Locating the Break: Security-Specific Failure Attribution at Agent Hand-offs in Multi-Agent LLM Code Generation Pipelines

Sheldon Menezes

Department of Computer Engineering (MBA Tech), SVKM's NMIMS Mukesh Patel School of Technology Management and Engineering (MPSTME), Mumbai, India

sheldon.menezes@mpstmeacm.com

Abstract: The multi-agent LLM pipeline for code generation deconstructs a coding task and hands off subtasks between specialized agents such as the planner, coder, and reviewer. When this handoff occurs, an issue can be introduced that is carried forward by subsequent steps until it surfaces at the end of the pipeline— but at this stage it is very difficult to determine which agent or reasoning step caused it. Current security tools for LLM-generated code identify vulnerabilities post-generation; they do not pinpoint the handoff where the flaw originated. And while multi-agent failure taxonomies and attribution techniques exist, they are generic and not tailored to the security domain. What is needed is a way to precisely attribute handoff failures in multi-agent pipelines that generate code. To this end we propose HOSA (Hand-Off Security Attribution). HOSA is a structured handoff framework that logs the input, reasoning trace and output of each agent transition, and then applies a specialized attribution technique that links detected vulnerabilities to the exact reasoning step and handoff where they occurred. We contribute: (i) a failure taxonomy specifically designed for security issues arising in multi-agent code generation pipelines; (ii) an attribution method that scores each handoff for likelihood of causing a given security issue based on similarity metrics drawn from prior security research; and (iii) an evaluation protocol complete with metrics for assessing how well HOSA performs relative to existing approaches based on unstructured prompts and partial structured handoffs. A full-scale, multi-model evaluation against reimplemented baselines remains ongoing; in its place we report preliminary results from a small pilot study using a lightweight, checklist-based instantiation of HOSA, alongside a comprehensive description of the framework, a discussion of related work, and expected outcomes grounded in recent failure-attribution research on general-purpose multi-agent workflows.

Keywords: *Multi-agent systems · Large language models · Code generation security · Explainable AI · Failure attribution*

1 Introduction

It is now more common to use large language based multi-agent systems for automating software development tasks that were previously handled either by single models or human developers. Such systems usually have specialized agents working on different parts of a job; for instance, a planner breaking down a task, a coder producing the code, and a reviewer verifying everything before final acceptance. The division of labor enhances task accomplishment in addressing complex issues but creates a failure surface – the hand-off – which is absent in single-agent systems. In every handover, one agent output becomes another agent' s trusted input but with silent propagation of any errors, gaps or misinterpretations introduced therein. It is difficult to detect most vulnerabilities when they are introduced. For instance, the same downstream vulnerability can arise from any of the following: a planner failing to include an input validation requirement, a coder meeting functionality demands with unsafe code, or a reviewer conducting only shallow inspections. And this goes beyond recognizing that the end product is unsafe.

Existing work addresses parts of this problem separately. One line of work secures the output of LLM-based code generation through static analysis, fuzz testing, and staged security review, catching vulnerabilities after the code has been produced. A second line of work studies why multi-agent systems fail in general, building taxonomies and automated attribution methods for task failures such as coordination breakdowns, planning errors, and verification gaps. A third line of work makes hand-offs traceable and accountable, instrumenting a structured pipeline to quantify which stage repairs or introduces an error and showing that accountable hand-offs measurably improve pipeline accuracy over unstructured ones. Related work on boundary metadata collapse further studies what contextual information is lost at the hand-off itself, independent of task correctness. None of these directly answers a narrower and, for secure software engineering, more actionable question: when a multi-agent pipeline introduces a specific class of security vulnerability, which agent and which reasoning step is responsible, and why.

2 Related Work

2.1 LLM Code Generation Security

Empirical studies show that LLM-generated code is frequently insecure. Khoury et al. [6] asked ChatGPT to generate 21 security-relevant programs and found that 16 of them fell below minimal security standards on first generation; the model could usually acknowledge the flaw and produce a corrected version when asked, yet often reproduced related weaknesses in subsequent generations. Two recent multi-agent systems try to prevent such vulnerabilities during generation rather than after it. AutoSafeCoder [1] couples a coding agent with a static-analysis agent and a fuzzing agent in an iterative loop; on the SecurityEval benchmark it reduces the share of vulnerable samples from 49% (59 of 121) for GPT-4o alone to 36% (44 of 121), a 13% reduction, while functionality on HumanEval drops by only about 3% (pass@1 from 0.9085 to 0.8780). MACGen [2] decomposes secure code generation into a planner, a security advisor, a code generator, and a reviewer that exchange only structured artifacts instead of a shared dialogue, a design the authors show avoids the role interference and context bloat of shared-dialogue approaches; it improves the joint functionality-and-security pass rate (F&S@1) over direct prompting by 19.61 percentage points on average on CWEval across six LLMs and five languages, and by 9.95 and 11.18 percentage points for GPT-4o and GPT-4o-mini on BaxBench. Both systems lower the rate at which vulnerabilities survive the pipeline, but both evaluate only the final artifact: when a vulnerability is detected, neither can say which agent or hand-off introduced it.

2.2 Multi-Agent Failure Taxonomies and Attribution

Cemri et al. [3] built the MAST taxonomy from a grounded-theory analysis of over 150 execution traces, validated to an inter-annotator agreement of kappa = 0.88, and used it to annotate MAST-Data, a public dataset of 1,642 traces from seven multi-agent frameworks; the taxonomy defines 14 failure modes in three categories (system design, inter-agent misalignment, and task verification). Three of its findings bear directly on this work: task-verification failures (premature termination, no or incomplete verification, and incorrect verification) account for 6.20%, 8.20%, and 9.10% of annotated failures respectively; MAST-guided interventions improved ChatDev task success by 9.4 and 15.6 percentage points in two case studies; and the authors caution that isolated fixes do not substitute for structural redesign. MAST is, however, a general-purpose taxonomy: none of its modes distinguishes a security-relevant failure from any other task failure.

Automated attribution of a failure to a responsible agent and step is known to be hard. On the Who&When dataset [7], which pairs failure logs from 127 multi-agent systems with human-annotated responsible agents and decisive error steps, the best automated method identifies the responsible agent only 53.5% of the time and the decisive step only 14.2% of the time, with some methods performing below random. TRAIL [8] reports a similar picture for trace debugging over 148 human-annotated traces (1,987 spans, of which 575 contain at least one error): the best long-context model tested scores 11% joint accuracy. Chen et al. [9] argue that part of this difficulty is an artifact of how attribution is evaluated: benchmarks such as Who&When expose only agent outputs, and their analysis finds that at least 21% of those failures cannot be reliably attributed from output-only logs. Their TraceElephant benchmark therefore provides full execution traces and reproducible environments for 380 traces (220 of them failed) from three agentic systems; with full traces, attribution reaches 65.9% agent-level and 30.3% step-level accuracy on average, improving step-level accuracy by up to 76% over the output-only setting. All of this work targets general task failure; none of it distinguishes security-relevant outcomes or vulnerability classes.

2.3 Hand-off Traceability and Accountability

Barrak [4] instruments a Planner-Executor-Critic pipeline with structured, accountable hand-offs and a blame-assignment method that labels each stage's answer as a repair, a harm, or a no-op against ground truth. Across eight configurations of GPT-4o, Claude Sonnet 4, and Gemini 2.5 Pro on three benchmarks totalling 673 prompts, the accountable protocol improves accuracy in most configurations, most dramatically on PythonIO, where the Claude-only pipeline rises from 61.42% to 97.64% (+36.22 percentage points), although a few configurations decline slightly. The study also quantifies role-specific behaviour: planner error is the strongest predictor of pipeline failure, with planner error rates ranging from 0.79% to 40.49% across models and tasks, and Claude Sonnet 4 is the strongest executor, repairing 10.01% of inherited errors with a 0.25% harm rate. Because the blame labels rely on discrete correct-or-incorrect answers, the method does not transfer directly to open-ended code artifacts, and it says nothing about security. Wang et al. [5] study the hand-off itself: when an upstream interaction is compressed into a hand-off artifact, summaries preferentially preserve operational facts while dropping the boundary metadata that governs how those facts may be used, a failure mode they call summary collapse. Boundary-marker survival falls from about 0.80 for uncompressed hand-offs to about 0.57 under a 25-word budget while operational-fact survival stays near ceiling, and downstream leakage depends on how explicitly the boundary was stated. Neither study attributes specific security vulnerability classes to the hand-off and reasoning step that introduced them, which is the gap addressed here.

3 Dataset

The evaluation uses coding tasks taken from real projects rather than synthetic benchmarks, spanning web, backend, and API development, so that the planner-coder-reviewer interaction is exercised the way it occurs in practice and the vulnerability classes observed are the ones practitioners actually introduce. The final task count, source list, and any de-duplication applied will be reported with the completed evaluation.

Concretely, candidate tasks are drawn from issue-resolution pairs in open-source Python and Java repositories: a merged pull request that closes a linked issue supplies both the task description (the issue text) and a reference implementation (the merged diff) against which the pipeline’ s output can later be compared. A candidate is retained only if resolving it requires touching more than one function or file and involves at least one design decision, such as how to validate input or which library to call, rather than a single-line fix, so that the task genuinely exercises the planner-coder-reviewer hand-off instead of being

resolvable by the coder alone. This filter is applied against a fixed, published checklist so that task selection remains auditable rather than ad hoc.

Task selection checklist (auditable criteria). A candidate is retained if and only if it satisfies all of the following:

- Linked issue + merged PR pair exists in a public GitHub repository under an OSI-approved license (MIT, Apache-2.0, BSD-2/3, or GPL-compatible).
- Diff touches ≥ 2 functions or ≥ 2 files.
- Resolution requires at least one non-trivial design decision (input validation strategy, secrets handling, dependency choice, serialization boundary, authorization check, or cryptographic primitive).
- Issue text is self-contained enough to serve as a planner prompt without external context beyond the repository's README.

Ground-truth vulnerability labels are produced in two passes. A static-analysis pass (using tools such as Semgrep or CodeQL) scans the final artifact and the intermediate artifacts recorded at each hand-off to generate candidate vulnerability flags, and a manual review then discards false positives, assigns each confirmed finding its CWE (Common Weakness Enumeration) class, and marks the hand-off at which the vulnerability first appears. These labels serve as the reference for the attribution metrics defined in Section 4.4.

Training-data overlap screening (external-validity threat mitigation). Before treating reference implementations as ground truth at scale, each candidate task is screened for likely membership in the pre-training corpora of the evaluated LLMs. Screening combines (i) n-gram overlap against publicly documented training-data snapshots where available, (ii) a membership-inference style probe that asks the model to complete truncated fragments of the reference diff under temperature 0, and (iii) manual inspection of any near-exact matches. Tasks flagged as high-overlap are either excluded or retained with an explicit overlap annotation so that sensitivity analyses can be reported.

Following the observability findings of Chen et al. [9], who show that output-only logs leave at least 21% of failures unattributable and that full traces improve step-level attribution accuracy by up to 76%, the logging layer described in Section 4.1 records complete inputs, reasoning traces, and outputs at every hand-off, so that the dataset supports step-level attribution rather than only end-state comparison.

4 Methodology

The proposed framework consists of the following components:

4.1 System Architecture

In this pipeline, there is a series of special agents: a planning agent which breaks down the programming job into smaller ones and mentions any security requirements; a coding agent which carries out the execution of every subtask; as well as a reviewing agent whose function is to appraise if the implementation can be accepted or not. At each exchange point between the agents, there is also a log added that specifies what was exactly passed on to the next agent, as well as the latter' s reasoning trace and output– giving total observability in execution compared with the partial, output-only observability mode that was seen to decrease attribution accuracy in [9]. Although this structure complies with the general model of accountable, well-organized transfers that have been found useful in enhancing pipeline reliability [4], its logging layer serves a different purpose than blame assignment at the level of answers –

i.e., it supports post-hoc vulnerability attribution.

Fig. 1. HOSA Architecture

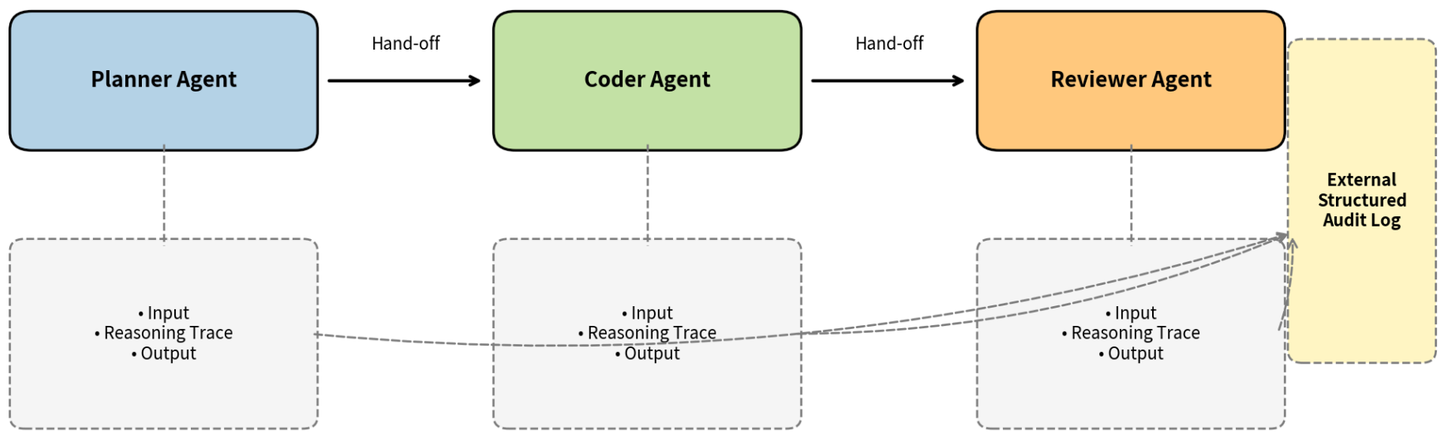

Fig. 1. Architecture diagram: Planner to Coder to Reviewer, with a logging layer at each hand-off recording input, reasoning trace, and output. Full traces are written to an external structured audit log; only functional requirements and security constraints are passed downstream (context decoupling).

A logging layer that records complete reasoning traces at every hand-off risks a failure mode of its own: naively forwarding accumulated traces downstream would inflate the context each subsequent agent must process, degrading the reviewer's performance and, over a long pipeline, exceeding the model's context window. HOSA avoids this by decoupling the audit log from the active prompt context. Each agent's full reasoning trace and boundary metadata are serialized and written to an external structured log reserved for later attribution, while the next agent in the pipeline receives only the functional requirements and security-relevant constraints it needs to act on, not the upstream reasoning that produced them. This keeps each agent's working context bounded by its own subtask rather than by the accumulated history of the pipeline, so context does not grow with pipeline depth even though the additional logging and hand-off structure does add some token overhead relative to an unstructured baseline (Section 5.2).

4.2 Security-Specific Taxonomy

Based on the reasoning recorded in the logging layer, we define eight vulnerability classes relevant to generated code and map each to the hand-off stage at which it is most commonly introduced:

- **Injection flaws** (SQL, command, or template injection) occur at the coder stage when an implementation choice meets functional requirements but doesn't sanitize external input.
- **Hard-coded credentials or secrets** happen at the coder stage, usually when the planner fails to specify a configuration or secrets-management requirement.
- **Unsafe dependency use** takes place at the coder stage when an old or known-vulnerable package is added without anyone (planner or reviewer) flagging a version constraint.
- **Improper input validation** may begin in the planner stage (an unstated validation requirement) or the coder stage (a stated requirement that's not fully implemented).
- **Insecure deserialization** comes up at the coder stage when someone uses a serialization library without limiting what it will deserialize.

- **Broken access control** can usually be traced back to the planner stage—an authorization requirement that's assumed rather than explicitly stated or broken down into components.
- **Weak cryptographic practice** (e.g. deprecated algorithms or hard-coded initialization vectors) is introduced at the coder stage.
- **Insufficient or incorrect verification** is not a vulnerability in the generated code itself. It happens when reviewers fail to detect vulnerabilities from one of the other classes listed above; this class corresponds to task-verification failures identified by MAST's taxonomy [3].

The first six are what MAST calls system design issues when there is no planner requirement stated. They become reasoning-action mismatches if the cause can be traced to something the coder did that violated a specified requirement (see [3]). Accordingly we place these classes either at the planner stage or coder stage depending on whether the cause can be traced to a failure in the planner's specification versus an incorrect assumption made by a coder that there was such a specification. The seventh class is aligned with MAST's task-verification category for security-specific failures. Defining our taxonomy in terms of these same stages used in [3] allows cross-comparison with general multi-agent failure analysis approaches while addressing shortcomings identified with general failure taxonomies in describing security-related problems.

Pilot-to-full taxonomy mapping. The pilot instantiation (§5.1) used a deterministic regex/substring checklist over four classes. The mapping to the full eight-class taxonomy is:

1. **Hard-coded credentials** → Hard-coded credentials or secrets (full class 2).
2. **Missing input validation** → Improper input validation (full class 4); may also surface as a contributing planner-origin cause under Broken access control when the missing check is authorization-related.
3. **Unsafe dynamic execution** → Injection flaws (command/template injection subset of full class 1) and, when the dynamic construct is a deserializer, Insecure deserialization (full class 5).
4. **Unsafe dependency import** → Unsafe dependency use (full class 3).

The remaining full-taxonomy classes (Broken access control, Weak cryptographic practice, Insufficient/incorrect verification) have no direct pilot counterpart and will be detected in the full campaign via static analysis (CodeQL/Semgrep) plus the manual CWE labeling pass described in §3. Checklist rules for these classes are provided in Appendix A (to be finalized with the full evaluation).

4.3 Attribution Method

If a vulnerability is found in the final output, the attribution method will look through the chain of reasoning to find the hand-off that likely caused it. Attribution scores candidate hand-offs h according to Eq. (1)

$$A(h) = w_1 \cdot C(h) + w_2 \cdot R(h)$$

where A(h) is the attribution score for hand-off h; C(h) is the context-overlap score between the hand-off's input/output diffs and the vulnerability code location, computed by structurally and textually comparing the artifact at the hand-off against the vulnerability span; R(h) is the reasoning-trace similarity score between the hand-off's recorded reasoning and vulnerability-introducing reasoning patterns labeled for the corpus, computed as embedding similarity against the exemplars for each taxonomy class in Section 4.2; and w_1 and w_2 are weighting coefficients determined empirically via a grid search on the manually labeled hand-off ground truth described in Section 3.

Embedding-based scorer (full protocol). In place of the pilot's lexical token-overlap and Dice similarity,

the full protocol computes:

- **C(h)**: structural AST-level overlap (Python ast / Java parser) of the vulnerability span with the artifact produced at hand-off h, combined with cosine similarity of sentence-transformer embeddings of the surrounding code context. Default embedding model: all-mpnet-base-v2 (or a code-specialized alternative such as CodeBERT if compute permits).
- **R(h)**: cosine similarity between the embedding of the hand-off's recorded reasoning trace and a set of class-specific exemplar traces. Exemplars are constructed by (i) taking the manually labeled ground-truth origin hand-offs from the dataset, (ii) extracting the free-text reasoning that preceded the first appearance of the vulnerable construct, and (iii) clustering within each taxonomy class to obtain 3–5 representative exemplars per class.
- **Grid search for w_1, w_2**: on a held-out subset of the manually labeled hand-offs, search $w_1 \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$ with $w_2 = 1 - w_1$; select the pair that maximises Attribution Accuracy (AA) on the held-out set. The selected weights are then frozen for the evaluation campaign.

The highest scoring candidate hand-off is attributed as the origin. This design mimics the backward-tracing logic employed in assigning blame along a structured pipeline [4], and the agent- and step-level attribution framework for general task failures [7], except that it replaces discrete correctness checks— which are only applicable to checkable answers— with continuous scores appropriate for open-ended code artifacts.

4.4 Evaluation Metrics

Building on the attribution score A(h) defined in Eq. (1) of Section 4.3, we define three metrics for evaluating HOSA against unstructured prompting and existing baselines:

$$VIR = N_v / N_t$$

$$AA = N^c / N_v$$

$$ES = (1/N_v)\,\Sigma\, sim(e_model, e_human)$$

VIR (Vulnerability Injection Rate) is the fraction of tasks in which at least one tracked vulnerability class was introduced: N_v is the number of tasks with at least one detected vulnerability, and N_t is the total number of tasks. VIR is analogous to the overall MAS failure rate reported by MAST [3], narrowed to security-relevant failures. AA (Attribution Accuracy) is the fraction of detected vulnerabilities that are correctly traced back to their originating hand-off: N_v is the total number of vulnerabilities detected across all tasks, and N^c is the number of those vulnerabilities for which HOSA's top-scoring hand-off matches the ground-truth origin recorded in Section 3. Finally, ES (Explainability Score) is computed via Eq. (4), where sim(e_model, e_human) measures how closely the model's explanation of the vulnerability's cause matches an independent human reviewer's explanation for the same case, following the annotation procedure used to establish MAST's inter-annotator agreement ($\kappa = 0.88$ in [3]).

5 Evaluation Protocol and Expected Outcomes

The full-scale campaign described here — four conditions (unstructured prompting, a reimplementation of AutoSafeCoder [1], a reimplementation of the accountable hand-off pipeline of Barrak [4] extended to emit security-relevant blame signals, and HOSA) run across the complete task set of Section 3 and multiple LLM families — was only partially completed before the OpenRouter API credits funding this study were exhausted. Sections 5.1 and 5.2 report preliminary results from a smaller pilot run using a lightweight

instantiation of HOSA, together with the deviations from the full protocol that this entailed; Section 5.3 states the expectations the full-scale campaign is designed to test, and Section 5.4 discusses threats to the validity of both the pilot and the planned campaign. The four-condition, multi-model campaign itself remains future work.

5.1 Pilot Instantiation and Deviations from the Full Protocol

Due to exhaustion of the OpenRouter API credits funding this study, the pilot reported below uses a single model (xiaomi/mimo-v2.6-flash) rather than the multiple LLM families the full protocol calls for, and covers 8 of the planned tasks with n = 2 runs each, rather than the complete task set of Section 3. Only two conditions were run: an unstructured baseline (a single planner-coder pass, no logging or structured hand-offs) and a lightweight instantiation of HOSA (“HOSA-lite”); the reimplemented AutoSafeCoder [1] and Barrak [4] baselines were not run in this pilot.

HOSA-lite also simplifies three components of the full protocol described in Section 4. First, in place of the eight-class taxonomy of Section 4.2, the reviewer agent applies a deterministic, regex/substring checklist over four vulnerability classes (hard-coded credentials, missing input validation, unsafe dynamic execution, and unsafe dependency import), which cover a subset of the full taxonomy and, by construction, perform no interprocedural or dataflow analysis. Second, in place of the embedding-based context-overlap and reasoning-trace similarity scores of Eq. (1), the pilot's attribution scorer computes C(h) and R(h) as lexical token-overlap and stopword-filtered Dice similarity respectively, with w_1 = 0.6 and w_2 = 0.4 fixed a priori rather than grid-searched against manually annotated ground truth; ground truth itself is defined programmatically, as the first coder hand-off whose output contains the flagged code pattern, rather than by the static-analysis-plus-manual-review protocol of Section 3. Third, in place of the human-validated Explainability Score (ES) of Eq. (4), the pilot reports an Explainability Coverage (EC) proxy: the fraction of hand-offs carrying a non-empty reasoning trace. EC measures whether an explanation was produced at every hand-off, not how closely it matches an independent human reviewer's account, and should not be read as an estimate of ES.

5.2 Pilot Results

Table 1 reports the pilot's results, with VIR defined at the task level exactly as in Eq. (2) (a task counts as vulnerable if any of its units carries at least one finding). Two variants of vulnerability counting are shown: a raw, per-unit count, and a deduplicated count that collapses repeated findings sharing an underlying root cause (for example, the same missing-validation pattern recurring across two decomposed subtasks) into one group before computing VIR, since the unstructured baseline reviews a task as a single unit while HOSA-lite reviews each of its planner-issued subtasks separately, and the two conditions are not directly comparable on a per-unit basis without this correction. HOSA-lite's task-level VIR (0.625 ± 0.177) exceeds the baseline's (0.375 ± 0.177) under both the raw and the deduplicated accounting, though the margin narrows under deduplication, consistent with part of the raw gap being attributable to decomposition increasing the number of reviewable units rather than to a change in the underlying code's security properties.

Metric	Baseline (unstructured)	HOSA-lite (pilot)
VIR — task-level (Eq. 2)	0.375 ± 0.177	0.625 ± 0.177
VIR — dedup (root-cause groups / task)	0.625 ± 0.177	1.125 ± 0.177
AA (Eq. 3, lexical-overlap scorer)	N/A	1.000 ± 0.000 (28/28)
EC — Explainability Coverage (proxy for ES)	0.000 ± 0.000	1.000 ± 0.000

Tokens / task (mean)	3,714 ± 37	4,692 ± 149
Wall-clock latency / task, s (mean)	82.3 ± 4.4	72.3 ± 3.1

Table 1. Pilot-study comparison of an unstructured baseline against a lightweight instantiation of HOSA (8 tasks, n = 2 runs each, single model: xiaomi/mimo-v2.6-flash via OpenRouter). Reimplemented AutoSafeCoder and Barrak baselines were not run in this pilot; values are mean ± sample standard deviation across runs.

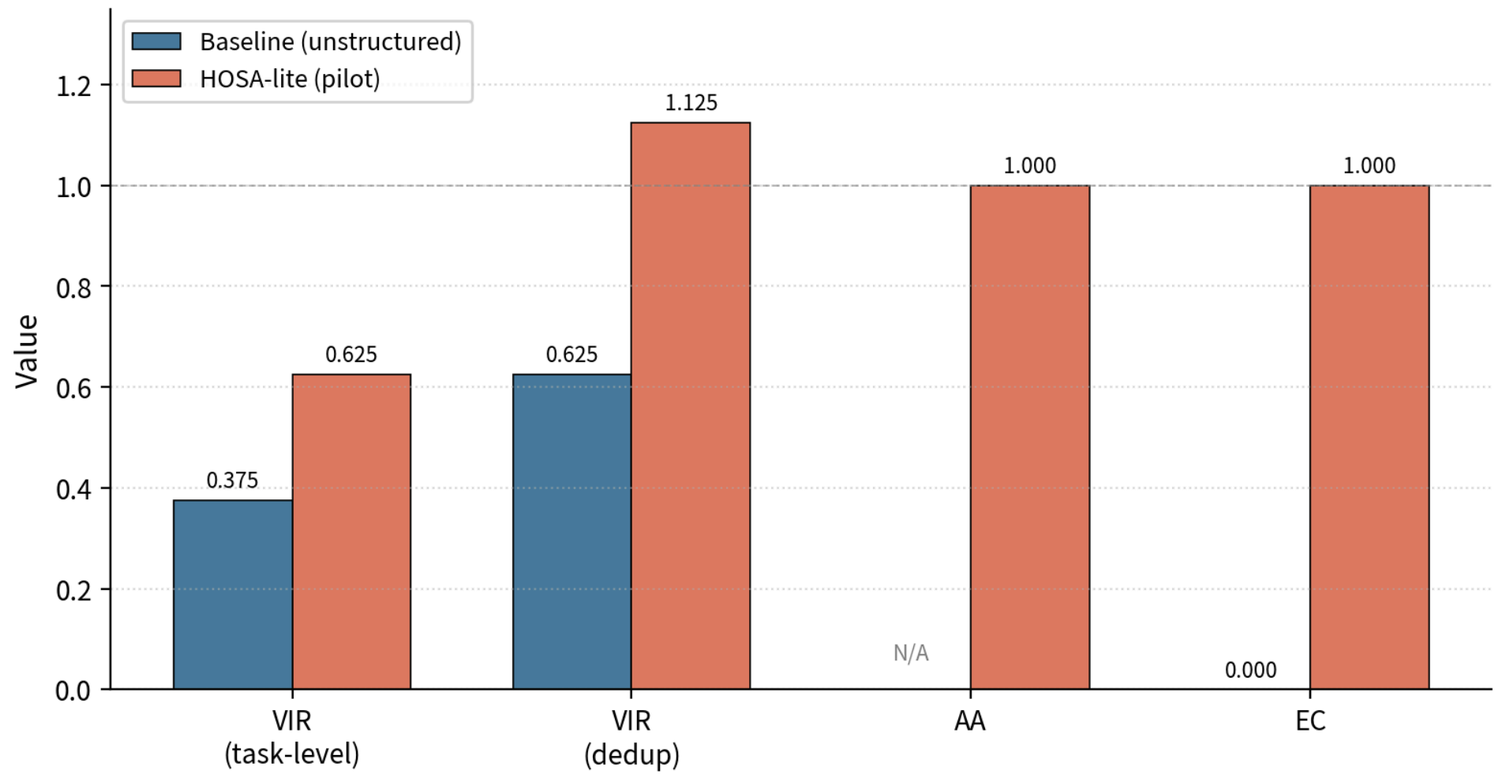

Fig. 2. Bar chart comparing the pilot's two conditions (unstructured baseline vs. HOSA-lite) across VIR, AA, and Explainability Coverage (EC). A four-condition comparison including the reimplemented baselines awaits the full-scale campaign.

Attribution Accuracy and Explainability Coverage were both at ceiling across every pilot run (AA = 1.000 ± 0.000, 28 of 28 flagged findings correctly attributed to their originating hand-off under the programmatic ground truth; EC = 1.000 ± 0.000 for HOSA-lite against 0.000 for the baseline, which carries no reasoning trace at all). Because both metrics use proxy definitions narrower than their full-protocol counterparts (Section 5.1), this result should be read as evidence that hand-off-level logging and lexical-overlap attribution behave consistently on this pilot's task set, not as a claim about the harder embedding-based and human-validated metrics of Section 4.4.

One pilot finding anticipates the first expectation of Section 5.3 below: for the pilot's shell-command-execution task, the planner's own decomposition explicitly suggested the vulnerable pattern (“e.g., subprocess.run with shell=True”) before any coder hand-off, and the coder's recorded reasoning trace explicitly deferred validation to a later subtask; attribution correctly named the coder hand-off as the code's literal origin, but the reasoning trace makes the planner's contribution to the underlying design decision directly inspectable. This is the kind of planner-origin cause the full protocol's taxonomy (Section 4.2) is designed to surface, obtained here from a single-task, single-model pilot rather than the full campaign. On cost and latency, HOSA-lite used more tokens per task than the baseline (4,692 ± 149 vs. 3,714 ± 37) but comparable or lower wall-clock latency (72.3 ± 3.1 s vs. 82.3 ± 4.4 s), suggesting that the structured pipeline's per-hand-off logging and additional agent calls do not necessarily cost wall-clock time even though they cost tokens.

5.3 Expected Outcomes

Three expectations follow from published results. First, planner-origin omissions should account for a

large share of injected vulnerabilities: Barrak [4] finds the planner to be the strongest predictor of pipeline failure, with planner error rates reaching 40.49% on the hardest benchmark, so requirements that never enter the decomposition, such as the validation requirement above, are a plausible blind spot in current practice. Second, review-origin failures will be frequent: verification failures account for 6.20%, 8.20%, and 9.10% of the 1,642 annotated traces in MAST-Data [3], and MAST's authors observed that existing verifiers often perform only superficial checks even when prompted to verify thoroughly. Third, general-purpose attribution baselines should underperform a security-specific method: even with full traces, the best automated methods reach only 53.5% agent-level and 14.2% step-level accuracy on Who&When [7] and 11% joint accuracy on TRAIL [8], and these numbers fall further under output-only observability [9]. Vulnerabilities that span two or more hand-offs, such as the example above, should be harder than single-origin ones, because the method must divide blame across hand-offs rather than assign it to one.

5.4 Threats to Validity

Construct validity is threatened by two design choices noted above: the pilot's four-class checklist and lexical-overlap scorer are simplified stand-ins for the full eight-class taxonomy and embedding-based attribution score of Section 4, so the pilot results (Section 5.2) should be read as evidence about the hand-off logging architecture rather than about the final attribution method; and the taxonomy of Section 4.2 was authored by the same team running the evaluation, which is why the human study proposed in Section 6 deliberately recruits reviewers who did not take part in its construction.

Internal validity is threatened by the pilot's programmatic ground truth, which defines a vulnerability's origin as the first coder hand-off whose output contains the flagged pattern; because both this label and the pilot's AA score are derived from similar surface features of the same artifact, the pilot's AA figures should not be read as an estimate of attribution accuracy under the full protocol. The full protocol's static-analysis-plus-manual-review labels (Section 3) are derived independently of the attribution scorer and do not share this dependency.

External validity is limited by the use of a single model family in both the pilot and the full protocol so far, and by sourcing tasks from public GitHub repositories, which may overlap with the pretraining data of the LLMs under evaluation; a model that has memorized a reference fix could appear more, or less, vulnerable and attributable than it would on genuinely novel code. Screening the task set for likely training-data overlap is planned before the reference implementations recovered during dataset construction (Section 3) are treated as ground truth at scale.

Conclusion validity is limited by the pilot's small sample (8 tasks, n = 2 runs per condition), reflected in the wide confidence intervals in Table 1; the full-scale, multi-model campaign described in Section 5 and Section 6 is designed to address this once API access is restored.

6 Conclusion

This paper introduces HOSA, a structured hand-off framework for attributing security vulnerabilities in multi-agent LLM code generation pipelines to the specific agent and reasoning step responsible for introducing them.

The first contribution is a security-specific failure taxonomy of eight vulnerability classes, organized along the same planning, implementation, and review stages used by general-purpose multi-agent failure taxonomies. The second contribution is an attribution method that scores each hand-off by context

overlap with the vulnerability's location and by similarity to known vulnerability-introducing reasoning patterns, extending backward-tracing blame attribution from discrete, checkable answers to open-ended code artifacts. The third contribution is an evaluation protocol, with associated metrics, for comparing HOSA against unstructured prompting and the closest published baselines.

This work has several limitations. The full-scale, multi-model evaluation against reimplemented baselines has not yet been completed; in its place we report preliminary results from a small pilot (Section 5) using a lightweight, checklist-based instantiation of HOSA covering four of the taxonomy's eight vulnerability classes, a lexical-overlap attribution scorer with fixed rather than grid-searched weights, and an Explainability Coverage proxy rather than the human-validated Explainability Score of Section 4.4; the reasoning-trace similarity component itself has only been validated qualitatively so far; and the ground-truth vulnerability & hand-off labels used in the full protocol rely partly on manual review, which the pilot's programmatic ground truth does not yet substitute for at scale.

Furthermore, both the full protocol and the completed pilot use a single LLM family; the pilot itself was additionally limited to 8 tasks and 2 runs per condition by the exhaustion of the API credits funding this study before the planned multi-model campaign could be run. Extending the evaluation across multiple LLM families and to the full task set remains important future work, since general-purpose attribution is hard even with full traces and multiple model families; other related work has noted this challenge (see references [7, 8, 9]).

As such, future work includes completing the full-scale, multi-model empirical evaluation described in Section 5, including the reimplemented AutoSafeCoder and Barrak baselines and the complete eight-class taxonomy, once funding for API access is restored; replacing the pilot's lexical-overlap attribution scorer and Explainability Coverage proxy with the embedding-based and human-validated metrics of Section 4.4; extending the taxonomy to additional languages and frameworks; conducting a human study to validate the explainability score with reviewers who did not participate in building the taxonomy; and cross-LLM evaluation.

Human validation study protocol (ready to run). The study will recruit independent reviewers with baseline computer science or data science training who did not contribute to the taxonomy of Section 4.2 (target pool: ACM MPSTME student chapter members and faculty volunteers). Protocol:

5. **Materials.** A shared subset of 20–30 flagged vulnerabilities drawn from the full-scale evaluation, each packaged with (a) the final vulnerable artifact, (b) the complete hand-off log (inputs, reasoning traces, outputs), and (c) the CWE label assigned by the static-analysis + manual pass.
6. **Task.** For each case the reviewer independently (i) names the originating hand-off (planner / coder / reviewer) and (ii) writes a free-text root-cause explanation in ≤ 80 words.
7. **Agreement.** A subset of cases is double-annotated; inter-annotator agreement is reported with Cohen's κ on the hand-off label and with pairwise cosine similarity (same embedding model used for R(h)) on the free-text explanations.
8. **Scoring.** HOSA's reasoning-trace attribution is compared against the human explanations using the same sim(e_model, e_human) function defined for ES in Eq. (4). Reviewers are blinded to HOSA's top-scoring hand-off until after they have submitted their own labels.

7 References

1. Nunez A, Islam NT, Jha SK, Najafirad P (2024) AutoSafeCoder: A multi-agent framework for securing LLM code generation through static analysis and fuzz testing. arXiv:2409.10737

2. Yu M, Choi J, Lee Y, Paek Y (2026) MACGen: Toward functionally correct and secure code generation via multi-agent collaboration. arXiv:2608.25457

3. Cemri M, Pan MZ, Yang S, Agrawal LA, Chopra B, Tiwari R, Keutzer K, Parameswaran A, Klein D, Ramchandran K, Zaharia M, Gonzalez JE, Stoica I (2025) Why do multi-agent LLM systems fail? arXiv:2503.13657

4. Barrak A (2025) Traceability and accountability in role-specialized multi-agent LLM pipelines. arXiv:2510.07614

5. Wang Y, Goyal A, Chandrasekharan E, Sundaram H (2026) Facts without rules: Boundary metadata collapse in multi-agent LLM handoffs. arXiv:2608.29028

6. Khoury R, Avila AR, Brunelle J, Camara BM (2023) How secure is code generated by ChatGPT? In: 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp 2445-2451

7. Zhang S, Yin M, Zhang J, Liu J, Han Z, Zhang J, Li B, Wang C, Wang H, Chen Y, Wu Q (2025) Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In: Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, pp 76583-76599. arXiv:2505.00212

8. Deshpande D, Gangal V, Mehta H, Krishnan J, Kannappan A, Qian R (2025) TRAIL: Trace reasoning and agentic issue localization. arXiv:2505.08638

9. Chen M, Wang J, Mu F, Wang Y, Liu Z, Feng H, Wang Q (2026) Seeing the whole elephant: A benchmark for failure attribution in LLM-based multi-agent systems. In: Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL), pp 19888-19905. arXiv:2604.22708