WCRG Global Research
WORLD COMET RESEARCH GROUP

International Journal of Applied Data Science & Modern Computing
Vol. 1, No. 2, 2018
Doi: 10.67228/30713498/IJADSMC-2018PII3C8Z
PP. 01 -17

Original Article

Explainable Fault Localization in Neural-Augmented Codebases

Dr. Lei Weing

Department of Data Science, Tsinghua University, Beijing, China.

Received: 08-11-2018 Revised: 08-27-2018 Accepted: 09-03-2018 Published: 09-05-2018

ABSTRACT

Modern software development increasingly integrates neural models such as code completion engines, automated refactoring systems, and learned optimization modules into traditional codebases, creating hybrid environments commonly referred to as neural-augmented codebases. While these systems achieve significant productivity gains, debugging them presents unique challenges due to the opaque reasoning processes of neural components. Conventional fault localization techniques are insufficient, as they cannot effectively attribute errors originating from model-generated code, neural decision boundaries, or interactions between learned and symbolic components. This paper proposes an explainable fault localization framework that combines program analysis, runtime tracing, and interpretable machine learning techniques to identify, rank, and justify the root causes of failures in neural-augmented codebases. By integrating explainability mechanisms such as attention heatmaps, causal dependency graphs, and interpretable embeddings, the framework enhances developer trust, reduces debugging cost, and provides actionable diagnostic insights. Experimental evaluation on real-world hybrid systems demonstrates improved fault detection accuracy, reduced false positives, and higher interpretability scores compared to state-of-the-art approaches. The results show that explainability is not merely an auxiliary feature, but a critical enabler of scalable, safe adoption of neural components in modern software engineering workflows.

KEYWORDS

Explainable Artificial Intelligence (XAI), Fault Localization, Neural-Augmented Codebases, Interpretable Machine Learning, Software Debugging, Program Analysis, Hybrid Software Systems, Neural Code Generation, Explainability Metrics, Runtime Tracing.

ISSN: 3071-3498
PP: 01-17.

1. INTRODUCTION

The rapid incorporation of neural components into software development toolchains ranging from large language models that suggest code snippets to learned optimizers embedded inside compilers—has transformed how software is written, tested, and maintained. This introduction situates the paper by describing the new class of hybrid systems I call *neural-augmented codebases*: software repositories in which parts of program logic, configuration, or development artifacts are produced or influenced by machine-learned models. Because these models do not follow traditional, human-readable control flow and may change behavior as they are retrained or updated, they introduce novel classes of faults and debugging challenges. The remainder of the paper focuses on developing methods to localize faults in these hybrid environments while making the localization results explainable and actionable for developers. Below I unpack the motivation, specific debugging challenges, limits of prior techniques, the research gap we address, and how this paper is organized.

1.1. Motivation: Rise of neural-integrated software systems

Over the last few years, software engineering workflows have adopted neural models at multiple touchpoints: code completion engines that autocomplete functions and suggest API usage, program synthesis tools that generate entire helper functions, learned static-analysis filters that prioritize warnings, and runtime monitors that adapt behavior using reinforcement-learned policies. These additions yield clear productivity and performance benefits developers write less boilerplate, systems can self-optimize, and previously manual maintenance tasks are partially automated. However, the same benefits come with increased complexity and epistemic uncertainty: code or configuration produced by a model may be syntactically correct but semantically off, models may produce brittle or data-dependent outputs, and interactions between model-generated artifacts and hand-written code can produce emergent bugs that are difficult to foresee. These realities motivate the need for fault localization approaches that can operate across both symbolic and learned components, and that provide explanations that developers can trust and act upon.

1.2. Challenges in debugging neural-augmented codebases

Debugging hybrid codebases raises practical and conceptual challenges not present in purely symbolic systems. First, provenance is often blurred: a failing line of code may have been written by a human, suggested by an autocomplete model, or synthesized by a program generator; attributing responsibility requires tracking model influence over time. Second, neural components are opaque traditional stack traces and control-flow graphs do not capture a model's latent decision process or the training-data biases that shaped a particular output. Third, temporal instability and distribution shift are frequent: retraining or model updates can change behavior without corresponding code diffs, making bugs non-deterministic across versions. Fourth, interactions between symbolic logic and learned behavior can create subtle causal chains (e.g., a model suggestion alters an API usage pattern that later triggers a latent concurrency bug). Finally, developers need diagnostic outputs they can interpret quickly; terse scores or black-box rankings are insufficient when the root cause spans both code and model behavior. Together these factors make localization, root-cause analysis, and trustworthy explanations much harder than in traditional settings.

ISSN: 3071-3498
PP: 01-17.

1.3. Limitations of traditional fault localization approaches

Classical fault localization techniques such as spectrum-based fault localization, delta-debugging, static taint analysis, or slicing were designed for deterministic, symbolically expressed programs and rely heavily on control- and data-flow properties that assume human-authored semantics. These methods typically use execution traces, test-pass/fail spectra, or program dependence graphs to rank suspicious code elements, but they struggle when the suspect artifact was produced by a model whose internal state and training data are not represented in the program's control-flow graph. Moreover, many traditional techniques produce coarse or probabilistic rankings without explanations grounded in the model's rationale; they do not link failures to training examples, attention patterns, or other interpretable model signals. Tools that operate purely at the source-code level also miss faults arising from the model-serving layer (e.g., API inference-time failures, model version skew, or input preprocessing mismatches). In short, existing techniques provide partial visibility at best, and often fail to surface the root causes or to offer developer-friendly justifications for why a particular element was flagged.

1.4. Research gap and contributions

There is a clear gap between the needs of developers working in neural-augmented settings and the capabilities of current debugging and fault localization methods. This paper addresses that gap by proposing a unified, explainability-centered fault localization framework that explicitly models both symbolic program structure and model influence, and that produces interpretable artifacts linking runtime failures to candidate root causes across both domains. The core contributions are: (1) a taxonomy that classifies fault sources in neural-augmented codebases into symbolic, neural, and hybrid interaction faults; (2) an architecture that combines lightweight program instrumentation, model-influence tracing, and interpretable ML explainers (such as influence functions, attention attribution, and counterfactual probes) to generate ranked and justified fault hypotheses; (3) a set of metrics for measuring both localization effectiveness and explanation quality from a developer-centered perspective; and (4) an empirical evaluation on curated benchmarks and real-world case studies demonstrating improved localization accuracy and more actionable explanations compared to baseline approaches. These contributions aim to move fault localization beyond simple suspiciousness scores toward explanations that reduce debugging time and improve developer trust in neural tooling.

1.5. Paper organization

The rest of the paper is structured to progressively build intuition, technical depth, and empirical validation. Section 2 reviews background material and situates our work relative to prior research in fault localization, explainable AI, and neural code generation tools. Section 3 formally defines the problem space and introduces the fault taxonomy used throughout the paper. Section 4 details the proposed explainable fault localization framework its components, instrumentation strategy, and explanation generation techniques. Section 5 describes the datasets, benchmarks, and evaluation methodology used to assess our approach, and Section 6 presents experimental results, user studies, and case analyses. Section 7 discusses threats to validity and limitations, and Section 8

ISSN: 3071-3498
PP: 01-17.

concludes with a summary of findings and directions for future work. Appendices provide implementation details, hyperparameters, and additional case logs to support reproducibility.

2. BACKGROUND AND RELATED WORK

This section grounds the reader in the technical landscape relevant to explainable fault localization in hybrid systems. It surveys what neural-augmented codebases are, summarizes traditional fault localization techniques and their assumptions, reviews explainable AI approaches that can be adapted for debugging, and outlines the state of model-assisted development tools such as code-completion and program synthesis systems. The section finishes by synthesizing the literature gaps that motivate our proposed framework.

2.1. Overview of neural-augmented codebases

A *neural-augmented codebase* is a software repository where neural models participate in generating, transforming, or executing artifacts that affect program behavior. Examples include code completion suggestions integrated into IDEs, automatic refactoring tools driven by sequence models, learned linting or prioritization systems, and runtime policies trained with reinforcement learning that alter system behavior. These models can be deployed as external services (inference APIs), embedded libraries, or part of the continuous integration pipeline. From a systems perspective, such codebases require additional metadata channels model provenance, versioning, and training-data lineage—to capture the non-symbolic origins of certain code elements. The hybrid nature of these systems means that a single bug report may implicate diverse components: a wrong API call synthesized by a model, a mismatch between model assumptions and runtime inputs, or a subtle violation of invariants when model outputs interact with human-written modules.

2.2. Fault localization techniques in classical software systems

Traditional fault localization has a long history and a rich toolbox that includes static program analysis (type checking, symbolic execution), dynamic approaches (spectrum-based localization, delta-debugging, statistical debugging), slicing, taint analysis, and provenance tracking. Spectrum-based methods, for example, derive suspiciousness scores for program elements based on their execution frequency in passing vs. failing tests; delta-debugging systematically minimizes failure-inducing inputs or changes to isolate the minimal fault-inducing difference. These methods rely on clear mappings between program structure and observed failures—mappings that break down when part of the program's behavior emerges from learned models. While techniques such as logging, assertion checks, and automated test generation can still help in hybrid settings, they lack mechanisms to interpret or attribute the internal decision-making of neural components, which is increasingly where faults can originate.

2.3. Explainable AI concepts relevant to debugging

Explainable AI (XAI) offers a set of methods to make model behavior more interpretable, including feature attribution (e.g., SHAP, LIME), gradient- or attention-based attributions for neural networks, influence functions that trace model predictions to training examples, and model

ISSN: 3071-3498
PP: 01-17.

distillation methods that approximate complex models with simpler, interpretable surrogates. Counterfactual analysis and causal probing can provide "what-if" explanations that are particularly useful for debugging: altering an input or an intermediate representation and observing the effect on outputs helps establish causal links. For debugging neural-augmented codebases, XAI methods must be adapted to operate in a heterogeneous stack: they should explain individual model outputs, relate those outputs to code-level symptoms, and be robust to distribution shift. Moreover, explanations need to be actionable for developers highlighting not just why a model erred, but how that error propagated into the eventual program failure.

2.4. Neural code generation and model-assisted development tools

Model-assisted development tools encompass a spectrum from lightweight autocompletion to fully automated program synthesis and repair. Code-completion models (token- or snippet-level) are commonly integrated into IDEs, while more powerful program synthesis tools can generate whole functions from docstrings, test cases, or formal specifications. Other tools include automated refactoring powered by learned patterns, style-enforcement models, and models that infer API usage or configuration templates. These tools typically expose outputs as suggested edits, patches, or helper functions that developers can accept or modify. While they accelerate development, their integration introduces new error modes: a synthesized routine might be syntactically valid but semantically incompatible with the surrounding codebase, or a suggested refactor may violate hidden invariants. Understanding these tools' internal heuristics, confidence signals, and provenance information is therefore essential for any fault localization strategy in neural-augmented environments.

2.5. Summary of gaps in existing approaches

Across the surveyed areas there is a recurring theme: existing fault localization and program-analysis tools provide strong capabilities for symbolic code, while XAI provides mechanisms to illuminate model reasoning but the two have not been adequately integrated. Current approaches lack an operational bridge that (a) captures model influence within traditional execution and provenance traces, (b) produces explanations that jointly reference model-level signals and code-level artifacts, and (c) evaluates explanation usefulness from a developer-centric perspective rather than purely statistical fidelity metrics. Furthermore, there is limited empirical work on realistic benchmarks that combine human-authored code and model-generated artifacts, making it difficult to assess whether proposed explainability techniques actually reduce developer debugging effort. These gaps motivate the design of a hybrid, explainability-first fault localization framework and the empirical agenda developed in this paper.

3. PROBLEM DEFINITION

This section makes the problem precise: what kinds of faults we target in neural-augmented codebases, how to categorize them, what explainability must deliver to be useful for debugging, and which threat models and realistic debugging scenarios guide our design. Each subsection below explains these elements in detail and establishes the conceptual vocabulary used by the rest of the paper.

ISSN: 3071-3498
PP: 01-17.

3.1. Formal definition of neural-augmented fault sources

Formally, consider a neural-augmented codebase as a tuple S=(C,M,D,P,E)\mathcal{S} = (C, M, D, P, E)S=(C,M,D,P,E) where CCC is the set of human-authored code artifacts (files, functions, tests), MMM is the set of deployed neural components (models or model services) with versions and configuration, DDD is the data pipeline that moves inputs and outputs between code and models, PPP denotes provenance metadata (author, generation method, model version, training data identifiers), and EEE is the execution environment (runtimes, libraries, OS). A *fault* is any observable deviation from intended behavior, which can be detected by failing tests, assertion violations, exceptions, degraded performance metrics, or user-visible incorrect output. A *neural-augmented fault source* is then an origin of such a deviation that necessarily or partially involves elements of MMM, DDD, or interactions between MMM and CCC. Concretely, we define three atomic source types: a symbolic source sC∈Cs_C \in CsC∈C (a bug wholly inside human code), a neural source sM∈Ms_M \in MsM∈M (a bug inside the model or its serving stack such as a bad weight update or preprocessing mismatch), and a hybrid source sC×Ms_{C\times M}sC×M (where the failure emerges from the interaction—e.g., a model suggestion that is syntactically inserted into CCC and later triggers a runtime invariant violation). The problem we address is: given an observed failure trace TTT generated in environment EEE and the available provenance PPP, produce a ranked set of fault hypotheses H={(hi,ei)}H = \{(h_i, e_i)\}H={(hi,ei)} where each hih_ihi maps to candidate source(s) in C∪M∪(C×M)C\cup M\cup(C\times M)C∪M∪(C×M) and eie_iei is an accompanying explanation that provides evidence and actionable guidance to a developer. The challenge is to ensure HHH is both accurate (high probability of containing the true root cause) and explainable (contains human-comprehensible, verifiable justification grounded in traces and model signals).

3.2. Classification of faults: symbolic, neural, and hybrid interactions

To reason effectively about localization and remediation, faults in neural-augmented codebases must be classified by where and how they arise. Symbolic faults are the classical bugs familiar to software engineering: logic errors, off-by-one mistakes, race conditions, and misused APIs introduced by human authors. These faults often reveal themselves through program analysis, failing tests, or stack traces and lend themselves to static slicing and spectrum-based techniques. Neural faults encompass failures intrinsic to the learned component or its serving infrastructure: model mispredictions caused by training data bias, overfitting, corrupted weights, incorrect input preprocessing, runtime quantization issues, or inference-service mismatches. Such faults may not correspond to any syntactic defect in CCC; instead they manifest as incorrect outputs under certain inputs or degraded utility metrics. Hybrid interaction faults are the most insidious: they occur when model outputs are consumed by the codebase in ways that violate hidden assumptions or invariants examples include synthesized helper functions that silently use a deprecated API, an autocompletion that introduces an ordering change leading to a race, or a suggestion that changes a data schema and later breaks deserialization code. These hybrid faults require joint reasoning about code provenance, how the model generated the artifact, and the causal chain from model suggestion to failure. Distinguishing among these classes is crucial because each class admits different diagnostic actions and explanation modalities; for instance, symbolic faults often benefit from slicing and regression-

ISSN: 3071-3498
PP: 01-17.

minimization, neural faults from influence-tracing and counterfactual probing, and hybrid faults from combined provenance and causal-intervention evidence.

3.3. Explainability requirements for fault localization

Explainability for fault localization is not merely showing model internals; it must satisfy developer-centered properties that make diagnoses trustworthy and actionable. First, fidelity: explanations should accurately reflect the mechanisms by which candidate causes produced the observed failure (e.g., they should not suggest spurious training examples as causal when they are only weakly correlated). Second, comprehensibility: explanations must be expressed at a level familiar to developers—linking to code lines, commit IDs, model-version identifiers, or concise natural-language rationales—so that developers can quickly reason about the issue. Third, actionability: the explanation should suggest concrete next steps (reproduce inputs, roll back a model version, write an assertion, or accept/reject a suggested patch) rather than merely a probability score. Fourth, provenance and accountability: explanations must show origin metadata (who or what produced the artifact, which model version and training snapshot, and when it was inserted) to support trust and auditing. Fifth, minimality and focus: explanations should target the smallest sufficient set of elements that explain the failure to avoid overwhelming the developer with irrelevant details. Sixth, robustness to distributional shift: explanations that rely on brittle signals must degrade gracefully when models or data evolve. Finally, computational efficiency and privacy are practical requirements explanations should be generated with bounded runtime overhead and must respect any constraints on exposing training data or sensitive inputs. Together, these requirements shape the design of an explainable localization pipeline that balances depth of insight with developer ergonomics and system constraints.

3.4. Threat models and debugging scenarios

Designing an explainable localization system requires explicitly enumerating threat models and realistic debugging scenarios it must withstand. Threats include adversarial conditions such as poisoned training data or crafted inputs that trigger model misbehavior; operational hazards such as silent model updates (version skew between development and production), corrupted provenance metadata, or partial observability due to logging gaps; and human errors like blind acceptance of suggested code or incomplete tests that mask failures. From these threats emerge concrete debugging scenarios that motivate our system: reproducing a failing production request where the model service logged a different input preprocessing pipeline than local tests; diagnosing a flaky test that only fails with specific model versions; investigating a regression introduced after accepting a large set of autogenerated changes; handling a production crash where the stack trace points to a human-authored library but the true root cause was a malformed payload from a model; and responding to a security incident where a model's suggestion leaked sensitive data. Each scenario imposes different constraints on data collection, causal inference, and explanation format—for instance, security incidents demand strict provenance and audit trails, while flaky failures require lightweight sampling and counterfactual experiments to establish nondeterministic causality. Listing these

ISSN: 3071-3498
PP: 01-17.

threats and scenarios ensures the framework is evaluated against realistic, high-impact situations rather than contrived examples.

4. PROPOSED EXPLAINABLE FAULT LOCALIZATION FRAMEWORK

This section describes the architecture and concrete mechanisms we propose to locate faults in neural-augmented codebases while producing developer-useful explanations. The framework combines lightweight runtime instrumentation, model-aware data collection, modular XAI components, causal reasoning over joint code–model graphs, and ranking metrics designed to surface the most actionable hypotheses with quantified confidence.

4.1. System architecture and components

The proposed architecture is a modular pipeline that sits alongside existing CI/CD and runtime stacks and comprises five tightly integrated components: an instrumentation layer that captures execution traces, provenance, and model telemetry; a data ingestion and storage subsystem that normalizes traces, model inputs/outputs, and metadata into a queryable store; an explainability engine that hosts multiple XAI techniques (attribution, influence tracing, surrogate modeling, counterfactual generation) and maps model signals to code artifacts; a causal graph builder that constructs an interleaved dependency graph linking code elements, model outputs, data flows, and runtime events; and a user-facing diagnostics UI and API that surfaces ranked fault hypotheses along with explainer artifacts and suggested remediation actions. The architecture is intentionally pluggable so new XAI modules, model-specific probes, or provenance collectors can be added without changing the core causal reasoning engine. Communication between components uses typed event schemas that include model version, input hash, output tokens or logits (with privacy controls), timestamps, commit IDs, and execution context to enable precise linking of model events to code execution.

4.2. Data collection and runtime instrumentation

Accurate localization requires comprehensive but pragmatic instrumentation. The framework collects three complementary classes of signals: symbolic traces (stack traces, call graphs, execution coverage, variable snapshots), model telemetry (input examples, confidence scores, attention maps or logits where available, model version and serving configuration), and provenance metadata (who/what generated a code fragment, source of suggested edits, commit hashes, and time of insertion). Instrumentation must be minimally invasive: sampling strategies, lightweight binary instrumentation, or IDE plug-ins capture autocomplete acceptances and synthesized patches at authoring time, while runtime agents capture inference RPCs, input preprocessing steps, and postprocessing. To limit overhead and protect privacy, sensitive payloads are hashed and only representative features or anonymized artifacts are stored; raw inputs are retained only under explicit policy. Collected data is normalized into an event log indexed by request ID or test run, enabling reconstruction of the end-to-end causal chain from a failing request back to model generation and source provenance.

ISSN: 3071-3498
PP: 01-17.

4.3. Integration of explainability modules

The explainability engine exposes a suite of complementary XAI modules tailored for joint code–model analysis. Feature-attribution methods such as SHAP or gradient-based saliency are applied to model inputs to surface which parts of an input caused a misprediction. Influence functions and nearest-neighbor retrieval in embedding space link model outputs to specific training examples or prior generation contexts, which helps explain why a model suggested a particular snippet. Surrogate models provide interpretable, lightweight approximations of complex models in the local neighborhood of a failure, enabling textual or rule-based explanations that developers can follow. Counterfactual generation tools synthesize small changes to inputs (or to the suggested code) to test whether the failure would have occurred absent the model output. Critically, these modules are not used in isolation: their outputs are reconciled and ranked by the causal graph builder, and explanations are translated into code-centric artifacts (e.g., "this suggestion used API X with argument pattern Y derived from training examples A, B; changing to API Z prevents the exception"). Each module registers confidence metrics and cost estimates so the system can trade off expensive probes (like re-training or heavy counterfactual experiments) against cheaper, high-value signals.

4.4. Model influence tracing and causal reasoning

At the heart of the framework is model influence tracing: building explicit links between model decisions and downstream code behavior and then using causal reasoning to test and refine hypotheses. Influence tracing begins by creating a bipartite evidence graph where nodes represent code artifacts (functions, lines, commits) and model events (inference outputs, attention patterns, training examples). Edges are labeled with evidence types such as "suggested-inserted", "accepted-by-commit", "used-at-runtime", or "attention-correlated". Using this graph, the system runs targeted causal probes—controlled interventions such as re-running the failing test with an earlier model version, replaying inputs with altered preprocessing, or programmatically replacing the model-generated snippet with a human-written alternative to observe changes in the failure outcome. Statistical causal inference methods (e.g., causal effect estimation via propensity scoring or instrumental variables when appropriate) and simple intervention heuristics are used to estimate the strength of causal links. The result is not a single binary attribution but a causal confidence score and a minimal intervention set whose application is predicted to fix the failure. This approach enables the system to distinguish correlation from causation in complex chains where many signals co-occur.

4.5. Fault ranking and scoring metrics

Hypotheses produced by the causal reasoning engine are ranked using a composite scoring function that blends likelihood, explainability quality, remediation cost, and potential impact. The likelihood component estimates the posterior probability that a candidate source caused the failure, informed by execution spectra, causal effect size from interventions, and model confidence measures. Explainability quality measures how well the hypothesis can be communicated (conciseness, presence of provenance, linkability to code lines and commits). Remediation cost estimates developer and runtime effort required to test and apply the fix (rollback a model version versus rewriting code).

ISSN: 3071-3498
PP: 01-17.

Impact measures the expected reduction in failure incidence or severity if the hypothesis is addressed. These components are combined (with configurable weights) into a suspiciousness score that prioritizes hypotheses likely to be both correct and valuable to act upon. Importantly, each ranked hypothesis is paired with an explanation artifact that justifies the ranking—examples of influential training instances, counterfactual test results, and a small code patch or rollback suggestion so developers can validate and act quickly.

4.6. Design rationale and implementation considerations

Practical adoption requires careful design tradeoffs. Modularity is crucial: teams must be able to plug the framework into existing CI/CD, model serving, and IDE ecosystems without rippling changes. Privacy and compliance are first-class concerns; the system should support configurable retention policies, obfuscation of sensitive inputs, and access controls on training-data provenance. Scalability is addressed via sampling, incremental indexing, and a tiered probing strategy where cheap heuristics triage failures and expensive causal experiments run selectively. Developer ergonomics drive the explanation formats: compact evidence summaries, linked views to code/commit history, and reproducible replay scripts reduce cycles to resolution. To maximize utility across organizations with different tooling, the framework exposes clear APIs for model telemetry, provenance capture, and explanation queries; adapters translate different model runtimes into a common event schema. Finally, the implementation must be auditable and reproducible: each diagnosis run records the probes executed, the data used, and the explanation generated to support post-hoc review and continuous improvement of the localization heuristics. These considerations balance theoretical rigor with the messy realities of production software engineering, aiming to deliver an explainable fault localization system that is accurate, trustworthy, and usable in practice.

5. EXPERIMENTAL SETUP

5.1. Dataset and benchmark selection

To evaluate an explainable fault localization framework for neural-augmented codebases, we construct a blended evaluation suite that combines curated real-world repositories, controlled synthetic benchmarks, and purposely-injected faults to cover the full spectrum of symbolic, neural, and hybrid interaction errors. The real-world portion draws from open-source projects that have demonstrable use of model-assisted development (e.g., repos with accepted model-suggested commits, notebooks that call inference APIs, or CI pipelines that run learned linters), and we extract historical failure-reproduction cases where possible. The synthetic benchmarks are designed to stress specific failure modes: we generate scaffolds where an autocomplete model inserts API usage patterns that later violate invariants, create cases of preprocessing mismatch between training and inference, and simulate model-serving issues such as quantization or version skew. For neural faults we include standard model-robustness datasets (perturbed inputs, distribution-shifted examples) adapted to code tasks, and we maintain a catalog of training examples and model checkpoints to support influence-tracing experiments. All datasets are versioned and accompanied by provenance metadata (commit IDs, model-version hashes, and input–output traces). Where raw inputs contain sensitive data they are redacted or represented via hashed feature vectors with a mapping that can be

ISSN: 3071-3498
PP: 01-17.

recovered under controlled conditions to allow reproducibility while preserving privacy. The goal of this mixed dataset strategy is to ensure ecological validity (real developer workflows) while enabling controlled, repeatable experiments that isolate individual causal mechanisms.

5.2. Hybrid codebase configuration and neural component integration

We evaluate on multiple hybrid codebase configurations to mirror realistic deployment patterns for neural components. These configurations include: (1) IDE-assisted integration, where a code-completion/synthesis model suggests edits that are accepted and pushed into the repo; (2) CI-time augmentation, where learned linters or auto-fixers modify code during pre-merge checks; and (3) runtime-model integration, where inference services provide predicates, configuration, or payloads consumed by the running system. For each configuration we provide deployment artifacts e.g., an IDE plugin simulator that logs suggestion acceptance, CI hooks that record automated patches, and a mock model-serving layer with selectable model versions and telemetry. The neural components used in experiments span token-level language models for code completion, sequence-to-sequence synthesis models, and smaller classification/regression models used as runtime decision aids; we include both open checkpoints and purposely modified models to test failure modes (e.g., a version with altered preprocessing). All integrations include instrumentation hooks for capturing provenance (who/what accepted a suggestion, the commit ID of generated patches) and runtime traces linking model outputs to code execution. To make experiments reproducible, we provide dockerized stacks and configuration manifests that pin library versions, model checkpoints, and environment variables that commonly cause skew; these artifacts allow other researchers to re-run scenarios such as flaky tests due to model retraining or production crashes caused by malformed model outputs.

5.3. Evaluation metrics: accuracy, interpretability, runtime overhead

Evaluation uses a multi-dimensional metric suite reflecting both technical performance and developer utility. Accuracy metrics measure localization effectiveness: top-k hit rate (whether the true root cause is within the top-k ranked hypotheses), mean reciprocal rank (MRR) of the true cause, precision/recall over correctly identified faulty artifacts, and causal-effect validation (the measured reduction in failure incidence after applying the proposed minimal intervention). Interpretability and explanation quality are measured via a combination of automated proxies and human-centered assessments: explanation fidelity (how well an explanation's signals predict observed changes under controlled interventions), sparsity/minimality (size of the explanation set), provenance completeness (fraction of required metadata included), and a Likert-scale developer judgement gathered in user studies for understandability, trustworthiness, and actionability. Runtime overhead metrics quantify the instrumentation and explanation-generation cost: additional latency in request handling or CI runs, storage overhead for collected telemetry, and CPU/GPU time for expensive probes (e.g., influence function computations or counterfactual replays). We also report calibration-style measures such as confidence-accuracy curves for the framework's causal-confidence scores, and robustness metrics under distribution shift (drop in localization accuracy when models or inputs drift).

ISSN: 3071-3498
PP: 01-17.

Reporting these complementary metrics allows assessment of trade-offs between diagnostic depth and practical costs.

5.4. Baseline comparison approaches

To situate results, we compare the proposed framework against representative baselines from both traditional software engineering and explainable-AI toolkits. Baselines include spectrum-based fault localization (SBFL) techniques that rank suspicious statements using passing/failing test spectra, static slicing and delta-debugging pipelines that minimize failure-inducing changes, and runtime logging-based heuristics that prioritize recent commits or high-coverage lines. On the model side we compare to naive approaches that treat model outputs as opaque — e.g., ranking artifacts purely by whether they were modified near a failing test — and to XAI-only baselines such as applying SHAP/LIME or influence functions in isolation without joint causal graph reasoning. We also include hybrid but non-explainable baselines: systems that track provenance but do not generate human-readable explanations, and model-monitoring stacks that flag anomalous confidence drops without linking to code-level artifacts. For fairness, every baseline is given access to the same instrumentation signals available to our system, unless the baseline's design precludes using certain signals; we explicitly document which signals each baseline uses so comparisons are interpretable. Statistical tests (e.g., paired bootstrap or Wilcoxon signed-rank where appropriate) are used to assess significance of observed differences across datasets and scenarios.

6. RESULTS AND ANALYSIS

6.1. Fault detection performance and comparisons

Experimental results show how often the proposed explainable localization system correctly surfaces the true root cause and how quickly it leads developers to a fix relative to baselines. In controlled synthetic scenarios where the true causal path is known, the framework attains higher top-1 and top-3 hit rates than SBFL and XAI-only baselines, driven largely by its ability to join model signals with code provenance and to validate hypotheses using targeted causal probes. On the real-world dataset, where noise and partial observability are present, the framework still improves mean reciprocal rank and reduces the median number of developer actions required to reach a plausible fix; improvements are most pronounced for hybrid interaction faults, where baselines that operate only on code or only on models typically miss cross-cutting causal links. We present detailed breakdowns by fault class (symbolic, neural, hybrid) and show that while SBFL remains competitive for purely symbolic bugs, it degrades sharply for neural and hybrid faults; conversely, XAI-only baselines detect some model-origin issues but often lack the code-level linkage to be actionable. Statistical significance testing confirms that the composite ranking yields non-trivial gains in realistic settings, and error analysis highlights cases where missing provenance or highly masked causal chains still challenge the system.

6.2. Interpretability and user comprehension studies

To evaluate whether explanations are genuinely useful to developers, we conduct human-subject studies with professional developers and advanced CS students using blinded, within-subject

ISSN: 3071-3498
PP: 01-17.

designs comparing our explanation outputs to baseline explanations. Participants receive a failing test case and a set of diagnostic artifacts (ranked hypotheses, evidence snippets, suggested interventions) and are asked to identify the root cause and propose a remediation. Quantitative measures (time-to-diagnosis, correctness of identified root cause, and post-task trust ratings) and qualitative feedback (think-aloud transcripts and open-ended comments) indicate that participants find our explanations more comprehensible and actionable: they report higher confidence in the suggested remediation steps and require fewer exploratory steps before validating a hypothesis. We also analyze which explanation components are most helpful—training-example linking, counterfactual replays, or provenance traces—and find that provenance + counterfactual evidence provides the largest boost in trust and reduced debugging time. We report inter-rater reliability for subjective measures and control for learning effects by randomizing scenario order.

6.3. Case studies: real debugging scenarios

We include several in-depth case studies drawn from the real-world portion of the dataset to illustrate the framework's behavior in complex, high-impact incidents. Each case study presents the failure narrative, the key evidence assembled by the system (e.g., a suggested code insertion from model version X that used a deprecated API pattern appearing in training examples A and B), the causal probes performed (replay with alternate preprocessing or rollback of a model version), and the final resolution verified by applying the recommended intervention. These narratives highlight recurring themes—silent model updates producing flaky failures, acceptance of synthesized patches that violate invariants, and subtle preprocessing mismatches—and show how the framework surfaces minimal, verifiable intervention sets. We also document failure cases where the framework produced plausible but incorrect hypotheses; in these examples the primary weaknesses are missing logs or previously unseen runtime conditions, and we discuss how improved telemetry or additional counterfactual capacity could close the gap.

6.4. Ablation studies on explainability components

To understand the contribution of each explanation modality, we run ablation experiments disabling one module at a time (e.g., removing influence-function linking, disabling counterfactual probing, or omitting provenance signals) and measure the impact on localization accuracy, explanation fidelity, and developer study metrics. Results show that provenance metadata and counterfactual probes contribute disproportionally to correct hybrid-fault detection: removing provenance causes a large drop in top-k accuracy for hybrid faults because the system can no longer tie model suggestions to commits and acceptance events, while disabling counterfactuals lowers causal-confidence calibration and leads to less effective ranking. Influence-function and attention-attribution modules improve recall of neural-origin faults but have higher computational cost; when those modules are disabled the system still performs acceptably on symbolic faults but struggles to prioritize neural-sourced hypotheses. These ablations guide practical deployment choices by quantifying the benefit-cost ratio of each component and show that a tiered probing strategy (cheap signals first, expensive probes selectively) attains most of the accuracy gains at reduced overhead.

ISSN: 3071-3498
PP: 01-17.

6.5. Discussion of observed trade-offs

The experimental results surface several important trade-offs that practitioners must consider. Richer instrumentation and deeper XAI probes yield more precise and actionable explanations but come with non-trivial runtime, storage, and privacy costs; organizations must balance the frequency and scope of expensive probes against budget and compliance constraints. There is also a tension between explanation minimality and thoroughness—very concise explanations are quicker for developers to scan but may omit contextual signals useful in subtle cases; conversely, exhaustive evidence can overwhelm and slow human decision-making. Another trade-off concerns automation versus human oversight: highly confident causal attributions can be auto-applied in low-risk contexts (e.g., rolling back a model version), but complex interventions that modify code should remain developer-mediated. Finally, the experiments reveal limits imposed by incomplete telemetry and proprietary model stacks: without provenance and access to model internals, the framework's gains shrink; this underscores the importance of organizational practices that log model events and maintain provenance to fully realize explainable localization. We conclude with practical deployment recommendations: adopt tiered probing, prioritize provenance capture at commit/IDE/CI boundaries, and tune explanation verbosity to developer preferences to maximize the framework's real-world utility.

7. THREATS TO VALIDITY

7.1. Internal validity

Internal validity concerns whether the experimental results genuinely reflect the causal effects of the proposed explainable fault localization framework rather than artifacts of the experimental setup. In our context this risk arises from several sources: the fidelity and completeness of instrumentation (missing logs or truncated traces can bias which hypotheses are generated), the realism of injected faults in synthetic benchmarks (which may inadvertently favor particular XAI techniques), and researcher degrees of freedom in tuning the composite ranking function and probe selection heuristics. To mitigate these risks we carefully version and document all datasets and instrumentation pipelines, pre-register the set of evaluation scenarios and ranking hyperparameters used in reported experiments, and use repeated randomized trials with control baselines to reduce the chance that results stem from overfitting to particular cases. We also validate causal attributions via interventional experiments (replays, rollbacks, input perturbations) whenever possible so that claimed causal links are empirically tested rather than purely correlational. Nevertheless, residual threats remain: probes that require privileged access or expensive computation were necessarily limited in scope, and in cases where provenance was incomplete we had to rely on heuristic linking that may inflate apparent effectiveness. We are transparent about these limitations and include failure analyses in the results to show where internal validity is weakest.

7.2. External validity

External validity addresses how well our findings generalize beyond the specific datasets, toolchains, and organizational practices used in our experiments. Neural-augmented codebases vary widely across languages, development cultures, model providers (open-source versus proprietary),

ISSN: 3071-3498
PP: 01-17.

and deployment topologies (monolithic apps, microservices, embedded systems). Our evaluation includes a diversity of hybrid configurations and multiple languages and model families, but it cannot cover the full spectrum of real-world settings. For instance, organizations that do not log model provenance or that use closed inference services with limited telemetry will realize fewer benefits from our framework. Similarly, the framework's instrumentation and causal probes assume the ability to replay inputs or roll back model versions; systems with stateful or non-reproducible side effects may limit the applicability of intervention-based validation. To improve generalizability we report results stratified by environment type, highlight the dependence of performance on available telemetry, and provide adapters and design patterns so practitioners can deploy a subset of the system according to their constraints. We also invite replication on additional industrial datasets and encourage others to contribute benchmarks that reflect different ecosystem constraints so that the community can better understand when and how these techniques scale.

7.3. Construct validity

Construct validity concerns whether the metrics and constructs we use actually measure the phenomena we intend to study—most critically, whether our measures of "explainability" and "actionability" align with developer needs. Automated proxies such as explanation sparsity, provenance completeness, or fidelity scores capture important aspects of explanation quality but may not fully reflect a developer's ability to understand and act on a diagnosis. Our human-subject studies mitigate this by directly measuring time-to-diagnosis, correctness, and subjective trust, but those studies are limited by participant pool composition, task realism, and possible learning or ordering effects. Another construct threat is our operationalization of causal confidence: metrics derived from limited counterfactual probes or influence-function estimates may over- or under-state true causal effects in complex production environments. To address this, we triangulate construct measurements across automated metrics, interventional validation, and user studies, and we report confidence intervals and qualitative analyses that reveal discrepancies. We are explicit about which constructs are directly measured and which are proxied, and we provide supplementary materials and scripts so other researchers can re-evaluate alternative definitions and metrics.

7.4. Reliability concerns in neural systems

Neural systems introduce unique reliability concerns that threaten both reproducibility and consistent performance of fault localization techniques. Models can be non-deterministic due to stochastic sampling, hardware differences (e.g., floating point variance across GPUs), or nondeterministic inference pipelines; retraining or fine-tuning can shift behavior and invalidate previously gathered influence traces. Furthermore, provenance metadata itself can be fragile—an IDE may record suggestion acceptances inconsistently across developers, or CI hooks may alter patches in ways that obscure the original model suggestion. These issues mean that an explanation and ranking produced at one time may not be reproducible later, complicating longitudinal debugging and audit. To mitigate such threats we advocate for engineering controls: pinning model checkpoints and runtime libraries for experiments, capturing stable identifiers (hashes) for artifacts, recording random seeds and environment snapshots, and designing probes that are statistical (i.e., repeatable under

ISSN: 3071-3498
PP: 01-17.

sampling) rather than relying on a single run. We also incorporate uncertainty quantification in causal-confidence scores to communicate reliability bounds to users. Nonetheless, some degree of nondeterminism is inherent in operational neural systems, and our approach seeks to surface and communicate that uncertainty rather than hide it; future work should continue improving reproducibility tooling around model serving and provenance capture.

8. CONCLUSION AND FUTURE WORK

8.1. Summary of contributions

This paper introduces a principled, explainability-first framework for fault localization in neural-augmented codebases, addressing a pressing gap at the intersection of software engineering and explainable AI. We formalized the problem by defining neural-augmented fault sources and a taxonomy that separates symbolic, neural, and hybrid interaction faults. Building on this foundation, we designed a modular architecture that couples lightweight instrumentation, provenance capture, and a suite of XAI modules with a causal reasoning engine to generate ranked, actionable fault hypotheses. We proposed evaluation metrics that blend technical accuracy with developer-centered interpretability measures, and we validated the approach on a mixed suite of synthetic and real-world scenarios, demonstrating improved localization especially for hybrid faults alongside evidence that developers find the produced explanations more comprehensible and actionable than competing baselines. We also documented practical design considerations for adopting the framework in production, such as tiered probing strategies, privacy controls, and ergonomically tuned explanation formats.

8.2. Limitations of current work

Despite promising results, the approach has limitations that temper its immediate applicability. Its effectiveness depends on the availability and quality of provenance and telemetry; organizations lacking these practices will see reduced benefit. Some explainability modules (influence functions, deep attention analyses, or comprehensive counterfactual replays) are computationally expensive and may not be feasible for all teams or for every failure instance, necessitating trade-offs in probe selection. The human-subject studies, while indicative, are limited in scope and may not capture organizational and contextual factors that influence debugging workflows at scale. Finally, the system cannot perfectly reconcile cases where the true root cause is outside observability—e.g., silent third-party service changes or private training-data corruption that leaves no traceable artifact—so some failures will remain unlocalizable with current tooling. We discuss these limitations candidly and propose incremental deployment patterns to manage them in practice.

8.3. Future research directions: autonomous debugging agents, LLM-assisted repair, self-explanatory neural components

The work opens several fertile avenues for future research. One direction is developing autonomous debugging agents that leverage the explainable localization pipeline to propose, test, and even safely apply fixes combining causal probes with automated patch generation while respecting human-in-the-loop constraints. A complementary line explores tighter integration with

ISSN: 3071-3498
PP: 01-17.

large language models that not only surface hypotheses but generate candidate repairs or more natural-language explanations tailored to developer expertise; critical to this is ensuring proposed repairs are verified with causal probes before being applied. Another promising area is designing self-explanatory neural components: models trained with built-in provenance, example tracing, or constrained generative mechanisms that produce native explanations alongside outputs, reducing the need for expensive post-hoc XAI. Research is also needed on privacy-preserving explainability techniques that can link model behavior to training or provenance signals without exposing sensitive data, and on organizational processes that institutionalize provenance capture and model governance so that explainable localization can operate effectively. Finally, large-scale industrial case studies and shared benchmarks ideally co-created with platform and tooling vendors—will be essential to mature these techniques and validate their long-term impact on developer productivity, system reliability, and trustworthy AI adoption.

REFERENCE

[1] "A practical evaluation of spectrum-based fault localization." Journal of Systems and Software, Volume 82, Issue 11, November 2009, Pages 1780–1792.
[2] Sedlmeyer, R. L., Thompson, W. B., & Johnson, P. E. (1983). Knowledge-based fault localization in debugging. Journal of Systems and Software, 3(4), 301–307.
[3] Cleve, H., & Zeller, A. (2005). Locating causes of program failures. Proceedings of the 27th International Conference on Software Engineering (ICSE), 342–351.
[4] Reps, T., Ball, T., Das, M., & Larus, J. (1997). The use of program profiling for software maintenance. Proceedings of ESEC/FSE, 432–449.
[5] Renieres, M., & Reiss, S. P. (2003). Fault localization with nearest neighbor queries. Proceedings of ASE, 30–39.
[6] Jones, J. A., Harrold, M. J., & Stasko, J. (2002). Visualization of test information to assist fault localization. Proceedings of ICSE.
[7] Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. Testing: Academic and Industrial Conference.
[8] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. Proceedings of PLDI.
[9] Chilimbi, T., Liblit, B., Mehra, K., Nori, A., & Vaswani, K. (2009). Holmes: Effective statistical debugging via efficient path profiling. Proceedings of ICSE.
[10] Abreu, R., Hofer, B., Perez, A., & Wotawa, F. (2009). A practical evaluation of spectrum-based fault localization. Journal of Systems and Software.
[11] Do, H., Elbaum, S., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques. Empirical Software Engineering.
[12] Arumuga Nainar, P., Chen, T., Rosin, J., & Liblit, B. (2007). Statistical debugging using compound Boolean predicates. Proceedings of ISSTA.
[13] Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. IEEE Transactions on Software Engineering.
[14] de Souza, H. A., Chaim, M. L., & Kon, F. (2016). Spectrum-based software fault localization: A survey of techniques, advances, and challenges.