2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Interaction-Aware Patch Assessment for Multi-Fault Automated Program Repair

Omar I. Al-Bataineh
Gran Sasso Science Institute, L'Aquila, Italy
omar.albataineh@gssi.it

***Abstract*—Patch overfitting remains a persistent challenge in automated program repair (APR), especially when validation depends on incomplete test suites. We argue that this problem is significantly exacerbated by the overlooked presence of multiple interacting faults, a common yet under-addressed reality in real-world software. Conventional APR tools typically treat faults in isolation, neglecting subtle interactions that can mask faults or introduce regressions. To address this, we develop a taxonomy of five fault interaction-aware patch assessment strategies, supported by a formal model that identifies when and how each should be applied. Our framework guides robust multi-fault repair and exposes how fault interactions critically influence patch outcomes. To our knowledge, this is the first formal treatment of patch assessment and overfitting in multi-fault settings, offering a foundation for more reliable and practical APR.**

I. Introduction

Automated Program Repair (APR) aims to enhance software reliability by automatically creating patches that fix bugs in code. Usually, a patch is considered plausible if it passes all tests in a validation suite, and valid if it truly fixes the original problem. However, since test suites provide only a partial specification of intended behavior, APR techniques often generate overfitting patches that pass tests but do not genuinely fix the root cause [1], [2]. This overfitting issue remains a major barrier to the practical use of APR, prompting the community to focus on better patch evaluation, prioritization, and filtering.

While patch overfitting has been widely studied in single-fault scenarios [3], [4], [2], [5], [6], its occurrence in multi-fault settings remains less explored. Real-world software often contains multiple faults whose effects can interact, where the presence, manifestation, or repair of one fault influences another's behavior or observability [7]. Existing APR techniques often assume faults are independent, neglecting how such interactions impact patch correctness and generalizability.

As a result, existing tools that fix faults in isolation may produce partial fixes that pass current test suites but do not fully restore program correctness, resulting in multi-fault overfitting. This underscores the importance of rethinking patch evaluation methods with explicit focus on fault interactions, a challenge well acknowledged in debugging research [8], [9], [10], [11].

To address this critical gap, we investigate how fault interactions influence patch assessment in APR. We identify significant limitations in current assessment methods, which mainly depend on the single-fault assumption. Using examples from multi-fault programs, we demonstrate how overlooking fault interactions can cause patch overfitting and false positives. To improve this, we introduce a taxonomy of five assessment strategies: Traditional Independent Assessment, Fault-Aware Assessment, Incremental Assessment, Collaborative Assessment, and Predictive Data-Driven Assessment, each designed to explicitly consider interactions among faults during patch evaluation. These strategies aim to enhance the reliability and precision of APR in complex scenarios, offering a more solid framework for detecting truly effective patches.

II. Fault-Aware Model

Existing research often evaluates patches in isolation, focusing narrowly on their immediate and direct effects. However, in multi-fault scenarios, such assessments can be misleading: a valid patch for one fault may be incorrectly discarded or introduce new issues due to complex interdependencies with another. These interdependencies, manifesting as masking, cascading failures, or unintended synergies, are central to understanding and improving patch evaluation. We refer to them as *fault interactions*, defined as follows.

*Definition 1 (**Fault Interaction**):* Let P be a program containing two distinct faults, f_i and f_j. A *fault interaction* occurs if there exists at least one program execution trace in which the presence, manifestation, or repair of f_i affects the behavior, detectability, or repair outcome of f_j, or vice versa.

Such interactions may manifest as *masking*, *cascading failures*, or *synergistic effects*, and can lead to misleading repair outcomes, causing valid patches to be discarded or faulty ones to be accepted. Explicitly modeling fault interactions is thus crucial for advancing APR in realistic multi-fault settings.

A. Modeling Fault Interactions for Patch Assessment

We define the key conceptual components involved in patch assessment for multi-fault scenarios as follows:

- $F = \{f_1, \ldots, f_n\}$: the set of faults in program P.
- T: a multi-fault test suite that triggers all faults in F.
- $\mathrm{MFL}(P, T)$: a multi-fault localization function that identifies suspicious code locations for each $f_i \in F$.
- $\mathcal{I} \subseteq F \times F$: a relation capturing fault interactions.
- $\mathcal{R}(P, F, T, \mathcal{I})$: a repair function generating patches for faults in F, considering tests T and relevant interactions.
- $\mathcal{A}(\text{patch}, C, \mathcal{I})$: an assessment function that evaluates a patch under criterion C, guided by fault interactions $\mathcal{I}$, and returns either *valid* or *overfitted*.

Each $(f_i, f_j) \in \mathcal{I}$ falls into one of the following types:

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00319

1) *Independence (I)*: Fixing f_i does not affect f_j. Independent faults support isolated patch evaluation.
2) *Cascading (C)*: Fixing f_i alters the manifestation or detection of f_j, which may lead to new test failures.
3) *Synergy (S)*: A fix to f_i partially or fully resolves f_j, suggesting opportunity for joint repair.
4) *Masking (M)*: Fault f_i hides the effects of f_j, potentially causing a patch to be misjudged as correct or incomplete.

Each component plays a vital role in enabling fault-aware assessment. Multi-fault localization (MFL) identifies where faults reside, guiding targeted program patching and detailed interaction mapping. The repair function $\mathcal{R}$ models how patches are generated for faults in F, potentially in isolation, but crucially incorporates knowledge of interactions to avoid generating misleading fixes. The assessment function $\mathcal{A}$ judges patch correctness based on customizable criteria and the interaction context. Finally, the interaction relation $\mathcal{I}$ helps determine whether patches should be evaluated independently or collaboratively, depending on the nature of fault coupling.

B. Modeling Fault Interactions in Practice

While our formal model defines the interaction relation $\mathcal{I}$ abstractly, approximating it in practice requires program analysis. One approach is to analyze control- and data-flow dependencies among suspicious locations identified by MFL(P, T). Static analysis can help identify potential interactions without executing the program—for example, by detecting shared variables or overlapping control paths. Although static techniques are efficient, they often suffer from false positives.

Dynamic techniques offer additional insight by observing the behavior of subpatches during execution. For instance, comparing test coverage, runtime state differences, or error traces across executions can surface evidence of cascading effects, masking, or synergy. While exact interaction classification remains challenging due to inherent program complexity and undecidability, combining static and dynamic heuristics provides a practical foundation for estimating $\mathcal{I}$ and choosing suitable patch assessment strategies.

C. AI Assistance in Estimating Fault Interactions

While tools like ChatGPT [12] cannot analyze program semantics or run code, they can still help developers reason about fault interactions, especially when given the right context. For example, by providing the program P, the test suite T, and suspicious code regions (e.g., from MFL(P, T)), one can ask questions like *"If these two faults show up in tests, could fixing one affect the other?"* or *"Do these faults seem independent, or might they mask each other's effects?"* These queries, while speculative, can guide developers in prioritizing fault pairs for deeper analysis or help explain patch behaviors.

Used in conjunction with static and dynamic techniques, AI tools provide a lightweight, conversational interface for exploring and estimating $\mathcal{I}$ in practice, particularly when navigating large codebases. While not guaranteed to be correct, such AI-generated insights complement formal and empirical analyses and may require expert validation.

The formal model of fault interactions, including the classification of faults as independent, cascading, synergistic, or masking, provides a critical foundation for designing effective patch assessment strategies. This theoretical framework moves beyond the traditional single-fault assumption by highlighting how different interaction types demand different evaluation approaches. In the following section, we introduce a taxonomy of five assessment strategies, each directly motivated by and tailored to handle these specific interaction types, thereby translating our formal model into a practical guide for more robust and reliable patch validation in multi-fault settings.

III. Formalizing Multi-Fault Patch Assessment

Let P be a program with faults $F = \{f_1, f_2, \dots, f_n\}$. Let pt_i denote a patch for fault f_i produced by an APR system. A composite patch is $pt = pt_1 \oplus pt_2 \oplus \cdots \oplus pt_n$, where $\oplus$ denotes patch composition. The assessment method $\mathcal{A}(pt, C_\phi)$ labels a patch as *correct* or *overfitting* according to criterion C_ϕ. By definition, if any pt_i overfits, the composite patch also overfits; it is correct only if all subpatches are correct.

However, current assessment methods [13], [14], [15], [16], [17] typically evaluate the composite patch, or its subcomponents, independently. This approach disregards the influence of other faults in $F \setminus \{f_i\}$. This isolation can obscure critical interactions, leading to inaccurate assessments. A patch might appear to rectify a fault, but another active fault could invalidate the fix or mislead the test suite. Consequently, despite progress in patch assessment techniques, evaluating correctness in multi-fault programs remains challenging, especially when a patch's validity depends on how its components interact.

Why evaluating subpatches matters: In multi-fault repair, assessing only the final composite patch can lead to misleading results. A composite patch may be labeled invalid even if some of its subpatches are valid. For example, a correct fix for one fault might be overshadowed by an overfitting change to another, causing the entire patch to be rejected. To better assess patch quality and avoid discarding partially correct solutions, it is crucial to evaluate subpatches individually or in combinations. The five assessment strategies we propose (see Definition 2) can all be instantiated at the subpatch level, which helps isolate effective repairs and build more reliable repair pipelines.

Definition 2: (**Assessment strategies**). A patch assessment strategy $\mathcal{A}$ can be instantiated in five forms, each offering a specific way to evaluate patches in multi-fault scenarios:

1) *Traditional Independent Assessment* $\mathcal{A}(pt_i(P), C_{\phi_i})$: Evaluates each patch pt_i solely based on criterion C_{ϕ_i} specific to fault f_i, explicitly disregarding other co-existing faults and their potential effects. This represents the prevalent approach in most conventional APR.
2) *Fault-Aware Assessment* $\mathcal{A}(pt_i(P), C_{\phi_i}, F)$: Considers the presence and potential influence of other known faults in F during the evaluation of patch pt_i. The goal here is to ensure the patch for f_i effectively addresses its target without interfering with or exacerbating other defects.
3) *Incremental Assessment* $\mathcal{A}(pt_i(P), C_{\phi_i}, H_{i-1})$: Evaluates candidate patches sequentially. The assessment of

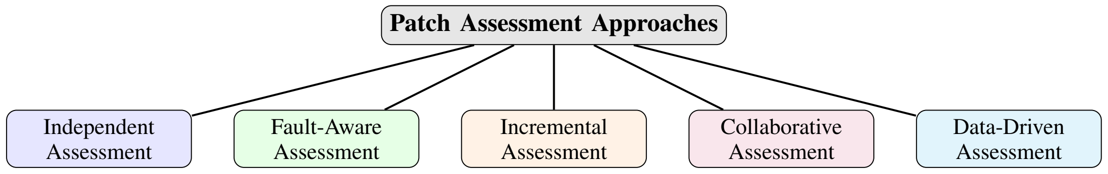

Fig. 1. Taxonomy of patch assessment strategies in multi-fault scenarios. Color highlights the conceptual distinction among the five approaches.

patch pt_i explicitly factors in the cumulative effects and changes introduced by previously applied patches stored in the history H_{i-1}. This aligns with a dynamic, step-by-step validation process for interdependent faults.

4) *Collaborative Assessment* $\mathcal{A}(\{pt_k(P)\}_{k\in S}, C_S)$: Assesses the collective behavior of *multiple candidate patches jointly* (i.e., a set of patches $\{pt_k(P)\}_{k\in S}$) against a unified criterion C_S, rather than evaluating them in isolation. This approach focuses on analyzing the integrated effects of coordinated repairs, which is critical for highly interacting or synergistic defects.
5) *Predictive, Data-Driven Assessment*: $\mathcal{A}\big(\phi(pt_i(P)), \mathcal{M}\big)$: Leverages extracted features $\phi(pt_i(P))$ of a candidate patch, often in conjunction with a learned model $\mathcal{M}$ (e.g., derived from historical patch data), to predict its *validity* or *propensity for overfitting*. This enables proactive prioritization and filtering of patches, even before extensive execution-based validation.

While five assessment methods (see Fig. 1) constitute the core decision strategies for patch validation in APR, they do not exhaust the space of possible validation or prioritization techniques. In practice, one often augments any of the above methods with complementary evaluation layers (see Definition 3), such as formal verification, human-in-the-loop review, performance checks, or heuristic ranking. These techniques can be applied in conjunction with each core method to improve confidence, reduce false positives, or lower testing overhead.

Definition 3: (**Complementary patch assessment methods**). In addition to the assessment methods described in Definition 2, the following techniques can be used to augment patch quality:

1) *Formal verification assessment*: A patch pt is *formally verified* if, after applying pt to program P yielding P', formal methods (e.g., theorem proving, model checking, symbolic execution) establish that P' satisfies a specification φ_{corr}, guaranteeing no new violations.
2) *Human-in-the-loop evaluation*: A patch pt undergoes *human-in-the-loop evaluation* when experts or crowd-workers review it in the context of P to judge correctness, clarity, and side effects beyond automated checks.
3) *Performance/Resource-aware assessment*: A patch pt is assessed under *performance/resource-aware criteria* by measuring non-functional metrics—runtime overhead ΔT, memory usage ΔM, energy consumption ΔE—and ensuring each remains within thresholds $\epsilon_T, \epsilon_M, \epsilon_E$.
4) *Search-based/Heuristic assessment*: A set of patch candidates $\{pt_i\}$ is evaluated via *search-based or heuristic assessment* by defining a fitness function $f(pt)$ (e.g., coverage gain, patch size) and using metaheuristic algorithms (genetic algorithms, simulated annealing) to rank or select the most promising patches.

In Table I, we outline five core approaches to patch assessment, each with strengths suited to different fault scenarios.

Traditional independent assessment is suited to single-fault programs or non-interacting defects, offering simplicity and efficiency. In multi-fault systems, however, the independence assumption risks overfitting or missing masked faults.

Fault-Aware assessment explicitly considers co-existing faults during evaluation to avoid misleading results from previously masked interactions, significantly improving assessment accuracy in multi-fault contexts, though at the cost of deeper analysis and higher computational effort.

Incremental assessment sequentially validates each new patch in the context of previously applied fixes, enabling cumulative validation for cascading faults, but requiring robust patch history tracking and careful management of application order.

Collaborative assessment jointly evaluates multiple patches together to uncover cumulative effects and subtle fault interactions, making it particularly suitable for synergistic or interdependent faults, yet computationally expensive as the number and size of patch sets grow.

Predictive, data-driven assessment leverages learned models and extensive historical repair data to estimate patch quality even before full test execution, enabling rapid triage and prioritization in large codebases, but relying heavily on the quality and coverage of training data.

A. Choosing Assessment Strategies

Our formalization of fault interactions enables a principled approach to selecting the most appropriate assessment strategy for each subpatch. For instance, when faults are believed to be truly independent, a fault-aware independent assessment is suitable to confirm non-interference. Conversely, complex interactions like cascading or masking necessitate more sophisticated strategies, such as incremental or collaborative assessment. Table II summarizes this mapping.

This view supports the strategic application of our five assessment strategies (see Definition 2). Depending on the identified interaction type, APR systems can dynamically choose suitable evaluation strategies, significantly improving accuracy and robustness in complex multi-fault scenarios.

IV. Demonstration: Fault-Aware Patch Assessment

Traditional patch evaluation may incorrectly discard valid subpatches in multi-fault programs due to fault masking. We

TABLE I
COMPARISON OF FIVE PATCH ASSESSMENT STRATEGIES IN APR

Patch Assessment Strategy	Formal Representation	Applicability	Key Advantage	Key Disadvantage
Independent Assessment	$\mathcal{A}(pt_i(P),\, C_{\phi_i})$	Single-fault programs	Quick Check	Overfit Risk
Fault-Aware Assessment	$\mathcal{A}(pt_i(P),\, C_{\phi_i},\, F)$	Multi-fault programs (side-effect checking)	Contextual Accuracy	High Cost
Incremental Assessment	$\mathcal{A}(pt_i(P),\, C_{\phi_i},\, H_{i-1})$	Sequential multi-fault repair	Adaptive Flow	Sequential Burden
Collaborative Assessment	$\mathcal{A}(\{pt_k(P)\}_{k\in S},\, C_S)$	Interacting faults (complex dependencies)	Holistic View	High Complexity
Data-Driven Assessment	$\mathcal{A}(\phi(pt_i(P)),\, \mathcal{M})$	Large Codebases / Rapid Triage	Rapid Triage	Data Dependent

TABLE II
MAPPING FAULT INTERACTIONS TO SUITABLE ASSESSMENT METHODS

Fault Interaction Type	Recommended Assessment Methods
Independence (I)	Fault-aware independent assessment
Cascading (C)	Incremental, Collaborative assessment
Synergy (S)	Collaborative assessment
Masking (M)	Fault-aware, Collaborative assessment
Ambiguous/Unknown	Predictive, Data-driven assessment

illustrate how our fault-aware strategies provide a more reliable assessment by exposing such interactions.

A. Example: Masked Error Activation

Consider a C-like program with two faults: `f1` in `process_input` (improper early exit for input 0) and `f2` in `calculate_ratio` (division-by-zero when denominator is 0). Their interaction is shown in Listing 1.

```c
// Global context (conceptual: could be a shared
    struct/state)
int global_status = 0; // 0: OK, 1: Error
// f1: Early-exit logic bug
void process_input(int value) {
    if (value <= 0) { // f1: should be 'value < 0'
    for valid non-zero inputs
        global_status = 1; return; // Indicate error
     prematurely for value = 0
    }
    // ... further processing ...
    global_status = 0;
}
// f2: Division by zero bug, reachable only if f1
    does not exit early for value=0
int calculate_ratio(int num, int denom) {
    if (denom == 0) { // f2: lacks robust check
    before division
        printf("F2: Division by zero error!\n");
        return -1; // Indicate error
    }
    return num / denom;
}
int main() {
    int input;
    printf("Enter a value: ");
    scanf("%d", &input);
    process_input(input);
    if (global_status == 0) {
        calculate_ratio(100, input);
    }
    return 0;
}
```

Listing 1. Two interacting faults: f1 masks f2

If `input = 0`, `f1` causes an incorrect early termination, preventing `f2` from executing. Hence, `f1` **masks** `f2`. A test here might incorrectly attribute the failure solely to `f1`.

B. Assessment Strategies Compared

Traditional Assessment Fails: An APR tool generates patch `pt1` fixing `f1` (changing <= to <). Now, with `input = 0`, execution proceeds, triggering `f2` and causing a crash. A traditional assessment $\mathcal{A}(pt_1, C_{\phi_1})$ marks `pt1` as *invalid*—a false negative, discarding a correct fix due to an unmasked downstream fault. This leads to overfitting for the program.

Fault-Aware Assessment: A context-sensitive evaluation $\mathcal{A}(pt_1, C_{\phi_1}, F)$, using fault interaction information (e.g., via static analysis identifying data flow), recognizes that `pt1` may reveal hidden faults. This prompts re-evaluation to detect `f2`'s activation post-`pt1`, preserving valid partial progress.

Incremental Assessment: Assessment $\mathcal{A}(pt_1, C_{\phi_1}, H_0)$ executes updated tests after applying `pt1`, correctly detects `f2` (now exposed), and retains `pt1` for continued repair. This prevents premature rejection and supports stepwise patching.

Predictive Assessment: Model-driven $\mathcal{A}(\phi(pt_1), \mathcal{M})$ predicts that `pt1` may expose hidden faults based on its change footprint, data flow, or learned interaction models. This anticipates false negatives and suggests refined evaluation.

Collaborative Assessment: If a candidate fix `pt2` for `f2` exists, collaborative evaluation $\mathcal{A}(\{pt_1, pt_2\}, C_S)$ checks the pair jointly. This ensures holistic correctness when both patches are applied, preventing overlooked interactions.

Summary. This example highlights the critical importance of interaction-aware assessment. Ignoring fault interactions leads to misleading judgments and persistent patch overfitting, while our modular strategies more accurately reflect true correctness in complex multi-fault settings.

V. CONCLUSION AND FUTURE WORK

This work examined the critical challenges of patch assessment in multi-fault scenarios, where traditional validation often fails due to complex fault interactions. We emphasized the need for interaction-aware assessment to improve repair reliability, introduced a taxonomy of five validation strategies, and proposed a formal model to guide their use. By accounting for fault interactions, we move beyond test-passing as the sole validation criterion, offering a more robust foundation for evaluating patches in realistic software systems.

In future work, we plan to devise practical techniques for identifying fault interactions in real-world programs, empirically assess how our approach minimizes false positives and overfitting in generated patches, and explore integration with existing APR tools to guide automated repair in multi-fault environments more effectively.

References

[1] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? overfitting in automated program repair,” in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 532–543.

[2] M. Monperrus, “A critical review of ”automatic patch generation learned from human-written patches”: Essay on the problem statement and the evaluation of automatic software repair,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

[3] X.-B. D. Le, D. Lo, C. Le Goues, and W. Visser, “Overfitting in semantics-based automated program repair,” in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 163–173.

[4] S. K. Jiang, Thushari, and D. Lo, “Shaping program repair space with existing patches and similar code,” in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 298–309.

[5] A. Nilizadeh, G. T. Leavens, X.-B. D. Le, C. S. Păsăreanu, and D. R. Cok, “Exploring true test overfitting in dynamic automated program repair using formal methods,” in *2021 14th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 229–239.

[6] J. Petke, M. Martinez, M. Kechagia, A. Aleti, and F. Sarro, “The patch overfitting problem in automated program repair: Practical magnitude and a baseline for realistic benchmarking,” in *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*, 2024, pp. 452–456.

[7] O. I. Al-Bataineh, “Automated repair of multi-fault programs: Obstacles, approaches, and prospects,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 2215–2219.

[8] V. Debroy and W. E. Wong, “Insights on fault interference for programs with multiple bugs,” in *2009 20th International Symposium on Software Reliability Engineering*, 2009, pp. 165–174.

[9] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, “Statistical debugging: simultaneous identification of multiple bugs,” in *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML)*, vol. 148, 2006, pp. 1105–1112.

[10] N. DiGiuseppe and J. A. Jones, “Fault interaction and its repercussions,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 3–12.

[11] J. A. Jones, M. J. Harrold, and J. F. Bowring, “Debugging in parallel,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 16–26.

[12] OpenAI, “Gpt-4 technical report,” https://openai.com/research/gpt-4, 2023, accessed: 2025-04-23.

[13] H. Ye, M. Martinez, and M. Monperrus, “Automated patch assessment for program repair at scale,” *Empirical Software Engineering*, vol. 26, no. 20, pp. 1–25, 2021. [Online]. Available: https://link.springer.com/article/10.1007/s10664-020-09920-w

[14] A. Ghanbari and A. Marcus, “Patch correctness assessment in automated program repair based on the impact of patches on production and test code,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. ACM, 2022, pp. 1–13.

[15] Q.-N. Phung, M. Kim, and E. Lee, “Identifying incorrect patches in program repair based on meaning of source code,” *IEEE Access*, vol. 10, pp. 12 012–12 030, 2022.

[16] F. Molina, J. M. Copia, and A. Gorla, “Improving patch correctness analysis via random testing and large language models,” in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024, pp. 317–328.

[17] Z. Fei, J. Ge, C. Li, T. Wang, Y. Li, H. Zhang, L. Huang, and B. Luo, “Patch correctness assessment: A survey,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, 2024.