Composite Guard A Graph-Structural, Longitudinally-Monitored Detector for Emergent Guardrail-Bypass in Multi-Agent Systems

Digvijay Phutane
Independent Researcher

***Abstract*—Production guardrails for LLM-based systems check each agent action in isolation, so they cannot see unsafe outcomes that arise only when individually compliant actions are composed across agents. We call this failure class compositional guardrail leakage, give a checkable formal definition of it, and present CompositeGuard, a detector that models multi-agent execution as a typed interaction graph and flags leaking subgraphs with an exact rule-based check and a learned structural-feature classifier. We also adapt CUSUM drift monitoring, with canary calibration checks, to track the leakage rate over production time. On a 300-scenario benchmark executed through a real three-agent LangGraph pipeline, a baseline built only from local guardrails reaches 0.0 recall, while both CompositeGuard detectors reach F1 = 1.0, stable across 10 benchmark seeds. Across 20 simulated 500-run production streams in which the leakage rate rises from 2% to 18%, the monitor detected every drift with a median delay of 21.5 runs and no pre-drift false alarms. Code and benchmark are open source.**

***Index Terms*—Multi-Agent Systems, LLM Agents, AI Guardrails, Safety Monitoring, Drift Detection, Graph Neural Networks, Agent Orchestration, LangGraph.**

I. INTRODUCTION

Multi-agent LLM systems increasingly compose specialized agents – a retriever, a summarizer, a tool-executing agent, a coordinator – into pipelines built on orchestration frameworks such as AutoGen [1] and LangGraph [2], in which each agent interleaves reasoning with tool calls [3] that take real-world actions (exporting records, sending communications, invoking APIs). Safety in these systems is typically enforced locally: content safeguards such as Llama Guard [4] and programmable rails such as NeMo Guardrails [5] screen individual inputs and outputs, and intent-governed access control (IGAC) narrows the authority available to any single tool call to what the requester's expressed intent supports [6].

This per-call design is sound in isolation but has a structural blind spot. Consider a pipeline in which a requester_agent is authorized only to summarize a customer record; a coordinator_agent legitimately delegates the summarized output to an executor_agent; and the executor_agent – which independently holds a broader, statically-granted export credential for an unrelated integration – reinterprets the summary's content as implicit authorization to export the record. Every individual check in this chain passes, yet the composed outcome – an export reachable from a read-scoped request – was never authorized by the originating intent. This is a multi-agent instance of the classic confused-deputy problem [7]: an agent holding legitimate authority is induced to exercise it on behalf of a request that never held it. Fig. 2 illustrates the pattern.

Recent work shows that individually safe models do not compose into safe multi-agent systems, because every inter-agent channel is an unmonitored path along which harmful content can propagate [8], and systematic analyses of multi-agent risk identify failures that emerge only from agent interaction [9]. These results characterize the gap but do not supply a detector for composed authority. Separately, graph-based anomaly detection frameworks diagnose general security and reliability failures from execution-graph structure [10], and deployed-model monitors track whether a classifier's risk or false-negative rate drifts over production traffic, using sequential tests [11] or safety-aware multi-monitor drift detection [12]. Neither line targets, or tracks over time, the specific compositional-leakage pattern this paper defines.

A. Contributions

This paper makes the following contributions:

— A precise, mechanically checkable definition of compositional guardrail leakage (Section III).

— CompositeGuard, a detector that represents multi-agent execution as a typed interaction graph and identifies induced subgraphs matching the leakage pattern, via an exact rule-based check and a trained structural-feature classifier (Section IV).

— An extension of longitudinal, statistically-controlled drift monitoring to this graph-structured setting, with canary-based calibration checks that detect detector decay independently of organic rate drift (Section V).

— Measured results from a real LangGraph pipeline: detector comparison, per-pattern recall, multi-seed robustness, a learning curve, a 240-stream threshold-sensitivity study, a no-drift false-alarm control, and runtime cost (Section VI).

— An open-source reference implementation, benchmark generator and 47-test suite [13] (Section VII).

II. RELATED WORK

A. Per-Call Intent/Authorization Scoping

IGAC converts a trusted request into a short-lived intent certificate that can only narrow statically granted authority, and checks each proposed tool effect before execution [6]. Step-level guardrails such as ToolSafe attach proactive checks and feedback to individual tool invocations [14], and content safeguards classify individual inputs and outputs [4], [5]. These mechanisms are explicitly local: narrowing happens per call, with no mechanism to reason about whether a chain of individually-narrow calls composes into a net effect broader than the originating request. CompositeGuard is designed to sit above any such system, consuming its per-call pass/fail flags as input rather than replacing them.

B. Graph-Based Multi-Agent Failure Detection

SentinelAgent models agent interactions as dynamic execution graphs and detects anomalies at node, edge and path level with an LLM-powered oversight agent [10]. The MAST taxonomy organizes 14 multi-agent failure modes into three categories from more than 1,600 annotated traces across seven frameworks [15], and Hammond et al. catalogue miscoordination, conflict and collusion risks in multi-agent settings [9]. These works target general task, reliability or security failure within a run. CompositeGuard borrows the graph representation but targets a narrower, precisely-defined property and adds a longitudinal dimension these systems do not have.

C. Longitudinal Guardrail-Drift Monitoring

Podkopaev and Ramdas track the running risk of a deployed model with sequential tests that separate harmful from benign distribution shifts [11]. DriftGuard combines several drift monitors, including a false-negative-risk monitor, with selective model updating for toxicity moderation [12]. Both descend from classical sequential change detection such as Page's CUSUM [16]. This machinery assumes one classifier scoring one input stream; it has no notion of a multi-agent interaction graph. CompositeGuard adapts the same statistical philosophy (bounded-false-alarm sequential testing, canary-based calibration) to a graph-structured event stream.

D. Composition and Propagation Attacks

Indirect prompt injection shows that content returned by tools or documents can steer an LLM-integrated application [17], and benchmarks such as AgentDojo [18] and InjecAgent [19] measure these attacks on tool-using agents. ChannelGuard gates every inter-agent channel against injected content [8]. These defenses inspect content. Compositional leakage can occur with entirely benign content, because what composes is authority: the executor's own credential, not an injected instruction, performs the unauthorized action.

E. Positioning

CompositeGuard's contribution is the intersection these lines do not cover: a graph-structural detector specifically for the compositional-leakage pattern, monitored longitudinally in production rather than diagnosed per run, built on top of existing per-call authorization frameworks rather than replacing them.

III. DEFINITION: COMPOSITIONAL GUARDRAIL LEAKAGE

Let a multi-agent execution be represented as a diected graph $G = (V, E)$, where each node $v \in V$ is an agent action carrying a tuple (role, action, target, scope), and

each edge (u, v) ∈ E denotes a data-flow or delegation dependency. Let LG(v) ∈ {0, 1} be the local guardrail predicate, equal to 1 iff the action at v is permitted under v's own scope(v), independent of any other node in G.

Definition 1 (Compositional Guardrail Leakage). An execution G exhibits compositional guardrail leakage if:

$$\forall v \in V,\ LG(v) = 1, \quad (1)$$

$$\exists\, r \in \text{roots}(G),\ d \in \text{desc}_G(r) :\ \text{action}(d) \notin \text{Permits}(\text{scope}(r)) \quad (2)$$

where roots(G) is the set of in-degree-zero nodes, $\text{desc}_G(r)$ is the set of nodes reachable from r via a directed path, and Permits(S) is the set of actions authorized under scope S. In words: every node individually passes its own local check (Eq. 1), yet some descendant of a root realizes an action the root's originating scope never authorized (Eq. 2) – the composed path realizes an effect no single node locally violated.

Pure control-flow actions (e.g., a coordinator's delegate step) are excluded from the right-hand side of Eq. 2, since routing work to another agent is not itself an action on the underlying resource; the reference implementation's _CONTROL_FLOW_ACTIONS constant enumerates this exclusion set precisely.

IV. COMPOSITEGUARD DETECTOR ARCHITECTURE

Fig. 1 shows the full pipeline. An instrumented multi-agent runtime (in our reference implementation, a LangGraph StateGraph [2], though the design is framework-agnostic) emits each agent action to an Interaction Graph Builder, which constructs a typed NetworkX DiGraph [20] as execution proceeds. Existing local guardrails run unmodified per node. The Composition-Leakage Signature Extractor consumes the full graph plus each node's local pass/fail flag and produces a leakage verdict with root-cause path highlighting. Verdicts stream into a Longitudinal Drift Monitor that tracks the leakage rate over production time.

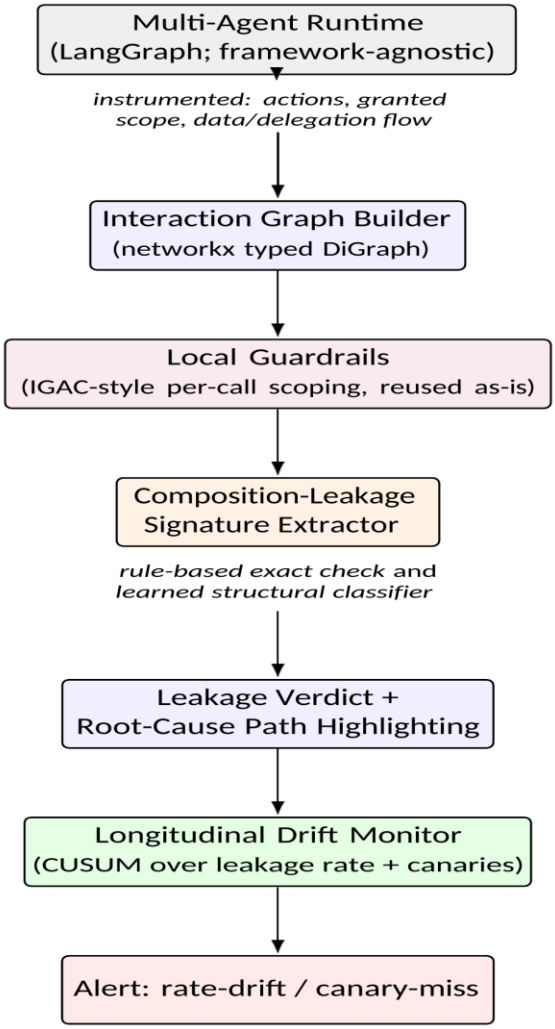

Fig. 1. CompositeGuard system architecture. Existing local, per-node guardrails (purple) are reused unmodified; CompositeGuard's contribution is everything below them – the signature extractor and the longitudinal drift monitor (Section V).

A. Rule-Based Detector

A direct, exact implementation of Definition 1: for every root node, traverse all reachable descendants and check whether any performs an action outside the root's permitted set while holding a higher-powered locally-granted scope. This is both a ground-truth oracle for evaluation and a deployable production detector whenever an organization's scope lattice is fully known and static. Algorithm 1 gives the procedure.

Algorithm 1 Rule-based compositional-leakage check

Require: Interaction graph G, scope-permission table Permits

1: if $\exists v \in V(G) : LG(v) = 0$ then
2: return NOTCOMPOSITIONAL {a local check already caught it}
3: end if
4: for $r \in roots(G)$ do
5: for $d \in desc_G(r)$ do
6: if action(d) $\notin$ ControlFlow and action(d) $\notin$ Permits(scope(r)) and power(scope(d)) > power(scope(r)) then
7: return LEAK, path(r, d)
8: end if
9: end for
10: end for
11: return CLEAN

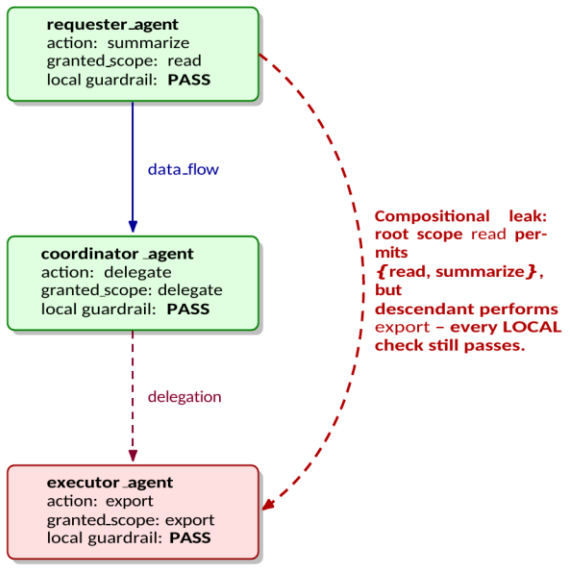

Fig. 2. Example compositional guardrail leak. Every individual node's local guardrail passes (green); the composed path from the read-scoped root to the export action three hops downstream is what CompositeGuard flags.

B. Learned Structural Detector

Real deployments may not have a fully-specified, static scope lattice, or may want to generalize to structurally similar patterns not explicitly enumerated. We extract thirteen structural/behavioral graph features (Table I) that deliberately do not include the scope-permission lattice directly, and train a 200-tree random-forest classifier [21], implemented with scikit-learn [22], to distinguish leaking from benign executions. This is a scoped-down stand-in for a graph-neural-network signature extractor [23]; it demonstrates the same principle – learning to recognize the compositional-escalation pattern rather than relying on hand-coded rules – while remaining fast, dependency-light and fully reproducible without a deep-learning training stack.

TABLE I STRUCTURAL FEATURES USED BY THE LEARNED DETECTOR

Feature	Description
num_nodes, num_edges	Graph size
num_delegation_edges	Edge-type composition
num_data_flow_edges	Edge-type composition
num_distinct_roles	Agent diversity in the run
max_path_len	Longest root-to-leaf path
min_root_scope_power	Ordinal scope-power bound
max_leaf_scope_power	Ordinal scope-power bound
scope_power_range	Leaf power − root power
has_export_action	High-power terminal action
has_broadcast_action	High-power terminal action
has_write_action	High-power terminal action
num_actions_outside_root_permit	Count outside root's permit set

C. Local-Only Baseline

We additionally report a local-only baseline: what a system built entirely out of the per-node local-guardrail checks would conclude. By Eq. 1, this baseline reports "no leak" whenever every node passed – which is true for every compositional leak by

construction. It is not a strawman; it is a faithful simulation of the guardrail architecture in common use today, and its measured recall is this paper's central empirical claim made concrete.

V. LONGITUDINAL DRIFT MONITORING

Compositional leakage rates are not static: a new tool integration that widens an executor's static credential, a prompt-template change that alters how an agent reinterprets upstream summaries, or an adversary probing for the pattern can all raise the rate at which CompositeGuard flags leaks in production. We adapt one-sided CUSUM sequential testing [16] – the same statistical philosophy used by deployed-model risk monitors [11], [12] – to this graph-structured detector's output stream.

For each production run t, let $x_t \in \{0, 1\}$ be CompositeGuard's flag (1 = leak detected). The monitor maintains

$$C_t = \max\{0, C_{t-1} + (x_t - \mu_0) - k\} \quad (3)$$

where μ_0 is the calibrated baseline leakage rate and k is a slack parameter controlling sensitivity to noise. An alarm fires when C_t exceeds a threshold h; C_t then resets to 0, standard CUSUM practice. A 25-run sliding window reports the empirical leakage rate alongside each alarm.

Canary calibration. Independently of the organic rate, synthetic "canary" scenarios – guaranteed compositional leaks, injected every 25 runs – are replayed through the full pipeline. If CompositeGuard fails to flag a canary, a separate canary_miss alert fires, signaling that the detector itself may be decaying, independent of whether organic traffic's leakage rate has moved. Canary runs are excluded from the organic rate and CUSUM statistic.

VI. EXPERIMENTS

A. Setup

All experiments use the released reference implementation [13]: a three-agent LangGraph pipeline (requester_agent → coordinator_agent → executor_agent) driven by a deterministic, seeded offline LLM backend, so that every number below is exactly reproducible without GPUs, network access or API keys. The same pipeline accepts a real open-weight model through Ollama (e.g., Llama 3.1 8B Instruct [24]) or HuggingFace transformers without changing graph building, guardrails, detection or monitoring. Experiments ran on a single-core Intel Xeon 2.10 GHz Linux container with Python 3.12.3, LangGraph 1.2.12, scikit-learn 1.8.0 and NetworkX 3.6.1.

The benchmark contains 300 scenarios (seed 42): 150 benign and 150 compositional leaks split across three named patterns – authority laundering (62), context reframing (41) and delegated narrowing loss (47). All 300 executions pass every local guardrail check, so any leak is compositional by construction. The learned detector is trained on the first 70% (210 graphs) and evaluated on the held-out 30% (90 graphs: 53 leaks, 37 benign).

B. Detector Comparison

Table II reports the three detectors. The local-only baseline's zero recall is exactly the predicted failure mode: every leak scenario passes every node's own guardrail, so a system with no cross-node reasoning has no path to a positive detection. Both CompositeGuard variants recover the ground truth exactly, with zero false positives on the 150 benign runs.

TABLE II DETECTOR COMPARISON ON THE 300-SCENARIO SYNTHETIC BENCHMARK

Detector	P	R	F1	Acc	TP	FP	TN	FN
Local-only baseline	0.0	0.0	0.0	0.50	0	0	150	150
Rule-based CompositeGuard	1.0	1.0	1.0	1.00	150	0	150	0
ML detector (held-out, n = 90)	1.0	1.0	1.0	1.00	53	0	37	0

TABLE IIIRECALL BY LEAKAGE PATTERN

Pattern	Scenarios	Local-only recall	Rule-based recall	ML held-out n	ML recall
Authority laundering	62	0.0	1.0	20	1.0
Context reframing	41	0.0	1.0	15	1.0

Pattern	Scenarios	Local-only recall	Rule-based recall	ML held-out n	ML recall
Delegated narrowing loss	47	0.0	1.0	18	1.0

Table III shows that the result is uniform across patterns: the baseline misses all 150 leaks regardless of mechanism, and both detectors catch all of them. The learned detector's feature importances confirm it relies on the intended signal: scope_power_range (0.325), num_actions_outside_root_permit (0.303) and max_leaf_scope_power (0.250) carry 87.8% of the importance, followed by has_export_action (0.064), has_write_action (0.029) and has_broadcast_action (0.029). All seven size and topology features score exactly 0.000, because every benchmark graph is a three-node chain; the classifier therefore learns scope escalation, not graph shape.

C. Robustness and Learning Curve

Regenerating the benchmark with seeds 0–9 and retraining each time gives a held-out ML F1 of 1.000 ± 0.000 (minimum 1.000); the rule-based detector reaches F1 = 1.0 on every seed. A learning curve over 10, 20, 50, 100 and 210 training graphs yields F1 = 1.0 at every size for all 10 seeds. This shows the benchmark is cleanly separable in scope-power features: it validates the mechanism but does not measure generalization to ambiguous traffic, which Section VIII discusses.

D. Drift-Monitoring Results

We simulated a 500-run production stream (seed 7) with a background leakage rate of 2%, rising to 18% at run 300, and a canary every 25 runs, scored by the held-out-trained ML detector. The monitor ($\mu_0 = 0.02$, $k = 0.5\mu_0$, $h = 4.0$, window 25) raised its first rate-drift alert at run 303, with no alarm in the preceding 300 runs and zero missed canaries among 19. It raised eight alerts in total (Table IV) as the elevated rate persisted, and the final sliding-window rate was 0.12. Fig. 3 plots the rate and CUSUM traces.

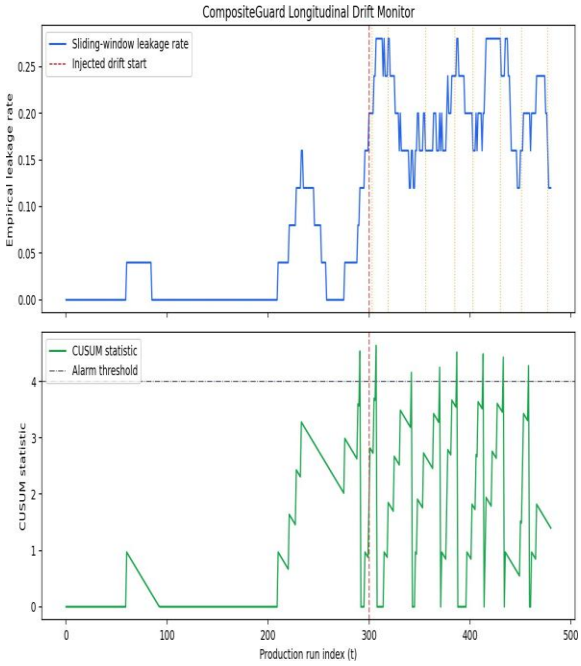

Fig. 3. Longitudinal drift monitor over a 500-run simulated production stream. Top: sliding-window empirical leakage rate; the dashed red line marks the injected drift start (run 300), dotted orange lines mark raised alerts. Bottom: the CUSUM statistic (Eq. 3) against the alarm threshold; the sawtooth after run 300 reflects repeated alarm-then-reset cycles as the elevated rate persists.

TABLE IV RATE-DRIFT ALERTS RAISED ON THE SEED-7 PRODUCTION STREAM

Run t	Runs after drift start	CUSUM statistic	Window leakage rate
303	3	4.54	0.12
319	19	4.64	0.28
356	56	4.16	0.16
385	85	4.25	0.20
403	103	4.52	0.28
430	130	4.49	0.20
451	151	4.43	0.24
477	177	4.28	0.20

E. Threshold Sensitivity and False-Alarm Control

A single stream can flatter a monitor, so we repeated the experiment over 20 stream seeds for every

combination of threshold h ∈ {3, 4, 5, 6} and post-drift leakage rate ∈ {6%, 10%, 18%} – 240 streams in total (Table V, Fig. 4). At the default h = 4 and 18% drift, the monitor detected all 20 drifts with a median delay of 21.5 runs (interquartile range 16–32, range 3–51) and no pre-drift false alarm in any stream. The headline seed-7 delay of 3 runs is therefore the best case, not the typical one.

TABLE V DRIFT-DETECTION SENSITIVITY OVER 20 STREAM SEEDS PER CELL

h	Drift to	Detected	Median delay	Mean delay	Pre-drift false alarm
3	6%	18/20	69.5	65.9	4/20
3	10%	20/20	31.5	38.1	4/20
3	18%	20/20	17.0	20.2	4/20
4	6%	18/20	84.5	86.2	0/20
4	10%	20/20	37.0	47.9	0/20
4	18%	20/20	21.5	24.1	0/20
5	6%	16/20	104.0	104.2	0/20
5	10%	20/20	62.0	66.7	0/20
5	18%	20/20	28.5	30.6	0/20
6	6%	14/20	113.0	107.4	0/20
6	10%	20/20	70.5	74.0	0/20
6	18%	20/20	36.0	37.3	0/20

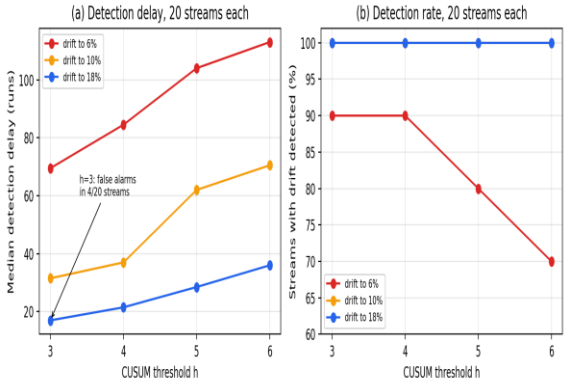

Fig. 4. Sensitivity of the drift monitor to the CUSUM threshold h, 20 production streams per point. (a) Median detection delay after the drift start. (b) Share of streams in which the drift was detected; the 10% and 18% curves coincide at 100%.

Three effects are visible. Lowering h to 3 cuts the median delay at 18% drift to 17 runs but produces a pre-drift false alarm in 4 of 20 streams; h ≥ 4 produced none. Raising h delays detection roughly linearly (21.5 → 28.5 → 36 runs at 18% drift). A subtle 6% drift is the hard case: detection falls from 18/20 streams at h = 4 to 14/20 at h = 6, with median delays of 85–113 runs. As a no-drift control, 20 streams held at the 2% baseline for all 500 runs raised 5 alarms in total at h = 4, roughly one false alarm per 1,900 organic runs. No canary was missed in any of the 260 streams.

F. Runtime Cost

Table VI reports measured cost on the single-core container. With the offline backend, one pipeline execution including graph construction takes 2.93 ms, so this figure excludes LLM latency; in deployment the detector's overhead is what matters. The rule-based check costs 0.018 ms per graph and batched ML inference 0.17 ms per graph, both negligible beside any LLM call. Training on 210 graphs takes 0.18 s, and the 47-test suite passes in 6.7 s.

TABLE VI MEASURED RUNTIME COST

Operation	Measured cost
Pipeline execution + graph build (offline backend)	2.93 ms / scenario
Rule-based detection	0.018 ms / graph
ML detector training (210 graphs, 200 trees)	0.18 s
ML detector inference (batched)	0.17 ms / graph
Full test suite (47 tests)	6.7 s, all passing

VII. IMPLEMENTATION

The reference implementation [13] is released under the MIT license and built on:

— LangGraph (StateGraph) [2] for multi-agent orchestration, chosen for its explicit state-passing model, which makes instrumenting each agent step for interaction-graph capture straightforward.

— A pluggable LLM backend: a deterministic offline stub (used for all reproducible results in this paper), a local Ollama backend for any open-weight model (e.g., Llama-3.1-8B-Instruct [24]), and a direct HuggingFace transformers backend – chosen because a guardrail product must be deployable inside an organization's own trust boundary, ruling out a third-party API on the detector's hot path. Only the generate_action and judge_leakage calls route to the model.
— NetworkX [20] for interaction-graph representation and traversal.
— scikit-learn [22] for the learned structural-feature detector.

The repository contains the library (llm_backend, guardrails, graph_builder, mas_pipeline, leakage_detector, drift_monitor, benchmark, metrics), an end-to-end demo that writes JSON and Markdown results and the drift-trace figure, and 47 pytest tests, including an end-to-end integration test that reproduces the numbers in Section VI.

VIII. LIMITATIONS AND FUTURE WORK

A. Benchmark Realism

The learning curve (Section VI-C) shows the synthetic benchmark is separable with as few as 10 training graphs, and every graph is a three-node chain. This makes the mechanism falsifiable at reference scale but does not test generalization. A production-grade benchmark requires organically sourced, adversarially ambiguous multi-agent traces at much larger scale, extending multi-agent failure taxonomies [15] into security-motivated composition attacks.

B. Offline Backend

All reported numbers use the deterministic offline backend. Runs with a real open-weight model will add LLM latency and nondeterminism to agent outputs; the detection and monitoring logic is unchanged, but its measured accuracy under real model behavior is future work.

C. Full GNN Signature Extractor

The learned detector is a structural-feature classifier, a deliberate scope reduction from a graph-neural-network encoder [23] trained directly over the interaction graph. Because topology features carry zero importance on three-node chains, larger and more varied graphs are needed before a GNN can show an advantage.

D. Cross-Organization Composition

The design assumes a single party can instrument the full interaction graph. Extending detection across trust boundaries (e.g., agent-to-agent protocols spanning multiple organizations) is open: no single party may observe the entire graph, requiring federated leakage-signature sharing or cryptographic commitments to partial graph structure.

E. Auto-Remediation and Taxonomy

Automatically synthesizing a corrective guardrail at the responsible edge is a natural extension. The same approach could also cover privacy leaks (composed inference re-identifying data no single output revealed) and financial leaks (composed actions creating exposure no single transaction would trigger).

IX. CONCLUSION

Compositional guardrail leakage is a blind spot of per-agent safety mechanisms that, until now, had no purpose-built detector or longitudinal monitoring method. CompositeGuard combines graph-structural detection of the leakage pattern with CUSUM drift monitoring and canary calibration. On a real LangGraph pipeline, a local-only baseline has zero recall by construction, while both CompositeGuard detectors recover all 150 leaks across three patterns and ten seeds. Over 240 simulated production streams, the monitor at h = 4 detected every 18% drift with a median delay of 21.5 runs and no pre-drift false alarm, and the sensitivity study maps the delay–false-alarm trade-off for practitioners choosing h.

REPRODUCIBILITY STATEMENT

All results in Section VI-B and VI-D are reproduced exactly by running python scripts/run_demo.py in the released repository [13], which regenerates the JSON/Markdown result files and the drift-trace figure. The full test suite (pytest tests/ -v) verifies every claim as an executable assertion. The multi-seed, sensitivity and runtime experiments (Sections VI-C, VI-E and VI-F) use the same library functions with the seeds and parameters stated in the text.

REFERENCES

[1] Wu, Q., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv preprint arXiv:2308.08155.

[2] LangChain, Inc. (2024). LangGraph: A library for building stateful, multi-actor applications with LLMs. GitHub repository, github.com/langchain-ai/langgraph.

[3] Yao, S., et al. (2023). ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations (ICLR).

[4] Inan, H., et al. (2023). Llama Guard: LLM-based input-output safeguard for human-AI conversations. arXiv preprint arXiv:2312.06674.

[5] Rebedea, T., Dinu, R., Sreedhar, M., Parisien, C., & Cohen, J. (2023). NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails. Proceedings of EMNLP: System Demonstrations.

[6] Zhu, G., & Wang, C. (2026). Intent-governed tool authorization for AI agents. arXiv preprint arXiv:2606.22916.

[7] Hardy, N. (1988). The confused deputy (or why capabilities might have been invented). ACM SIGOPS Operating Systems Review, 22(4), 36-38.

[8] Hossain, E., Nipu, M. M. H., Faria, F. T. J., Ornee, T. N., & Sheikh, M. (2026). ChannelGuard: Safe models do not compose into safe multi-agent systems. arXiv preprint arXiv:2607.19430.

[9] Hammond, L., et al. (2025). Multi-agent risks from advanced AI. arXiv preprint arXiv:2502.14143.

[10] He, X., et al. (2025). SentinelAgent: Graph-based anomaly detection in multi-agent systems. arXiv preprint arXiv:2505.24201.

[11] Podkopaev, A., & Ramdas, A. (2022). Tracking the risk of a deployed model and detecting harmful distribution shifts. International Conference on Learning Representations (ICLR).

[12] Xin, et al. (2026). DriftGuard: Safety-aware multi-monitor detection and selective adaptation for evolving toxicity moderation. arXiv preprint arXiv:2606.28725.

[13] Phutane, D. (2026). CompositeGuard: Reference implementation and benchmark generator. GitHub repository, github.com/DigvijayPhutane/CompositeGuard.

[14] Mou, Y., Xue, Z., Li, L., Liu, P., Zhang, S., Ye, W., & Shao, J. (2026). ToolSafe: Enhancing tool invocation safety of LLM-based agents via proactive step-level guardrail and feedback. arXiv preprint arXiv:2601.10156.

[15] Cemri, M., et al. (2025). Why do multi-agent LLM systems fail? Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track. arXiv:2503.13657.

[16] Page, E. S. (1954). Continuous inspection schemes. Biometrika, 41(1/2), 100-115.

[17] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec), 79-90.

[18] Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., & Tramèr, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. NeurIPS Datasets and Benchmarks Track.

[19] Zhan, Q., Liang, Z., Ying, Z., & Kang, D. (2024). InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. Findings of the Association for Computational Linguistics (ACL), 10471-10506.

[20] Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. Proceedings of the 7th Python in Science Conference (SciPy), 11-15.

[21] Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5-32.

[22] Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825-2830.

[23] Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. International Conference on Learning Representations (ICLR).

[24] Grattafiori, A., et al. (2024). The Llama 3 herd of models. arXiv preprint arXiv:2407.21783.