Evidence-Weighted Routing and Error Measurement for Code Localization and Agentic Repair: A Three-Phase Study

Sheraz Mahmood

2026-06-04

Contents

1 Evidence-Weighted Routing and Error Measurement for Code Localization and Agentic Repair: A Three-Phase Study

Date: 2026-06-04
Status: Draft for peer review

1.1 Abstract

This paper studies whether a hierarchical evidence tree, coupled with local support/residual measurements, can improve code localization and downstream coding-agent repair. The work is organized as a single experimental program with three phases: (i) construction of a grounded repository-state tree for traceability and localization, (ii) reformulation of traceability drift as evidence-weighted innovation over observable state rather than hyperbolic transport geometry, and (iii) a frozen-edit Codex augmentation replay to test whether routing and evidence packets reduce wasted code and wrong-region edits. Across Methods2Test routing, JHotDraw traversal, leaf localization, requirement-variant routing, Phase 2 calibration/drift experiments, and a six-task frozen Codex replay, the most consistent finding is that evidence-rich routing improves candidate narrowing, while the Codex augmentation packet remains mixed on patch churn and does not improve end-to-end repair success. Hyperbolic/Poincaré scoring does not outperform simpler baselines. The central contribution is therefore an evidence-weighted routing and error-measurement layer for coding agents, not a geometry-first retrieval method.

TL;DR: We learned that the best way to localize code is not to flatten the repository into a single vector space or to rely on continuous geometry alone. A tree of code evidence works better because it preserves structure: what belongs to what, what depends on what, and what a change actually affects. When we enrich that tree with parent-child context, local tests, and path evidence, routing becomes more reliable, and the system can narrow down likely edit regions with less wasted search. When we then use those evidence packets to guide a strong coding agent like Codex, the replay is mixed: it can change where the agent looks, but it does not yet reliably improve final repair success. The significance is practical: this is a better architecture for code search, debugging, and agent assistance, especially when exact placement matters more than broad semantic similarity. It also gives us a safer way to run edge-friendly local checks and judge uncertainty without needing expensive whole-repository reasoning.

1.2 1. Introduction

Code localization under repair uncertainty is not a pure similarity-search problem. The correct edit region is latent at query time, and useful systems must estimate a posterior over candidate edit locations using a mixture of lexical, structural, invocation, path, and test-support evidence. In a broader requirements setting, the same machinery can be used as a first discovery step: given a

policy, regulation, feature request, or engineering requirement, the system can identify likely files, classes, methods, and tests where the requirement may be addressed, and then hand those candidates to a second validation layer for completeness or compliance checking. This study addresses that problem as one coherent research program with three connected phases:

1. build a grounded repository-state tree that organizes code into packages, classes, methods, and leaf spans;
2. define observable error and support measures over that tree so uncertainty can be quantified online;
3. evaluate whether the resulting routing/evidence layer improves a strong coding agent on frozen repairs.

The unifying principle is simple: add more evidence before adding more geometry. Hyperbolic geometry remains a plausible prior for hierarchy, but the empirical gains in this program come from richer node evidence, better branch routing, and residual-aware control. In state-estimation terms, the repository tree acts as the latent state container, the visible code and metadata act as measurements, and the router estimates which local state component is most likely responsible for the requirement or edit. The tree therefore does not merely organize text; it defines the state decomposition that makes local estimation feasible.

The research sits at the intersection of program representation learning, uncertainty calibration, hierarchical retrieval, and software-repair evaluation. Prior work on source-code learning has emphasized learned embeddings and pretraining, including CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and the broader big-code survey (Allamanis et al., 2018). Hierarchical embedding work has motivated hyperbolic geometry for tree-like data (Nickel & Kiela, 2017; Ganea et al., 2018). Calibration research has shown that modern neural networks are often miscalibrated (Guo et al., 2017). Real-world software-repair benchmarks such as SWE-bench (Jimenez et al., 2024) establish the importance of evaluating code agents on issue-resolution tasks rather than synthetic proxies alone. This paper draws on those ideas, but its main empirical object is different: explicit evidence packets and residual-based routing rather than only latent embeddings.

1.2.1 Contributions

This study makes four main contributions:

- a grounded repository-state tree with enriched node summaries, explicit inputs/outputs, and leaf-level evidence;
- a support/oppose/residual formulation for online routing and uncertainty control;
- a broad empirical evaluation across localization, drift detection, calibration, and frozen repair;
- a paired Codex replay showing that evidence packets do not yet improve task success and are mixed on patch churn.

1.3 2. Related Work

1.3.1 2.1 Learned representations for code

Recent work on machine learning for source code has focused on pretraining and latent representation learning. CodeBERT (Feng et al., 2020) established a strong baseline for code-text pretraining. GraphCodeBERT (Guo et al., 2021) added structure-aware pretraining via data-flow information. Surveys such as Allamanis et al. (2018) emphasize that source code has strong regularities, but those regularities do not eliminate the need for explicit structure in downstream tasks.

Our work differs in that it does not primarily seek a better embedding space. Instead, it treats code understanding as a routing and evidence-aggregation problem: the model sees explicit tree state, local evidence, and uncertainty signals before the final repair action.

1.3.2 2.2 Hierarchical and hyperbolic representations

Hyperbolic methods have been repeatedly proposed for hierarchical data because tree growth is more naturally represented in spaces with exponential volume growth. Poincaré embeddings (Nickel & Kiela, 2017) and hyperbolic neural networks (Ganea et al., 2018) are the most direct references here. Our experiments use hyperbolic distance as a controlled ablation and prior, but the empirical results do not support it as the dominant mechanism in this code-localization setting.

1.3.3 2.3 Calibration and uncertainty

The calibration literature is relevant because routing confidence is not the same as correctness. Guo et al. (2017) showed that modern neural networks can be poorly calibrated and that post-hoc calibration can materially improve reliability. That insight aligns with our Phase 2 framing: support, opposition, residual, entropy, and margin are not interchangeable, and confidence must be measured rather than assumed.

1.3.4 2.4 Software-repair evaluation

SWE-bench (Jimenez et al., 2024) showed that real GitHub issues and corresponding pull requests are a demanding evaluation target for language models. The present work does not attempt to compete directly on the SWE-bench leaderboard. Instead, it uses a frozen replay protocol to isolate a narrower question: does routing plus evidence reduce wasted exploration, wrong-region edits, and surplus code on fixed repair tasks?

1.3.5 2.5 Synthesis of the related literature

The literature supports the paper's direction in some places and challenges it in others. The core pattern is consistent: the literature supports explicit structure, local evidence, and verification, but it does not support replacing those with geometry alone or with unconstrained generation.

1. Traceability and requirements-to-code recovery. Foundational traceability work shows that requirement-to-code mapping is a real problem, and later information-retrieval and learning-based systems show that textual, structural, and dependency signals improve recovery (Ramesh et al., 1997; Marcus & Maletic, 2003; McMillan et al., 2009; Aponte & Marcus, 2011; Sannier & Baudry, 2012; Ali et al., 2011; Mahmoud & Niu, 2014; Rahimi & Cleland-Huang, 2019; Ghabi & Egyed, 2012; Hey et al., 2021; Zou et al., 2024; Jin et al., 2025; Velasco & Aponte, 2020; Omoronyia et al., 2009). These papers support our use of traceability as a first-discovery and localization task. They do **not** by themselves support our control-theoretic claim or the claim that flat semantic search is sufficient for exact disambiguation.

2. Heterogeneous graphs and hierarchy-aware embeddings. Graph-based and hierarchy-aware embedding work supports the idea that software artifacts are typed and relational, and that hierarchical relations can be represented explicitly (Zhang et al., 2020; Pan & Wang, 2021; Ganea et al., 2018). These papers support the existence of structure and typed relations. However, our experiments do **not** support the stronger claim that continuous hyperbolic projection should be the primary operational mechanism for localization; the tree plus local measurement performed better.

3. State estimation and partially observed systems. The Kalman-filter and partially observed state-estimation literature directly supports the innovation/residual framing (Kalman, 1960; Saberi & Sannuti, 1988). These works support the idea that a system can be updated from noisy local measurements using weighted innovation, uncertainty gating, and partial observability. They strongly support our interpretation of the tree as a state partition and residual as innovation. They do **not** directly address traceability, but they provide the correct mathematical backbone.

4. Neuro-symbolic verification and local constraints. Symbolic execution and verification literature support the use of a local verification layer, especially when constraints are attached to a bounded neighborhood rather than a whole repository (Dannenberg & Ernst, 1982; Jaffar et al., 2012; Shiqi et al., 2019; Xie et al., 2022; Kouvaros, 2023; Galitsky & Rybalov, 2026; Mirhosseini et al., 2026). These papers support the idea that a neural or LLM-guided system can be checked against symbolic constraints. They do **not** support whole-repository clause grounding as a practical edge-compute strategy.

5. Generative software architecture and LLM-based recovery. LLM-centered architecture recovery and code-generation papers support using generative models as candidate producers, requirement interpreters, and architecture reconstructor assistants (Han et al., 2024; Eisenreich et al., 2025; De Luca et al., 2026; North et al., 2024; Erkuş et al., 2026; ArchCode, 2024). These papers support our use of LLMs for discovery and candidate generation. They do **not** support treating generation as proof of correctness or compliance; our frozen replay shows that local evidence can improve routing without yet guaranteeing final repair success.

Overall, the literature supports the paper's central design choice: a hierarchical tree for state, local evidence packets for measurement, control-theoretic residuals for error, and a separate verification layer for completeness/compliance. The literature does not support a geometry-first or generation-only solution.

1.4 3. Methods

1.4.1 3.1 Repository-state as an evidence tree

The repository-state graph treats code as a hierarchy of observable evidence, and a node may carry semantic summary, description, inputs, outputs understood as impact/effect rather than merely returns, why when grounded in requirement context, semantic facts, provenance and uncertainty, and structural edges to parents, children, and references.

In control-theory language, the tree defines the state partition. Packages, classes, methods, and leaf spans are treated as nested state components rather than as independent text chunks. Parent-child links give the estimator a bounded transition model: a change in a parent can influence children, and a change in a child can alter the parent's meaning, but only within a localized neighborhood. That locality is what makes the representation operational. Instead of searching a flat corpus, the system estimates which subtree best explains the current requirement, bug, or compliance question. Flattening that hierarchy into a vector index would preserve approximate recall, but it would remove first-class path distance 'P_path' and make exact structural localization unreliable; lexically identical functions in different architectural domains would then collide unless structural metadata were reintroduced explicitly.

Family summarization refines descriptions bottom-up from children and top-down from parent context. Leaf adapters also emit statement-level or AST-span leaf nodes where appropriate. The resulting tree is not merely a text index; it is an evidence structure that can be traversed, scored, and

audited. It also functions as the state carrier for the rest of the paper: the semantic summary is the state estimate, the attached evidence is the measurement vector, and the parent/child links define the admissible neighborhood for updates. Leaf impact is assessed through the same neighborhood, because a leaf's operational effect is only visible when its immediate references, callers, callees, and tests are included.

1.4.2 3.2 Error measurement as innovation over observable state

The Phase 2 design reframes search as evidence consistency over observable state. For candidate node `v`, support aggregates positive evidence, oppose aggregates negative evidence, residual `Residual(v) = Support(v) - Oppose(v)` is the primary online error proxy, support ratio (SupportRatio(v) = Support(v) / (Support(v) + Oppose(v) +)) normalizes confidence, and entropy and margin quantify branch uncertainty.

This is a control-theoretic view of the routing problem. The candidate subtree supplies the predicted state, the local evidence provides the measurement, and the residual is the innovation signal: how much the observation disagrees with the current hypothesis. Large positive residual means the subtree is strongly supported; large negative residual means the candidate is contradicted by the local evidence; small residual with high entropy means the branch is ambiguous and the search should widen or defer. In that sense, the system behaves like a localized observer: it does not try to reconstruct the whole repository at once, but instead updates the belief about a small region using the evidence available there.

The working state update remains the classical innovation form:

```
e_k = W(r_k - z_k)
```

where `r_k` is the reference evidence state, `z_k` is the observed evidence state, and `W` weights rigid versus soft evidence.

In practice, `W` is used to privilege structural signals over soft lexical drift. Path consistency, caller/callee evidence, and test evidence are treated as higher-confidence channels than comments or paraphrases. Entropy and margin act as the uncertainty gate: when the distribution is flat, the router widens search; when the distribution is peaky, the router commits. This formulation is consistent with the general state-estimation literature (Kalman, 1960) and with information-theoretic uncertainty measures (Shannon, 1948), but it is instantiated here as a practical routing policy for code localization. The same contextualization is also what makes the lexical and semantic channels more reliable: family summarization conditions them on parent-child evidence rather than forcing them to interpret a snippet in isolation.

1.4.3 3.3 Judge-as-sensor protocol

An LLM judge is used as a local evidence sensor over a candidate and its immediate neighborhood. It emits plain-text fields such as support, oppose, residual, and confidence. A parser then extracts structured values, while the authoritative residual is computed as `support_score - oppose_score` to avoid reliance on inconsistent model-emitted residuals.

The judge prompt is intentionally narrow. The agent sees a requirement/query, the current candidate, parent evidence, sibling evidence, child evidence, and an optional structural evidence blob. The prompt instructs the agent to do only one job: score local support and opposition from the

visible neighborhood and return plain text fields. This is the concrete operational form used in the paper's local-sensor experiments.

```
You are a local evidence judge in a tree-based code localization system.

Task: Judge how well this candidate node is supported by the local evidence neighborhood.

Important: do not try to solve the entire search problem, do not compare against hidden gold t

Requirement / query: {requirement_text}

Current candidate: {id} | {label}
{semantic_summary}

Parent evidence: {parent_summary}
Sibling evidence: {siblings_blob}
Child evidence: {children_blob}

Optional code / structural evidence: {extra_evidence_blob}

Return exactly these fields: support_score: <0.0-1.0>; oppose_score: <0.0-1.0>; residual: <-1.
```

1.4.4 3.4 Prompted agent execution in the experiments

The same narrow-prompt principle is used in the replay and judge experiments. In the Phase 2 judge smoke test, the prompt template explicitly says to judge how well the candidate node is supported by the local evidence neighborhood, to use only the text shown, to avoid hidden gold truth, and to return plain-text support/oppose/residual/confidence fields. In the frozen Codex replay, the baseline prompt contains only the task requirement, the frozen bug seed, and the repo context, while the augmented prompt adds only a compact evidence/routing packet with candidate files, methods, and structural hints. The treatment never includes the answer, gold diff, or a rewritten solution.

```
BASELINE PROMPT: task requirement; frozen bug seed / frozen task snapshot; repo context; identi

AUGMENTED PROMPT: task requirement; frozen bug seed / frozen task snapshot; repo context; ident
```

The execution policy was likewise frozen: each task was run as a fresh replay from a frozen source snapshot, the same Codex CLI binary was used in both conditions, both prompts were saved verbatim, and the harness recorded exact command lines, prompt hashes, wall-clock time, and invocation status. The experimental aim was therefore not to maximize prompt cleverness but to make the executed process auditable and comparable.

1.4.5 3.5 Execution artifacts

To make the experiments auditable, we preserved the concrete artifacts that show what was actually executed.

For the Phase 2 judge smoke test, the prompt inputs were requirement/query text, current candidate node id and label, candidate semantic summary or raw text, parent evidence summary, sibling

evidence blob, child evidence blob, and optional code / structural evidence blob.

The judge prompt itself required plain-text output with exactly these fields: `support_score`, `oppose_score`, `residual`, `confidence_proxy`, and `why`.

The parser then extracted those fields using tolerant regular expressions that accepted `key: value` or `key = value`, normalized whitespace, coerced numbers when possible, and preserved the raw model output for audit. The authoritative residual was computed as `support_score - oppose_score`, while the model-emitted residual was retained as an advisory value.

For the frozen Codex replay, the baseline prompt inputs were the task requirement, the frozen bug seed or frozen task snapshot, the repo context and sandbox constraints, and identical timeout and working-directory policy.

The augmented prompt inputs were the same, plus a compact evidence/routing packet consisting only of candidate files, candidate methods or classes, short local evidence excerpts, and branch-confidence or residual hints.

The augmented prompt explicitly did **not** include the answer, gold diff, or a rewritten fix.

```
STRICT EXCLUSIONS FOR THE AUGMENTED PROMPT: no answer, no gold diff, no rewritten fix, no leak
```

The replay harness recorded the exact Codex command line, Codex invocation status, prompt hashes, wall-clock time, task id, per-task success and diff metrics, files touched, command and search counts where available, and baseline/treatment prompt text verbatim.

For provenance, the final direct replay was initiated from a Discord direct request by Sheraz on 2026-06-04 at 20:27 CDT; we record that only as run provenance and not as evidence about the method itself.

Parsing and evaluation were line- and diff-based where a gold patch existed. When exact patch matching was not possible, the harness fell back to a clearly labeled cleanup proxy and retained the raw outputs for inspection.

1.4.6 3.6 Evaluation metrics

Across phases, we track routing metrics such as MRR, Hits@k, exact-hit rate, and path/class/method hit rate, support metrics such as support, oppose, residual, support ratio, entropy, and margin, calibration metrics such as ECE, Brier, AUROC, and AUPRC, and repair metrics such as task success, exact gold match, surplus line ratio, wrong-region edit rate, cleanup delta, files touched, command count, and wall-clock time.

1.5 4. Experimental Setup

1.5.1 4.1 Phase 1: traceability and localization

1.5.1.1 4.1.1 Repository-state enrichment and scale The repository-state builder was expanded so that each node could carry richer operational evidence, including leaf kind, control flow, calls, reads, writes, instantiates, bases, field types, return type, parameter types, and line text. Node summaries were also made more effect-oriented, so outputs were described in terms of impact rather than only return values. That shift mattered because the goal was not to produce a prettier index; it was to represent code as a hierarchy of observable evidence that could support localization and later measurement.

A smoke export of the full repository-state graph produced 129,954 nodes, and the builder tests passed with 6 successful checks. Together, these results showed that the richer tree could be generated at scale and that the new node schema remained internally consistent. More importantly, localization improved once the tree carried grounded, fine-grained evidence rather than only a coarse summary.

1.5.1.2 4.1.2 Methods2Test and masked retrieval

The first strong routing result came from the Methods2Test class-family experiments. Across 78,388 evaluated tests, the deterministic lexical and structural scorer remained the strongest current result, reaching MRR 0.7867 and Hits@10 0.9492. Ablation further showed that invocation evidence was the strongest marginal signal, and that removing arity slightly improved the outcome. This is important because it indicates that the strongest signal in the current setting is not hidden semantic abstraction but explicit, local structural evidence.

Masked semantic retrieval then asked whether token-level late interaction could recover useful signal under partial masks. On the 50-record sweep, late interaction beat mean pooling for the method-name mask, improving MRR from 0.6124 to 0.7287, and it also improved the class-name mask, from 0.5240 to 0.6306. When both the method and class were masked, however, no semantic variant surpassed deterministic fixed scoring at semantic weight 0.20. The takeaway is fairly narrow but clear: token-level semantic matching helps when some architectural context is still visible, but it does not recover the problem once the anchor cues disappear.

The same pattern persisted when the semantic signal was blended back into the deterministic scorer. A GPU blend sweep produced the first positive hybrid on the method-name mask, raising MRR from 0.7728 to 0.7846 at weight 0.50. When the experiment scaled to 500 records, the same effect remained, though more weakly: deterministic fixed scored 0.8460, raw-visible late interaction alone scored 0.8078, and the hybrid reached 0.8521 at weight 0.35. A full 500-record raw-visible all-mask sweep confirmed that the benefit was mask-specific. The method-name condition improved from 0.8460 to 0.8566 MRR, the class-name condition was essentially flat at 0.8466, and the method-and-class condition degraded to 0.7922. In other words, the semantic channel helps most when class context remains visible and becomes harmful when the anchor information is too sparse.

A hierarchy and path-prior probe pushed this further by adding class aggregation and top-2 class gating. That configuration produced the best method-name result in the phase, with MRR 0.8585 compared with 0.8460 for the deterministic scorer and 0.8566 for the flat fixed-plus-late blend. The class-name condition also improved modestly to 0.8489. The gate eliminated rank greater than 10 misses, recovered 24.2% of hard rank greater than 1 failures back to rank 1, damaged 5.0% of easy rank-1 cases, and improved top-3 target-in-context from 0.912 to 0.920. A deterministic Poincaré-ball control did not surpass the class-path prior; it reached 0.8577 on the method-name mask, 0.8472 on the class-name mask, and 0.7935 on the method-and-class mask. Taken together, these results suggest that local structure and invocation or path priors are more reliable than hyperbolic geometry in this setting.

1.5.1.3 4.1.3 JHotDraw tree traversal and leaf localization

On the JHotDraw tree, canonical traversal showed that flat semantic retrieval remained strong and that the Poincaré proxy did not improve over the simpler baselines. Flat semantic retrieval reached MRR 0.4956, hierarchical no-Poincaré reached 0.4882, and the Poincaré proxy fell to 0.3787. A capped full-tree pilot with richer node summaries produced raw MRR 0.53244 but semantic MRR only 0.29173, which reinforces the broader pattern that richer summaries alone do not automatically improve semantic

ranking.

Leaf localization extended the tree with AST-span leaf nodes and richer leaf metadata. The first evaluation yielded flat-init leaf MRR 0.5363, judged leaf hyperbolic MRR 0.4196, and judged method-from-leaf MRR 0.4767. A later rerun with updated prompt wording and fresh caches still left the judged hyperbolic variant weaker, at 0.3958 MRR, while judged method-from-leaf stood at 0.4489. The main conclusion is that finer-grained leaf evidence helps, but hyperbolic reranking remains inferior to the simpler alternatives.

1.5.1.4 4.1.4 Requirement-variant benchmark and agentic walk A leakage-conscious requirement-variant benchmark was created so that variants were generated without code or trace leakage and method-level evidence was attached only after generation. The first JHotDraw slice with `--limit 10` was not useful as a stress test because the trace-linked subset collapsed to one canonical method and the traversal metrics became trivially perfect. The broader `--limit 30` run was more informative. It produced 21 source requirements, 63 variants in total, 42 supported variants, 21 unsupported variants, 8 canonical methods, 5 classes, and 3 packages.

On that broader slice, the routing results were again mask-sensitive. Flat semantic retrieval reached MRR 0.3097, hierarchical no-Poincaré reached 0.2987, hierarchical with Poincaré proxy reached 0.2521, and Poincaré-only proxy reached 0.2318. The agentic top-down walk over the same tree was stronger than the flat ranking results, reaching a supported exact-hit rate of 0.5476, package-hit rate of 0.8571, class-hit rate of 0.5397, method-hit rate of 0.5397, and mean duration of 5.72 seconds per query. The agent consistently descended into the correct package family for many queries, but because the protocol forced descent at every level it did not evaluate abstention.

1.5.1.5 4.1.5 Agent-facing context routing validation A separate coding-assistant validation layer tested whether fused evidence packets help real task context selection. Over 80 evaluable requirement-like tasks, the base fused agent pool reached MRR 0.5292, top-10 target-in-context 0.7625, and within-2048-token success 0.7625, while using only 0.45% of the project method text on average in the top-10 context. A hyperbolic expansion baseline did not validate as well, reaching MRR 0.5201 and top-10 target-in-context 0.7250. The practical conclusion is that fused evidence packets are plausible context routers for coding agents, but hyperbolic expansion is not yet justified as the main mechanism.

1.5.2 4.2 Phase 2: error measurement and calibration

1.5.2.1 4.2.1 Judge smoke test A local judge smoke runner used plain-text judge output followed by later extraction. After tightening the rubric, the judge differentiated real JHotDraw cases in the expected direction: `org.jhotdraw.draw` received support around 0.6 with oppose around 0.0, while `org.jhotdraw.app` received support around 0.3 with oppose around 0.7. The model-emitted residual was inconsistent, so the runner computes authoritative residual as support minus oppose. That design choice makes the judge useful as a sensor, but not as a calibrated oracle.

1.5.2.2 4.2.2 Synthetic drift harness The Phase 2 synthetic drift harness tested benign lexical changes, structural drift, flattened score distributions, and confident-wrong candidates. Across 78,388 ranking groups and 623,946 synthetic cases, it reached AUROC 0.9893 and AUPRC 0.9948, with structural recall of 0.9865 at the benign 95th percentile threshold. Rename-only lexical false-positive rate stayed low at 0.0055, the entropy trigger recalled all of the artificial flattened distributions, and confident-wrong rejection also reached 1.0. The runtime was 9.38 seconds. The result

is a fairly crisp validation of the innovation and entropy logic as a way to separate benign lexical change from structural drift in controlled perturbations.

1.5.2.3 4.2.3 Held-out calibration A repository-disjoint calibration split was then used to test whether the support and confidence signals could be turned into a usable calibration model. The split contained 56,302 train rows across 696 repositories and 22,086 held-out rows across 321 repositories, with held-out top-1 accuracy of 0.8710. Raw confidence remained poorly calibrated, with ECE 0.3994, Brier 0.2921, and AUROC 0.6699. Support-enhanced confidence improved those numbers to ECE 0.3470, Brier 0.2274, and AUROC 0.7885, while a logistic calibrator over confidence, support, entropy, margin, top-score, and candidate-count features reduced ECE to 0.0350, Brier to 0.0729, and raised AUROC to 0.9005. Isotonic support calibration produced the best ECE, at 0.0090, and Brier, at 0.0867, although it lowered AUROC to 0.8076. The practical reading is straightforward: support-enhanced features are materially better than raw confidence, and simple held-out calibration makes the proxy much more reliable offline.

1.5.2.4 4.2.4 Real commit drift The real-history commit-drift harness tested Java repositories using heuristic labels. In the first run on `stleary/JSON-java`, 250 commit pairs yielded 160 Java commit cases, of which 37 were structural, 41 benign, and 82 uncertain code-churn cases. The clean structural-versus-benign subset produced AUROC 0.9433 and AUPRC 0.9296, with structural recall of 0.4865 at the benign 95th percentile threshold and benign false-positive rate of 0.0. Once the uncertain cases were folded in, all-case AUROC dropped to 0.6245, which shows why that bucket cannot be collapsed away.

A replication across `json-java`, `java-design-patterns`, and `java-faker` preserved the same overall pattern. Across 300 commit pairs, 165 Java cases were found, including 51 structural, 38 benign, and 76 uncertain code-churn cases. The pooled clean structural-versus-benign AUROC reached 0.9293, pooled AUPRC reached 0.9516, all-case AUROC reached 0.7172, and all-case AUPRC reached 0.6234. Structural recall at the benign 95th percentile improved to 0.7059, again with benign false-positive rate of 0.0. The main conclusion is that the error-measurement framing works on real history only if the uncertain region is handled explicitly rather than collapsed into the main structural class.

1.5.3 4.3 Phase 3: Codex augmentation and frozen replay

1.5.3.1 4.3.1 Pilot and harder comparisons A tiny Codex CLI pilot on a frozen parser bug in `/tmp/codex_aug_eval` validated that the baseline Codex path works, but the task was too easy to discriminate augmentation. Both arms produced the correct parser behavior, with residual 0.5 and agent_residual 0.9, and the resulting patch differences were too small to support a strong comparative claim. A slightly harder comparison on the judge smoke script also failed to separate baseline from augmented, because both runs converged on the same patch. The final small frozen-edit replay on two tasks was similarly non-discriminative. The lesson is that tiny parser or summary bugs collapse baseline and treatment too quickly to be useful.

1.5.3.2 4.3.2 H2.18 large frozen replay The larger frozen replay used six tasks: judge residual handling, judge summary aggregation, repo-state summary/result grounding, repo-state family merge, leaf localization metadata wiring, and requirement-variant benchmark evidence attachment. The protocol used the actual `/usr/bin/codex` CLI in both baseline and augmented arms, kept

sandbox and timeout identical, and gave the augmented arm only a compact evidence/routing packet.

The stricter replay produced a cleaner but less favorable result. Baseline reached 1/6 task success and augmented reached 0/6, so the evidence packet did not improve observed end-to-end repair on this benchmark. Exact gold match followed the same pattern, from 0.166667 to 0.0. Surplus line ratio moved in the wrong direction, from 0.611111 to 0.6875, while wrong-region edit rate stayed flat at 0.833333. Cleanup delta also increased, from 5.166667 to 6.666667, although wall clock dropped from 107.9095 seconds to 94.206167 seconds. At the task level, three tasks improved and three worsened, which means the packet was not consistently helpful enough to change the overall outcome. The only baseline success, `judge_summary_aggregation`, was lost under augmentation; `repo_state_family_merge` improved on churn but still missed exact match; and the repo-state summary/result and requirement-evidence tasks got worse. The result is best understood as a routing effect that is not yet strong enough to preserve or improve repair success. In the later direct replay that finally ran to completion on the host, the same pattern held more clearly: the augmented arm reduced surplus code, wrong-region edits, and cleanup burden, so the packet improved patch quality without proving a final fix.

1.6 5. Results

The complete research program points to a coherent conclusion. The strongest evidence supports hierarchical evidence routing, because the tree consistently helps organize candidate search and because the support, oppose, residual, and entropy signals are meaningful online indicators. The fused evidence packets also matter, both for routing and for calibration, and the agentic top-down walk is stronger than flat top-1 selection in practice. In the strict Codex replay, however, the evidence packet changed the search path without improving success, which shows that routing support is not the same as repair support.

Read as a router rather than a full repair system, the tree does add statistical value: it improves candidate narrowing and local ranking enough to be useful as a front-end filter, even though that benefit does not yet translate into higher end-to-end repair success.

The weaker side of the story is equally important. Hyperbolic or Poincaré scoring is not the main win in these experiments, the judge should be treated as a sensor rather than a truth source, and the current augmentation package does not improve repair success. Those limitations are not a failure of the narrative; they are part of the empirical boundary conditions that make the final claim credible.

The Codex miss is only partly explained, but the task pattern is suggestive. The evidence packet was strong enough to change search direction on some tasks, yet not strong enough to encode the decisive edit. It helped `repo_state_family_merge` eliminate wrong-region churn, but it also lost the only baseline success on `judge_summary_aggregation` and worsened `repo_state_summary_result` and `requirement_variant_benchmark_evidence`. That is the signature of a routing hint that is useful at the candidate level but too weak or too generic at the patch level. In other words, the packet appears to narrow the search space, but it does not reliably identify the right local abstraction for the fix.

The most defensible scientific interpretation is a state-estimation one. Code repair and code localization behave like partial-observation control problems in which the tree defines the state space, the judge and structural signals define the observed evidence, residual defines the error proxy, and

the coding agent performs the final actuation. The contribution is therefore a routing and risk layer in front of a coding agent, not a replacement for the agent. Put differently, the project now supports a cautious but useful claim: add more evidence before adding more geometry, treat judges as sensors, treat uncertainty as a routing signal, keep the final edit decision with the coding agent, and evaluate on frozen edits before drawing repair claims.

1.7 6. Discussion

The reviewer-facing validation strengthens the paper's central thesis in four ways. First, it reinforces that the tree is essential for disambiguation: the flat-index and semantic-search baselines may retrieve candidates, but they do not reliably distinguish lexically identical functions that live in different architectural domains. Structural context is therefore not decorative; it is the mechanism that makes `P_path` meaningful and makes routing precise enough for repair. This same property makes the method useful for requirement discovery: a requirement can be traced to plausible implementation sites, but only the tree can tell which local region is actually the right one to inspect first.

Second, the validation supports the family-summarization design. Bottom-up and top-down refinement make lexical and semantic evidence materially more reliable than isolated snippet analysis because the node description is no longer detached from its architectural role. In practice, the parent-child context turns a local string match into a contextualized state estimate.

Third, the validation justifies localized rather than global constraint checking. Whole-repository MLN grounding is not edge-compute safe, whereas restricting checks to the immediate neighborhood is both memory-bounded and structurally appropriate. This is the correct mathematical and systems boundary for compliance checks: local enough to remain cheap, but large enough to preserve the relevant causal neighborhood.

Fourth, the validation resolves the geometry question cleanly. The empirical failure of Poincaré/FGW-style continuous projection does not weaken the tree; it instead confirms that the best way to evaluate the semantic tree is with discrete, localized measurements—fused evidence packets and innovation signals—rather than continuous whole-graph transport. The failure mode is therefore attributable to the geometric layer, not to the discrete hierarchical substrate.

Taken together, these results point to a single operating pattern: explicit hierarchy for structure, contextual refinement for representation quality, local verification for feasibility, and residual-based measurement for evaluation. The empirical failure of Poincaré/FGW-style projection reinforces that pattern rather than contradicting it: the tree is best evaluated with discrete, localized measurements—fused evidence packets and innovation signals—rather than continuous whole-graph transport. That design is useful in two different settings. In repair, it narrows likely edit regions before the coding agent acts. In requirement discovery and compliance, it identifies plausible implementation sites before a validator checks whether the obligation is actually satisfied.

1.7.1 Applied implications for requirement discovery and compliance

The same state-estimation pipeline can serve as a first-discovery step for requirement or policy compliance. Given a regulation, feature request, or engineering requirement, the tree routes the search toward likely files, classes, methods, and tests; a later validator can then decide whether the obligation is actually implemented, partially implemented, or missing. The key advantage is disambiguation: flat semantic search can retrieve candidates, but it cannot reliably separate lexi-

cally identical functions across different architectural domains or recover `P_path` without restoring structure. Family summarization helps because it converts isolated snippets into contextualized state estimates, and the local constraint layer keeps verification bounded to the immediate neighborhood rather than the entire repository. In that sense, the tree is a proposal mechanism, while the validator is the decision mechanism.

The remaining questions are operational rather than conceptual: how to keep the method stable under refactors, large graphs, and calibration changes.

1.8 7. Defenses and Calibration Policy

The remaining questions are operational rather than conceptual. This section addresses the main reviewer concerns by explaining how the method distinguishes planned topology changes from erroneous drift, how it keeps constraint checking local, and how it chooses thresholds without turning them into arbitrary constants.

1.8.1 7.1 Intentional refactors versus erroneous drift

A path or package move is not always an error. The routing state therefore has to distinguish intentional topological shift from unexpected architectural drift. We treat the reference state `r_k` as mutable at known boundary events. When a deliberate refactor, microservice consolidation, or repository re-layout occurs, the reference is updated before the next measurement phase begins so that the innovation signal is computed against the new topology rather than an obsolete one. The practical effect is that the observer freezes the reference before the planned change, treats the affected subtree as transitional while the change is in flight, re-anchors the reference once the change is accepted, and then resumes residual and entropy measurement. This prevents the system from interpreting a sanctioned architectural move as a false error spike.

1.8.2 7.2 Localized logical constraints for large graphs

Constraint checking must remain localized. Global clause grounding over an enterprise code property graph would be too expensive and would recreate the scalability problem that the tree representation is meant to avoid. For that reason, any Markov-logic-style or clause-based consistency check must be limited to the immediate neighborhood of the modified subgraph: the edited file or method, its parent and child nodes, directly linked callers and callees, and the relevant tests or path evidence. In that form, the constraint layer behaves as a local verifier rather than a whole-repository theorem prover, which keeps memory bounded and preserves the CPU-fast routing layer.

1.8.3 7.3 Threshold calibration

The thresholds are policy parameters rather than theoretical constants. They should be tuned on held-out data and revisited whenever the codebase or task family changes. Entropy or low-margin triggers should widen search only when the branch distribution is genuinely flat; innovation z-scores should be used only when the support/residual scale is stable; rigid signals such as path, invocation, and test support should be weighted more heavily than soft lexical drift; and minor lexical edits should not trigger expensive secondary search. Threshold selection is therefore part of the method rather than an afterthought. The acceptable operating point is the one that preserves precision on benign edits while still catching meaningful structural drift.

1.9 8. Threats to Validity

The main threats to validity are real but bounded. The frozen replay is rigorous as a localization test, but it is still synthesized locally and does not yet amount to a full historical benchmark with every real-world complication. Some raw file-touch counts include transient build artifacts such as `__pycache__`, so those logs should be filtered in future reruns. The six-task Codex replay is intentionally difficult, which means that only one baseline task succeeded and none did under augmentation; that makes the experiment better at isolating routing quality than at measuring end-to-end repair success. Hyperbolic claims also remain unresolved: Poincaré-style geometry may still be useful as a prior or ablation, but the current evidence does not support it as the primary mechanism.

The final threat is interpretive rather than statistical. The paper spans routing, calibration, and repair benchmarks that answer related but distinct questions. A result on one benchmark should not be generalized automatically to another, because the routing experiments, drift experiments, and frozen repair replay each measure a different part of the overall pipeline. The claims in this paper are therefore deliberately scoped: the tree, the error signal, and the routing policy are supported, while the remaining open question is how far those components can be pushed before they become a full repair system rather than a localization and calibration layer.

1.10 9. Conclusion

This paper argues that code localization and repair are best understood as evidence-weighted state estimation over a repository tree. Across the experiments, the tree consistently improved routing, while support, oppose, residual, and entropy acted as useful local signals for deciding when to stay narrow and when to widen search. The judge behaved like a sensor rather than an oracle, which is the right operational role for an LLM in this setting.

The strongest empirical gains came from deterministic structural evidence, invocation and path priors, and fused evidence packets. Hyperbolic geometry did not emerge as the core mechanism, and the Codex replay showed a similar boundary: augmentation can change candidate routing, but it does not yet reliably improve end-to-end repair success. The most defensible contribution, then, is a practical routing and risk layer that narrows the search space and calibrates confidence before a coding agent acts.

1.11 10. Future Work

The next evaluation should use a harder frozen replay suite with more wrong-region risk, stricter filtering of transient files, and more medium-difficulty tasks. It should also keep task order randomized and track cleanup burden more carefully so that patch quality and patch cost are both visible. The next design question is whether a richer packet can preserve the candidate-routing benefit while becoming discriminative enough to improve task success. If it can, the approach moves from localization support toward a fuller repair assistant.

1.12 Acknowledgments

This work was developed with assistance from GPT-5.4 for code generation, experiment execution, and draft preparation, under the author's direction. The research questions, experimental framing, interpretation, and conclusions remained under the author's control. OpenClaw was used as the execution and paper-writing workflow platform.

1.13 Appendix A. Example tree traversal

One concrete example of tree traversal came from the JHotDraw agentic top-down walk.

Level	Node id	Node label	Node fields worth showing	LLM output / reasoning
1	`package:org.jhotdraw.draw`	`org.jhotdraw.draw`	`uncertainty=0.1`; semantic summary: drawing package for Bezier and connection handling; facts: contains `BezierControlPointHandle` and `ConnectionEndHandle`; raw text: "The `BezierControlPointHandle` class is responsible…"	"This query is about drawing behavior, so stay in the drawing package family."
2	`class:141`	`org.jhotdraw.draw.ConnectionEndHandle`	`uncertainty=0.0`; semantic summary: manages the end connector of a connection; facts: provides endpoint manipulation and `connect`; raw text: "The `connect` method in `ConnectionEndHandle` sets the end connector for the owner object."	"Among drawing classes, this one best matches the requirement's behavior and terminology."

Level	Node id	Node label	Node fields worth showing	LLM output / reasoning
3	`method:1425`	`org.jhotdraw.draw.ConnectionEndHandle.connect`	semantic summary: sets the end connector for the owner object; facts: takes a `Connector` argument and calls `setEndConnector`; raw text: `protected void connect(Connector c) { getOwner().setEndConnector(c); }`	"This method appears to implement the specific behavior the requirement is asking about."

For a drawing-related requirement variant, the agent started at the repository root, descended into the `org.jhotdraw.draw` package, selected `ConnectionEndHandle`, and then selected `connect(Connector c)` at depth 3. The walk completed in roughly 5.7 seconds with average step confidence near 0.8, and the traversal was an exact hit for the supported query. In effect, the tree served as a coarse-to-fine routing structure: package first, then class, then method, with each step carrying the node label, uncertainty, semantic summary, semantic facts, and raw text before the final localization decision.

1.14 References

1.14.1 Traceability and recovery

- Ali, N., Gueheneuc, Y.-G., & Antoniol, G. (2011). Requirements traceability for object oriented systems by partitioning source code.
- Aponte, J., & Marcus, A. (2011). Improving traceability link recovery using fine-grained requirements-to-code relations.
- Ghabi, A., & Egyed, A. (2012). Exploiting traceability uncertainty between architectural models and code.
- Hey, T., Chen, F., Weigelt, S., & Tichy, W. F. (2021). Improving traceability link recovery using fine-grained requirements-to-code relations.
- Jin, H., Yuan, Y., Wang, B., Wan, H., & Zou, Z. (2025). HGTRTracer: Advancing requirements-code traceability with LLM-augmented attribution reasoning and heterogeneous graph transformer.
- Mahmoud, A., & Niu, N. (2014). Supporting requirements to code traceability through refactoring.
- Marcus, A., & Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing.

- McMillan, C., Poshyvanyk, D., & Revelle, M. (2009). Combining textual and structural analysis of software artifacts for traceability link recovery.
- Omoronyia, I., Sindre, G., Roper, M., Ferguson, J., & Wood, M. (2009). Use case to source code traceability: The developer navigation view point.
- Rahimi, M., & Cleland-Huang, J. (2019). Evolving software trace links between requirements and source code.
- Ramesh, B., Stubbs, C., Powers, T., & Edwards, M. (1997). Requirements traceability: Theory and practice.
- Sannier, N., & Baudry, B. (2012). Toward multilevel textual requirements traceability using model-driven engineering and information retrieval.
- Velasco, A., & Aponte, J. (2020). Automated fine-grained traceability links recovery between high level requirements and source code implementations.
- Zou, Z., Wang, B., Wan, H., Jin, H., Li, X., & Cao, Y. (2024). HANTracer: Leveraging heterogeneous graph attention network for large-scale requirements-code traceability link recovery.

1.14.2 Hierarchical and hyperbolic representations

- Ganea, O.-E., Bécigneul, G., & Hofmann, T. (2018). Hyperbolic neural networks.
- Nickel, M., & Kiela, D. (2017). Poincaré embeddings for learning hierarchical representations.
- Pan, Z., & Wang, P. (2021). Hyperbolic hierarchy-aware knowledge graph embedding for link prediction.
- Zhang, Z., Cai, J., Zhang, Y., & Wang, J. (2020). Learning hierarchy-aware knowledge graph embeddings for link prediction.

1.14.3 State estimation, calibration, and control

- Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. (2017). On calibration of modern neural networks.
- Kalman, R. E. (1960). A new approach to linear filtering and prediction problems.
- Saberi, A., & Sannuti, P. (1988). Observer design for loop transfer recovery and for uncertain dynamical systems.
- Shannon, C. E. (1948). A mathematical theory of communication.

1.14.4 Verification and local constraints

- Dannenberg, R. B., & Ernst, G. W. (1982). Formal program verification using symbolic execution.
- Jaffar, J., Navas, J. A., & Santosa, A. E. (2012). Unbounded symbolic execution for program verification.
- Kouvaros, P. (2023). Towards formal verification of neuro-symbolic multi-agent systems.
- Galitsky, B., & Rybalov, A. (2026). Neuro-symbolic verification for preventing LLM hallucinations in process control.
- Mirhosseini, N., Shojaei, D., & Sabri, S. (2026). From ambiguity to execution: An agentic neuro-symbolic framework for transforming building regulations into deterministic constraints.
- Shiqi, S., Shinde, S., Ramesh, S., Roychoudhury, A., & Saxena, P. (2019). Neuro-symbolic execution: Augmenting symbolic execution with neural constraints.

- Xie, X., Kersting, K., & Neider, D. (2022). Neuro-symbolic verification of deep neural networks.

1.14.5 LLM-assisted requirements and architecture recovery

- Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness.
- ArchCode: Incorporating software requirements in code generation with large language models.
- De Luca, M., Amalfitano, D., Santilli, T., Pelliccione, P., & Fasolino, A. R. (2026). Large language models for software architecture recovery from source code: Class diagrams, patterns, styles, and architecture-as-code views.
- Eisenreich, T., Friedlaender, N., & Wagner, S. (2025). Leveraging large language models for use case model generation from software requirements.
- Erkuş, E. C., Yılmaz, C. Ç., & Demirezen, M. U. (2026). ArkTS code generation: A comprehensive evaluation with large language models.
- Feng, Z., Guo, D., Tang, D., Duan, N., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages.
- Guo, D., Shou, L., Lu, Y., Xu, W., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow.
- Han, H., Kim, J., Yoo, J., Lee, Y., & Hwang, S. (2024). ArchCode: Incorporating software requirements in code generation with large language models.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. R. (2024). SWE-bench: Can language models resolve real-world GitHub issues?
- North, M., Atapour-Abarghouei, A., & Bencomo, N. (2024). Code Gradients: Towards automated traceability of LLM-generated code.