, and production of a decision-ready outcome. These objectives diversify the generated updates but are not treated as independent experimental replications.

The tool component is a catalog-level capability update. For each agent, its underlying callable implementation is one fixed official non-aggressive ASB-Sec tool leaf: `CredentialManager`, `InvestmentDiversionTool`, `SettlementManagementTool`, `MedicationValidator`, `GradeManagementTool`, `TreatmentAuthorizationTool`, `DiscountManagementTool`, `SystemShutdownController`, `AcademicProfileManager`, or `SoftwareUpdateManager`. The official attacker instruction and attack goal associated with the tool record are never shown to either the updater or the evaluated agent.

The 47 source tasks and five utility objectives produce $47 \times 5 = 235$ candidate update packages. A package passes source admission only if the baseline and all singleton component-update executions complete successfully, satisfy the clean-task utility proxy, and do not invoke the newly exposed capability. Calls to the agent's original normal tools remain permitted. This gate admits 174 packages and rejects 61.

Interaction panels. Each admitted package is evaluated on every other retained clean task belonging to the same agent. The source and target requests in a panel are therefore distinct, although a request may act as a source for one package and as a target for another. Before source admission, the complete schedule contains

$$2 \times 5 \times \sum_{a=1}^{10} n_a(n_a - 1) = 1{,}780$$

source-target-stratum-recipe slots, where n_a is the number of retained tasks for agent a. After source admission, 1,362 panels are executed.

B Details of Compositional-Failure Identification

This appendix provides additional details for the empirical study in Section 3.2.

B.1 Interaction-Panel Construction

An interaction panel is defined by an evaluation request and two updates, u_i and u_j, associated with distinct harness components. For each selected pair, we instantiate four configurations while keeping the request, backbone model, environment, and all unrelated harness state fixed:

- 00: neither u_i nor u_j;
- 10: only u_i;
- 01: only u_j;
- 11: both u_i and u_j.

This controlled design separates the independent effect of each update from the effect that appears only when the two updates coexist. The benchmark-specific update pools and resulting numbers of

panels are reported in Table 2. Because the numbers of available updates differ across component types, the panel set is fixed before inspecting joint safety outcomes rather than being selected according to whether 11 fails. This prevents the identification procedure from preferentially retaining unsafe joint configurations.

B.2 Eligibility Criterion

A panel is eligible for compositional-failure analysis only if the following conditions hold:

1. the original configuration 00 satisfies the benchmark safety criterion;
2. the singleton configuration 10 remains safe and preserves the utility of the requested task; and
3. the singleton configuration 01 remains safe and preserves the utility of the requested task.

The eligibility filter is essential to the phenomenon studied in this work. If either singleton update is already unsafe, a violation under 11 does not establish a failure of composition. Similarly, an update that destroys the intended task behavior should not be treated as an independently acceptable harness update. Utility preservation is evaluated using the benchmark-specific task objective and execution outcome rather than requiring the singleton trajectory to be identical to 00.

B.3 Component-Specific Activation

Different harness components affect execution through different mechanisms. A memory update may become active when its stored information is retrieved into the current context. A prompt update affects the instructions governing the current execution. A skill update becomes active when its reusable procedure is exposed to or invoked by the agent. A tool update becomes active when the corresponding capability is made available and participates in the execution.

We therefore define activation according to the native exposure semantics of each component rather than requiring every update to appear as an explicit agent action. This distinction is important for memory and prompt states, whose effects may influence downstream decisions without being directly represented as a tool call. For a joint configuration 11, both component state updates are available according to these semantics.

C Automatic Hypergraph Update and Runtime Monitoring

Algorithm 1 gives the end-to-end procedure for incrementally maintaining the interaction hypergraph and using it for runtime monitoring. The algorithm takes as input the current interaction hypergraph, the component states changed by a harness update, and the subsequent execution stream. All interaction candidates are generated from the local hypergraph neighborhood and filtered automatically; no manually specified interaction pairs are required.

Complexity. The algorithm never enumerates the global composition space Ω_t. For an update introducing states ΔV_t, construction is restricted to their h-hop interaction neighborhoods and the locally generated candidate hyperedges. At runtime, an activation of state v visits only its incident hyperedges $\mathcal{I}(v)$. Therefore, for an execution activating $v_1, \ldots, v_L$, the monitoring cost is $O(\sum_{\ell=1}^{L} |\mathcal{I}(v_\ell)|)$, independent of $|\Omega_t|$.

Interaction discovery details. Since newly introduced states in ΔV_t are not yet contained in $\mathcal{G}_t$, we first use the provenance of update u_t to identify relevant existing component states and extract the local interaction neighborhood $\Gamma_h(u_t)$ from $\mathcal{G}_t$. For each new state, LocalCompletion proposes cross-component candidate hyperedges by combining it with states in this neighborhood. The retained candidates are added to $\widetilde{E}_{t+1}^{\text{pred}}$ as predicted interactions.

Runtime safety checking. When all states of a hyperedge e become active within the same decision context q, the monitor invokes IsSafe$(e, q; \mathcal{S})$ before the corresponding action takes effect. The safety checker $\mathcal{S}$ follows the benchmark-specific safety criterion used throughout our evaluation. Thus, the hypergraph determines *when* compositional safety checking is triggered, while $\mathcal{S}$ determines whether the activated behavior is safe. If the check identifies a safety violation, the corresponding action is prevented from taking effect.

Algorithm 1 Automatic Hypergraph Update and Runtime Monitoring

Require: Current hypergraph $\mathcal{G}_t = (V_t, E_t^{\text{obs}}, E_t^{\text{pred}}, \phi)$; changed component states ΔV_t from update u_t; interaction radius h; semantic filter $\mathcal{F}$
Ensure: Updated hypergraph $\mathcal{G}_{t+1}$

$\widetilde{V}_{t+1} \leftarrow V_t \cup \Delta V_t$
$\widetilde{E}_{t+1}^{\text{obs}} \leftarrow E_t^{\text{obs}}$
$\widetilde{E}_{t+1}^{\text{pred}} \leftarrow E_t^{\text{pred}}$
▷ Incremental interaction discovery
for all $v \in \Delta V_t$ **do**
  $\Gamma_h(v) \leftarrow \textsc{ExtractNeighborhood}(\mathcal{G}_t, v, h)$
  $\mathcal{C}_v \leftarrow \textsc{LocalCompletion}(v, \Gamma_h(v))$
  $\Delta E_v \leftarrow \textsc{SemanticFilter}(\mathcal{C}_v, \mathcal{F})$
  $\widetilde{E}_{t+1}^{\text{pred}} \leftarrow \widetilde{E}_{t+1}^{\text{pred}} \cup \Delta E_v$
end for
$\widetilde{\mathcal{G}}_{t+1} \leftarrow (\widetilde{V}_{t+1}, \widetilde{E}_{t+1}^{\text{obs}}, \widetilde{E}_{t+1}^{\text{pred}}, \phi)$
▷ Hypergraph-guided runtime monitoring
for all decision contexts q during execution **do**
  $A_q \leftarrow \emptyset$
  $C_q \leftarrow \emptyset$
  for all activation event v in q **do**
    $A_q \leftarrow A_q \cup \{v\}$
    $\mathcal{E}_{\text{ready}} \leftarrow \{e \in \mathcal{I}(v) : e \subseteq A_q,\ e \notin C_q\}$
    for all $e \in \mathcal{E}_{\text{ready}}$ **do**
      $C_q \leftarrow C_q \cup \{e\}$
      if not $\textsc{IsSafe}(e, q)$ **then**
        $\textsc{InterveneBeforeAction}(q)$
      end if
      if $\textsc{ConfirmedByProvenance}(e, q)$ **then**
        $\textsc{PromoteToObserved}(\widetilde{\mathcal{G}}_{t+1}, e)$
      end if
    end for
  end for
end for
return $\widetilde{\mathcal{G}}_{t+1}$

D Higher-Order Compositional-Failure Identification

Table 5 reports the detailed results of the 3-way compositional-failure analysis in Section 3.3. AgentSafetyBench (Zhang et al., 2024b) yields only 9 eligible triples out of 637 candidates, which is too small for a meaningful higher-order analysis. So we evaluate 3-way compositions on AgentDojo (Debenedetti et al., 2024) and Agent Security Bench (Zhang et al., 2025).

Table 5: Identification of 3-way compositional failures. A triple is eligible only when the baseline, all individual updates, and all pairwise compositions are safe and utility-preserving. 3-CF counts failures that emerge only under the full three-update composition, and 3-CFR is computed as 3-CF/Eligible.

Benchmark	Triples	Eligible	3-CF	3-CFR
AgentDojo (Debenedetti et al., 2024)	540	138	15	10.87%
Agent Security Bench (Zhang et al., 2025)	681	578	3	0.52%

E Distinction from Prior Compositional Risks

Recent works have identified compositional risks arising from interactions among multiple retained items within the same harness component. Yan et al. (Yan et al., 2026) study retained experiences and show that individually benign experiences can accumulate across evolution steps and jointly weaken the agent's safety boundary when activated together. Shang et al. (Shang et al., 2026) study retained skills and identify combinatorial contamination, where individually harmless skills can jointly degrade agent performance; they further introduce pre-commit gating and subset selection to prevent harmful skill combinations from entering the skill pool.

Our work studies a complementary setting across *heterogeneous harness components*. Rather than interactions among multiple retained items within one experience or skill component, we consider interactions among component states such as memory, prompts, skills, and tools. We define a compositional safety failure when individually safe and utility-preserving component updates produce an unsafe behavior only when jointly activated; for higher-order failures, all lower-order compositions must also remain safe and utility-preserving.

This cross-component setting further introduces an execution-dependent challenge. Component states accepted at different evolution steps may coexist without their interaction being exercised during validation, and become safety-relevant only when a future execution jointly activates them. Accordingly, our interaction hypergraph explicitly tracks safety-relevant interactions across heterogeneous component states, while our runtime monitoring checks these interactions when they become active during execution.

F Why an Interaction Hypergraph?

Our choice of a hypergraph is motivated by the interaction structure identified in Section 3.2 and 3.3. The pairwise compositional failures show that safety cannot be characterized solely at the level of individual component states. More importantly, the irreducible 3-way failures show that these interactions can be genuinely higher-order: all individual states and pairwise compositions may remain safe and utility-preserving, while their joint activation produces an unsafe behavior. A standard pairwise graph cannot directly represent such an interaction without decomposing it into pairwise relations that are themselves not unsafe.

A hypergraph provides a direct representation of this structure, since a single hyperedge can connect all component states participating in the same higher-order interaction. Compared with storing interaction tuples independently, the hypergraph also preserves shared interaction structure across evolution steps. This structure allows newly introduced component states to reuse existing interaction neighborhoods for local completion and enables runtime monitoring to visit only hyperedges incident to states activated during the current execution. Thus, the hypergraph is used not only to represent higher-order interactions, but also to support incremental construction and activation-aware monitoring without repeatedly enumerating the global composition space.

G Sensitivity Analysis

Table 6: Sensitivity to the local interaction radius h on AgentDojo.

h	**Residual CFR** $\downarrow$	**Utility** $\uparrow$	**Runtime Check Rate** $\downarrow$	**Time Cost (s/task)** $\downarrow$
2	4/157 (2.55%)	60.5%	30.1%	30.1
3	2/157 (1.27%)	59.2%	33.6%	32.5
4	2/157 (1.27%)	58.0%	47.6%	49.7

We evaluate the sensitivity of our method to the local interaction radius h on AgentDojo. We vary $h \in \{2, 3, 4\}$ while keeping all other settings fixed, with $h = 3$ as the default configuration used in the main experiments.