Foundations and Challenges of Multi-Fault Program Repair

OMAR I. AL-BATAINEH, Gran Sasso Science Institute, Italy

Automated Program Repair (APR) has achieved notable success in single-fault scenarios, yet real-world software often contains multiple interacting faults whose behavior fundamentally changes the repair challenge. This paper establishes the first formal foundations for multi-fault APR, treating it as a distinct research domain rather than a simple extension of the single-fault case. We introduce precise models of fault interaction, formalize principles for partial and composite patch validation, and identify intrinsic theoretical limits that define what multi-fault repair can and cannot achieve. Building on this foundation, we identify several pressing challenges for the field, including disentangling faults, designing interaction-aware oracles, developing reliable benchmarks, and defining sound evaluation criteria. To address these issues, we put forward the concept of orchestrated repair, where patch synthesis, validation, and analysis are guided by a central orchestrator that manages dependencies and brings together diverse tools. This reframing treats multi-fault APR as a coordinated, structured process and provides a concrete perspective for future work, helping researchers develop approaches that can handle the complexity of real-world software.

ACM Reference Format:
Omar I. Al-Bataineh. 2025. Foundations and Challenges of Multi-Fault Program Repair. 1, 1 (October 2025), 43 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Automated Program Repair (APR) has achieved notable success in single-fault settings [16, 27], yet real-world software systems seldom contain only one defect. Multiple interacting faults are common, and their presence changes the nature of the repair problem in fundamental ways. Moving from single- to multi-fault APR is not a straightforward extension; it raises challenges that strike at the core of current repair approaches. Beyond practical difficulties, multi-fault scenarios reveal limitations in existing validation and synthesis methods, showing the need for frameworks that account for fault interactions rather than treating defects in isolation.

Despite more than a decade of work on APR, the field still lacks rigorous formal underpinnings, particularly for the multi-fault setting. While extensive empirical studies exist [25, 28–30, 32], the absence of precise models for repair spaces, fault interactions, and patch composability leaves key questions unresolved. This paper lays a formal foundation for multi-fault APR, providing a theoretical framework to analyze, reason about, and guide future research in APR.

Building on this foundation, we outline a research direction centered on collaborative, orchestrated repair. This approach integrates dynamic input generation to expose fault interactions, formal reasoning tools to verify partial and composite fixes, and repair engines coordinated by a central orchestrator. Rather than converging on a single patch, such a system adapts, evaluates, and refines candidate solutions in response to emerging behaviors. By treating multi-fault APR as a structured, iterative process of coordination and feedback, this vision aims to develop repair systems that are resilient and capable of addressing the intricate interplay of multiple defects in real-world software.

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

1.1 Motivation for Multi-Fault APR

Multi-fault APR is an emerging and important area of research for several compelling reasons that justify treating it as a distinct problem space. First, multi-fault APR represents the general case of program repair. Single-fault APR, which has dominated the research landscape for over a decade, is a special case where the set of faults F contains exactly one element ($|F| = 1$). By addressing the general case, we aim to improve the robustness, effectiveness, and applicability of APR systems to the complexity of real-world software, where multiple co-occurring faults are common.

Second, while the APR community has largely overlooked the multi-fault setting, significant research exists on multi-fault localization [7, 17, 23, 41]. Tools and techniques such as CFaults [36], FLITSR [7], MSeer [15], and other related approaches have demonstrated that diagnosing multiple faults is both feasible and necessary [4]. Despite these advances, a critical gap remains: existing methods focus primarily on detection and localization, offering little guidance on how to repair multiple faults once identified. In other words, while we can locate where defects occur, we lack principled strategies to generate correct, coherent patches that account for their interactions. Our work addresses this gap by introducing a formal framework for multi-fault repair that complements and builds upon existing localization techniques, offering a structured pathway from fault identification to comprehensive, interaction-aware repair.

Third, addressing multiple faults together opens a promising path to reducing the risk of overfitting patches. Recent studies reveal a strong link between the operational strategies of current APR tools and the tendency to produce overfitting patches [4, 5]. These tools generally focus on finding the smallest change that makes a given test suite pass, which can result in brittle repairs that do not generalize well. By contrast, a strategy that considers all interacting faults systematically encourages a more comprehensive solution, yielding patches that are more robust and semantically sound by addressing multiple issues and accounting for a wider variety of program states and behaviors.

Finally, from a practical perspective, repairing multiple faults aligns with the realities of software development. Bug reports often point to several intertwined issues rather than a single isolated fault, and developers typically address these as part of a broader problem rather than in isolation. By focusing on programs with multiple interacting faults, our work aims to bring APR closer to the actual practices and needs of the software engineering lifecycle.

1.2 Why Multi-Fault Program Repair Remains Understudied

Despite progress in APR, the field remains largely anchored to the single-fault paradigm. This is not because repairing multi-fault programs is inherently impractical, but due to a combination of implicit assumptions, methodological biases, and infrastructural limitations that have shaped APR research. We unpack the key reasons for this oversight.

1. *Reductionist View.* A common, though rarely formalized, assumption in APR is that multi-fault programs can be fixed by repeatedly applying single-fault repair techniques. This reductionist view overlooks two important realities: *fault interference*, where fixing one fault may mask, alter, or exacerbate others, and *evaluation ambiguity*, where a partial fix can still lead to failing tests, making reliable progress difficult to measure. Such oversimplification significantly underestimates the complexity of repairing multiple interacting faults.
2. *Benchmark-Centric Bias.* The community's reliance on datasets such as Defects4J, ManyBugs, and IntroClass has yielded a benchmark-induced bias. These datasets predominantly contain single-fault programs, inadvertently tuning APR tools and evaluation metrics for a setting that underrepresents multi-fault realities. As a result, more complex but common multi-fault scenarios remain underexplored.
3. *Search Space and Oracle Constraints.* Repairing multiple co-occurring faults introduces a combinatorial explosion in the patch search space, as tools must consider combinations of edits across locations. Current search-based

techniques are poorly equipped to handle this scale. Moreover, conventional test-suite oracles provide only a binary pass/fail signal, which is too coarse to guide repair in the presence of interacting faults [5]. This oracle limitation is a fundamental theoretical challenge that has long constrained multi-fault APR research.

These factors, including flawed assumptions of fault independence, methodological biases, and infrastructural limitations, explain why multi-fault repair remains largely unexplored. This gap is not due to a lack of relevance but to the absence of foundational, rigorous models and tools. Our work addresses this need, reframing multi-fault APR as a distinct and tractable research problem.

1.3 Foundational Perspective

Multi-fault APR constitutes a distinct subdomain within program repair. Given its complexity, however, it is premature to embark directly on empirical studies or tool development without first establishing a solid theoretical foundation. This groundwork is necessary not only due to engineering challenges but also because of the inherent difficulty of the problem: programs with multiple faults often exhibit non-local fault interactions, non-compositional repair semantics, and test-suite ambiguities that undermine abstractions based on the single-fault assumption.

Without formal models defining fault interaction, repair expressiveness, and localization boundaries, APR tool behavior remains hard to predict, and empirical evaluation risks being misleading. We argue that developing scalable, robust techniques for multi-fault repair requires a prior or co-evolving theoretical framework that rigorously defines the space of repairable programs, the assumptions for repair, and the guarantees tools can provide.

This work provides a foundational framework for the community by clarifying the scope of the problem and its inherent limits. Our aim is to ensure that future advances rely on a principled understanding of the challenge rather than on ad hoc trial and error. This paper contributes by establishing the conceptual and formal groundwork on which future studies, benchmarks, and repair tools can be confidently built.

1.4 Novelty and Scope

This work presents the first thorough theoretical framework for the multi-fault automated program repair problem, establishing it as a distinct and rigorous subdomain. While our preliminary study [5] offered a focused analysis of the validation oracle problem under fault interaction, it addressed only one specific challenge.

In this paper, we present a foundational theoretical contribution. We bring structure to the problem space by defining a formal hierarchy of repair challenges and introducing the *fault interaction graph* ($\mathcal{G}_I$) as a semantic model. This model captures how faults and fixes may mask, reinforce, or cascade into one another, and clarifies the fundamental computability limits that constrain repair. These elements underpin the OrchestratedRepair procedure, offering a coherent framework for reasoning about repair. Our goal is to move beyond the limiting assumption of single-fault repair and embrace a principled perspective that treats repair as a coordinated, dependency-aware endeavour.

1.5 Contributions

This paper makes the following key contributions to the field of automated program repair:

- **A Solid Formal Foundation for Multi-Fault APR:** We introduce a theoretical framework for multi-fault APR by providing precise terminology, formal definitions, and models to reason about the problem. We argue that a deep theoretical analysis is necessary to establish a principled basis for future empirical studies, avoiding misleading assumptions and laying the groundwork for more reliable and practical tools.

- **A Formal Model for Fault Interactions:** We develop a formal model to reason about fault interaction relations, such as masking and synergy. This model is crucial for guiding both patch generation and validation, as it enables the computation of feasible repair sequences and helps in identifying correct, comprehensive patches. We explicitly link these formal interaction types to the core challenges of multi-fault repair, demonstrating how they are the root cause of issues like noisy localization and ambiguous oracles.
- **Systematic Deconstruction of the Problem:** We systematically analyze the multi-fault APR problem and identify its intrinsic challenges, such as combinatorial patch growth, oracle ambiguity, and patch interference. We show that these are structural characteristics of the problem rather than mere engineering inconveniences.
- **A Forward-Looking Research Agenda:** We propose a forward-looking agenda for collaborative, orchestrated repair. This vision shifts the focus from finding isolated patches to developing adaptive, feedback-driven repair systems capable of managing the complex interplay of multiple interacting defects. We emphasize that this new paradigm is not intended to replace single-fault APR tools but to coordinate and integrate them systematically, respecting the constraints imposed by fault interactions and enabling more robust, reliable repair outcomes.

These contributions establish multi-fault program repair as a rigorous subdomain of software repair research, providing a precise theoretical foundation, guiding the design of reliable repair techniques, and enabling systematic empirical studies rooted in a well-defined understanding of the problem.

2 Background and Terminology

Automated program repair has advanced considerably in recent years, yet most work continues to assume programs contain only a single fault. This simplifying assumption contrasts with the reality of modern software, where multiple defects often coexist and interact in complex, non-additive ways. In this section, we introduce the terminology and formal framework needed to reason about multi-fault APR. These foundations clarify how the multi-fault setting departs from the traditional paradigm and set the stage for the challenges explored in the following sections.

2.1 The Traditional APR Problem

Early work in automated program repair has primarily targeted the *single-fault setting*. This focus is natural, as isolating a single defect aligns with traditional debugging workflows, simplifies fault localization, and allows test suites to act as practical repair oracles. Seminal repair tools, such as GenProg [25], Angelix [31], and TBar [28], all operate under the foundational assumption that the program contains one isolated fault that can be localized and fixed independently.

Formally, the traditional APR task can be stated as follows. Given a faulty program P and a test suite $T = \{t_1, t_2, \dots, t_m\}$ with at least one failing test case, the goal is to synthesize a program P' such that:

(1) P' is obtained from P by a minimal sequence of edits, and
(2) P' passes all tests in T.

This abstraction has enabled impressive progress in practice, yet it rests on restrictive assumptions. The faulty program is assumed to contain a single defect, the test suite T is assumed to provide adequate coverage, and the notion of minimal edits is assumed to correlate with correctness. In reality, these assumptions do not hold universally. Even in single-fault programs, the correctness of P' cannot be guaranteed: test suites encode only partial specifications, and the fundamental undecidability of program correctness (by Rice's theorem) ensures that a truly sound fix cannot be guaranteed by testing alone. Consequently, overfitting remains an unavoidable artifact of test-based validation. These challenges are magnified in the multi-fault setting, where the very notion of an isolated defect breaks down.

2.2 A Formal Definition of a Program Fault

To establish a rigorous foundation for our analysis, we formally define a *program fault* in terms of its deviation from an intended specification. By framing faults this way, we move beyond the purely operational view in which faults are identified only through failing test cases, and instead adopt a deeper, semantic understanding of correctness. This perspective emphasizes the fundamental relationship between a program's behavior and its intended properties, enabling a more principled analysis of program defects and their repair.

Definition 2.1 (Program Fault)**.** A program P contains a *fault* if its behavior deviates from its specification. Formally, we represent a program and its specification as a pair (P, S), where P is a program and S is its specification. Let $\mathsf{beh}(P)$ denote the observable behaviors of P. A fault exists if $P \not\models S$, meaning there is at least one property $\varphi \in S$ that P fails to satisfy, i.e., $\mathsf{beh}(P) \not\models \varphi$. The specification S can be derived from a test suite, formal properties, or a combination of both.

In the context of automated program repair, the specification S typically consists of properties derived from a test suite. The presence of at least one failing test case indicates that the program fails to satisfy a corresponding property in S. This formal definition is essential for the discussions that follow, as it provides a precise, logical foundation for analyzing fault interactions and program repair.

2.3 Distinguishing Multi-Location and Multi-Fault Repair

In the literature, the terms *multi-location repair* and *multi-fault repair* are often used interchangeably, which has introduced ambiguity [20]. We argue that these are in fact distinct problems, each grounded in different theoretical assumptions. Making this distinction explicit is crucial for framing the challenges that our work seeks to address.

Definition 2.2 (Multi-Location Patch)**.** A patch π is a *multi-location patch* if it consists of a set of code edits $\{\delta_1, \ldots, \delta_k\}$ where $k > 1$, applied to distinct lines or AST nodes in the program.

A multi-location patch may be required to fix a single semantic fault. For example, a single bug in a loop might require correcting both the loop condition and a counter variable to restore correctness. While challenging, multi-location repair remains a form of single-fault repair, as the edits are driven by a single logical error and are often locally co-dependent.

Definition 2.3 (Multi-Fault Repair)**.** *Multi-fault repair* is the process of repairing a program that contains two or more distinct faults, $f_i, f_j \in F$ with $i \neq j$. A successful repair must address all faults in F.

The core insight is that not all multi-location patches are multi-fault patches, and, critically, multi-fault repair cannot be accurately modeled as an iterated series of single-fault repairs. The latter assumption, while convenient, fails to account for the most fundamental challenge: fault interactions.

The prevailing assumption in many APR frameworks is that a multi-fault program can be solved by an iterative process: find and fix one fault, then repeat. This approach is valid only in the simple case of independent faults. However, as we will formalize in Section 2.5, faults can exhibit masking or synergy. When this occurs, fixing one fault may alter the observable behavior of others, invalidating the original fault localization and requiring a complete re-evaluation of the problem space. This non-compositional behavior of patches for interacting faults is the central theoretical and practical challenge of multi-fault repair. We use the terms in this paper with this precise distinction in mind, providing a foundation for our subsequent formal analysis of interaction-aware repair.

2.3.1 *A Formal Hierarchy of Program Repair.*

To provide a clear conceptual framework, we formally define the different program repair problems as sets, revealing a precise hierarchy that grounds our work. We define the problem sets based on the characteristics of the underlying fault structure:

- Let PROG_{SF} denote the set of all Single-Fault repair problems, characterized by the existence of exactly one logical error, where the set of faults F has cardinality $|F| = 1$.
- Let PROG_{ML} denote the set of all Multi-Location repair problems, a subset of PROG_{SF} where the single logical fault requires edits spanning multiple code locations.
- Let PROG_{GEN} (or PROG_{MF}) denote the set of all General Program Repair problems, where the program contains one or more distinct faults ($|F| \geq 1$), encompassing all scenarios where interaction may occur.

Using the general repair set, we establish the following formal relationships that clarify their hierarchy:

(1) Multi-Location as a subset of Single-Fault: By definition, a multi-location repair problem is a specialized case of a single-fault problem where the underlying logical cause is singular. Thus, we have the formal relationship:

$$\text{PROG}_{\text{ML}} \subset \text{PROG}_{\text{SF}}$$

(2) Single-Fault as a subset of General Repair: Any single-fault problem is merely a special, simplified instance of the general repair problem where the fault set cardinality is $|F| = 1$. Since the general problem PROG_{GEN} covers all cases where $|F| \geq 1$, this leads to the relationship:

$$\text{PROG}_{\text{SF}} \subset \text{PROG}_{\text{GEN}}$$

These relationships reveal a clear hierarchy: the general repair problem (which we address as multi-fault repair in the body of this paper) forms the broadest domain, encompassing single-fault repair, which in turn includes multi-location repair. This nested structure, as shown in Fig. 1, confirms our central thesis: multi-fault repair represents the most general and challenging form of the problem, encompassing all other variants.

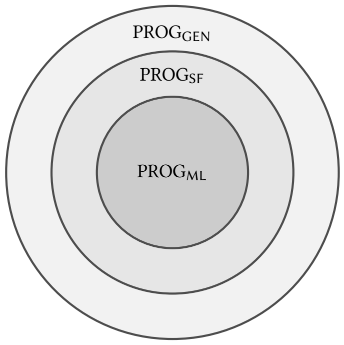
PROG_{GEN}
PROG_{SF}
PROG_{ML}

Fig. 1. Conceptual hierarchy of program repair problems. The set of multi-fault repairs PROG_{MF} strictly contains the set of single-fault repairs PROG_{SF}, which in turn strictly contains the set of multi-location repairs PROG_{ML}.

2.4 Multi-Fault Programs

Real-world software rarely contains just a single, isolated defect. Empirical studies consistently show that multiple faults often coexist within the same module or along the same execution path, creating subtle dependencies that make both debugging and automated repair significantly more challenging. This naturally raises the question: why not simply apply traditional APR iteratively, fixing one fault at a time until all failures are resolved?

Sequential single-fault repair is often inadequate for formal reasons. Iterative approaches implicitly assume fault independence, a condition rarely met in practice. The effectiveness of a repair sequence depends critically on fault ordering: an early patch may mask or alter the manifestation of other faults, while some only surface after others have been addressed. For example, consider a program with a masking fault f_1 and a second fault f_2. A naive tool might first fix f_1, producing a patch that is locally valid but unmasks f_2, introducing new test failures and requiring re-localization and re-repair. In extreme cases, mutual dependencies between faults can lead to an infinite repair loop. This shows that fault ordering is not merely a performance concern: it is a correctness issue that fundamentally shapes the search for a solution, validating the view that independence is the exception rather than the rule.

Second, without a reliable oracle capable of reasoning about partial correctness, iterative approaches cannot guarantee steady progress. A patch that fixes one fault but leaves other interacting faults unaddressed may still trigger test failures, leading the repair tool to incorrectly discard the patch as invalid. This inherent ambiguity in the oracle's signal can prevent iterative tools from making confident and systematic progress toward a complete and correct fix.

Definition 2.4 (Multi-Fault Program)**.** A *multi-fault program* is a program with a fault set $F = \{f_1, \ldots, f_n\}$, and $n > 1$.

To clarify this definition, we provide a specific example demonstrating how multiple faults within the same routine can interact in complex ways. The function in Listing 1 includes three faults that show both masking and synergistic effects, highlighting why fixing faults one at a time is not enough without understanding how faults interact.

```
 1 float process_transaction(float balance, float amount) {
 2     float fee = 0.0;
 3     // F1: Threshold check (off-by-one error)
 4     if (amount > 10.0) { // Should be 'amount >= 10.0'
 5         fee = 1.0;
 6     }
 7   // F2: Premature error (wrong condition - masking fault)
 8    if (balance > amount + fee) { // Should be 'balance < amount + fee'
 9         return -1.0;           // Premature exit masks later calculation errors
10     }
11     // F3: Calculation error (operator misuse)
12     balance = balance - amount + fee; // Should be 'balance - amount - fee'
13 
14     return balance;
15 }
```

Listing 1. Example of multi-fault function with three interacting faults

Masking Interaction ($f_2 \rightarrow f_3$)*.* Fault f_2 (Line 8) implements a *premature error check with an incorrect condition*. Specifically, it returns an error when the balance is actually sufficient (`balance > amount + fee` instead of the correct `balance < amount + fee`). This premature exit masks the downstream fault f_3 (Line 12), since execution halts before reaching it under the failure-inducing test case T_A. If a repair tool fixes f_2 in isolation, the previously hidden f_3 is exposed, causing new failures. A heuristic tool may then misinterpret the f_2 fix as a regression. Capturing such dependencies requires a structural model of interactions, which we introduce in the following subsection.

Synergistic Interaction ($f_1 \leftrightarrow f_3$). Fault f_1 (Line 4) contains an off-by-one condition, while fault f_3 (Line 13) misuses an arithmetic operator. Neither fault alone is sufficient to explain the observed failures:

- Fixing f_1 alone exposes the severity of f_3, leading to an inflated error (the fee is added instead of subtracted).
- Fixing f_3 alone still leaves the threshold condition incorrect.

The only valid solution is a joint patch $\mathcal{P}_{1,3}$ that addresses both faults atomically. Anticipating such clusters again calls for a formal framework to represent and reason about fault interactions, which we develop next.

This example shows that the challenge of multi-fault programs lies not simply in fault multiplicity, but in their structured interactions. Without an explicit model, even advanced LLM-based repair is prone to misinterpretation, backtracking, and wasted synthesis. The next subsection introduces the *fault interaction graph $\mathcal{G}_I$* which provides the structural foundation to orchestrate correct and efficient repair.

2.5 Fault Interactions

Real-world software rarely contains a single, isolated defect. Empirical studies have shown that multiple faults often coexist within the same module or execution path, resulting in subtle dependencies that complicate debugging and repair [4, 11, 12, 43]. These dependencies constitute the central distinction of multi-fault programs. We capture these interactions formally using a graph structure to enable a principled and systematic taxonomy of influence relations.

Definition 2.5 (Fault Interaction Graph ($\mathcal{G}_I$))**.** The interactions among a set of faults F in a program P are formally modeled by a *Fault Interaction Graph*, denoted $\mathcal{G}_I = (F, \mathcal{I})$, where:

- **Nodes** F: The set of all distinct faults in the program, $F = \{f_1, f_2, \ldots, f_n\}$.
- **Edges** $\mathcal{I}$: The interaction relation, defined as a subset of the Cartesian product $\mathcal{I} \subseteq F \times F \times \mathcal{R}$, where $\mathcal{R} = \{\text{Mask}, \text{Synergy}, \text{Indep}, \text{Cascade}\}$ is the set of interaction labels.

An edge $(f_i, f_j, r) \in \mathcal{I}$ exists *if and only if* fault f_i demonstrably influences the run-time behavior (manifestation, observability) or the repair orchestration strategy of f_j by the specific interaction type $r \in \mathcal{R}$. Two faults f_i and f_j are considered *strategically linked* iff either $(f_i, f_j, r) \in \mathcal{I}$ or $(f_j, f_i, r') \in \mathcal{I}$ for some $r, r' \in \mathcal{R}$.

The labeled edges of G_I capture the causal nature of dependencies between faults. We formally define the dependence modes observed in multi-fault programs, linking each to a unique graphical notation for the APR orchestrator.

- **Masking** (r = Mask): $f_i \rightsquigarrow f_j$. A fault f_i suppresses the manifestation of f_j (e.g., by causing an earlier crash), rendering f_j unobservable when both are active. This is an asymmetric causal dependency.
- **Cascading** (r = Cascade): $f_i \rightharpoonup f_j$. A state corruption or error caused by f_i propagates and acts as the input condition that directly activates f_j. This is an asymmetric flow of data or control dependency.
- **Synergy** (r = Synergy): $f_i \leftrightarrow f_j$. Faults f_i and f_j jointly produce a composite failure that cannot be triggered by either fault in strict isolation. This is a symmetric, joint dependency requiring combined repair.
- **Independence** (r = Indep): $f_i \leftrightharpoons f_j$. Faults f_i and f_j affect disjoint parts of the program state and exhibit no observable influence on each other's manifestation or repairability. This is a symmetric, non-causal relation.

These distinct interactions can obscure failures, mislead localization, and compromise validation [4]. Modeling the relation $\mathcal{I}$ is therefore essential for rigorous reasoning about repair correctness in multi-fault scenarios.

Aspect of Repair	**Heuristic/Greedy (Without G_I)**	**Orchestrated/Principled (With G_I)**
Repair Strategy	Reactive, iterative trial-and-error	Proactive, dependency-driven
Handling Masking	High risk of misinterpreting root fix as regression	Orders $f_{\text{mask}} \prec f_{\text{masked}}$; expects failure
Handling Synergy	Fails to converge; discards valid components	Mandates composite patch synthesis ($\pi_{i,j}$)
Search Space	Large, includes redundant/conflicting fix combinations	Pruned; focuses search on interacting clusters
Efficiency	Inefficient; wastes time on conflicting or partial fixes	Efficient; avoids incompatible fix combinations
Correctness	Weaker; relies solely on final test suite pass	Stronger; structurally ensures dependencies are satisfied

Table 1. Comparison of Multi-Fault Repair Strategies With and Without the G_I Model

Extensibility of the fault interaction model. While $\mathcal{R} = \{\text{mask}, \text{synergy}, \text{indep}, \text{cascade}\}$ captures the core functional dependencies documented in empirical multi-fault studies [11, 12], our formal framework is designed for extensibility. The labeled relation $\mathcal{I}$ can be readily augmented to incorporate additional structural dependencies that are critical for repair synthesis and validation. Examples include *Resource Contention*, where f_i and f_j compete for a limited resource; *API Version Conflict*, where fixing f_i necessitates an API change that inadvertently breaks f_j; and *Patch Interference*, where the patch for f_i syntactically overlaps with, and potentially invalidates, the patch for f_j. This inherent flexibility ensures that $\mathcal{G}_I$ remains both scalable and adaptable, capable of representing emerging fault interaction patterns.

2.6 Why $\mathcal{G}_I$ Matters: Orchestration vs. Heuristics

The key value of the fault interaction graph $\mathcal{G}_I$ is its transformation of the repair process from a heuristic, trial-and-error search into a principled, dependency-driven orchestration. While an LLM or traditional APR tool can synthesize a patch, only the $\mathcal{G}_I$ provides the structural intelligence required to ensure global correctness and optimal efficiency in the multi-fault setting. We illustrate this key distinction using two fundamental scenarios derived from our taxonomy $\mathcal{R}$:

Case 1: Masking Interaction. Consider two faults where f_1 masks f_2 (i.e., $(f_1, f_2, \text{mask}) \in \mathcal{I}$).

- *Without $\mathcal{G}_I$ (Heuristic)*: Upon identifying a failing test, the system attempts to repair f_1. Since f_2 remains, the test suite still fails (potentially with a different, f_2-induced failure). The heuristic model may misinterpret this as the f_1 patch being a *regression* or insufficient, leading to rollback or costly re-synthesis.
- *With $\mathcal{G}_I$ (Orchestrated):* The orchestrator, having inferred the $f_1 \prec f_2$ dependency, prioritizes fixing f_1. It then expects the test suite to still fail due to the unmasked f_2. This strategic expectation prevents misinterpreting the result, and the process proceeds directly to the successful localization and repair of f_2.

Case 2: Synergistic Interaction. Consider two faults that are synergistic, requiring a joint fix ($\pi_{1,2}$) to pass all tests.

- *Without $\mathcal{G}_I$ (Heuristic):* The system attempts sequential, single-fault fixes ($\mathcal{P}_1$ then $\mathcal{P}_2$). Since neither single fix will pass the full test suite, both are discarded as invalid. The process enters an endless, ineffective loop of trial and error, failing to realize the requirement for a joint repair.
- *With $\mathcal{G}_I$ (Orchestrated):* The orchestrator identifies f_1 and f_2 as a synergistic cluster. It bypasses sequential patching and mandates the LLM (or synthesizer) to generate a single, joint patch $\mathcal{P}_{1,2}$. This structural guidance efficiently guides the synthesis toward the only valid solution.

Table 1 compares heuristic-driven repair with orchestrated repair using $\mathcal{G}_I$. It synthesizes the insights from the preceding cases, demonstrating how $\mathcal{G}_I$ transforms multi-fault repair from a reactive trial-and-error process into a principled, dependency-aware approach that enhances correctness, efficiency, and robustness.

2.7 The Multi-Fault APR Problem

Building on the preceding discussion, the multi-fault automated program repair problem must explicitly account for interacting faults and for the possibility of addressing them either sequentially or through a unified repair. This fundamental redefinition of the problem is a central contribution of this paper.

Definition 2.6 (Multi-Fault APR Problem). Given a program P with a fault set F and an interaction-aware test suite T_{MF}, the objective is to synthesize a program P'_{MF} such that:

(1) P'_{MF} semantically repairs all faults in F, either through one composite patch or a set of partial patches, and
(2) P'_{MF} passes all tests in T_{MF} and any additional validation suites.

Importantly, this distinction highlights two primary repair modalities, which correspond to different classes of repair tools. Traditional incremental systems tend to produce *partial patches* that address subsets of faults, whereas modern AI-based approaches often attempt *holistic fixes* through *composite patches*. We formalize these patch types below.

Definition 2.7 (Partial Patch). Given a program P with a set of faults $F = \{f_1, \ldots, f_n\}$ and a test suite T, a patch $\mathcal{P}$ is a *partial patch* if, when applied to P yielding $P' = \text{apply}(P, \mathcal{P})$, P' passes a subset of tests in T that were failing for P, but P' fails at least one test in T. A partial patch may resolve some faults in F while leaving others unresolved or hidden.

Definition 2.8 (Composite Patch). Given a program P with a set of faults $F = \{f_1, \ldots, f_n\}$ and a test suite T, a patch $\pi_{\text{composite}}$ is a *composite patch* if it results from the application of a set of patches $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ (where each $\mathcal{P}_j$ addresses one or more faults in F) such that, when applied to P yielding $P_{\text{final}} = \text{apply}(\ldots \text{apply}(\text{apply}(P, \mathcal{P}_1), \mathcal{P}_2), \ldots, \mathcal{P}_m)$, P_{final} passes all tests in T. A composite patch is intended to collectively resolve all faults in F and restore the program's full intended functionality as verified by T.

From this perspective, three fundamental challenges emerge that reveal why multi-fault APR is qualitatively distinct:

- **Fault localization:** Detecting multiple interacting faults is an ill-defined task when masking or cascading relations distort failure observability. Traditional suspiciousness metrics, which are often based on a single fault hypothesis, can become noisy and misleading, guiding the repair process towards irrelevant code regions.
- **Patch synthesis:** Constructing composite patches requires reasoning over a combinatorial interaction space. As the number of faults and their dependencies grow, the patch search space expands exponentially. Sequential repair is insufficient, as the repair order can alter which faults remain detectable, while a single-shot, composite patch must holistically address all dependencies.
- **Patch validation:** Ensuring that P'_{MF} generalizes beyond the immediate test suite is significantly harder than in the single-fault case. Overfitting may occur at the level of individual faults (ignoring masked ones) or at the level of interactions (handling faults independently but not jointly), a challenge with profound implications for the correctness of any automated fix.

These challenges reveal that multi-fault APR is not a mere extension of the single-fault case. Rather, it constitutes a qualitatively distinct problem space, one that demands new theoretical foundations and repair techniques capable of orchestrating dynamic analysis, formal reasoning, and multi-fault synthesis.

3 Why Multi-Fault APR is More Challenging than Single-Fault APR

The transition from single-fault to multi-fault automated program repair is not merely an incremental step; it represents a qualitative shift to a new class of problems. As we formalized in the previous section, the presence of multiple faults gives rise to intricate dependencies and interaction types (e.g., masking, synergy, cascading) that fundamentally break the assumptions of traditional APR. In what follows, we outline seven core challenges that emerge from these formalisms, demonstrating how they distinguish multi-fault APR from the single-fault setting. These challenges are not just practical hurdles but are tied to the fundamental theoretical limits that structure the remainder of this paper.

3.1 Fault Interaction and Entanglement

Building on the formalisms from the previous section, the most significant challenge is the inherent entanglement of faults. Multi-fault programs rarely exhibit faults in isolation. Instead, their behavior emerges from complex interactions. A dominant fault may *mask* the manifestation of others, preventing their detection until the first is resolved. In other cases, multiple defects exhibit *synergy*, where the combination of otherwise benign faults leads to a failure. Cascading effects are also possible, where one fault creates an erroneous state that triggers another. These interaction patterns make both localization and repair non-trivial, as fixing one defect may inadvertently expose additional failures or introduce regressions. In contrast to single-fault APR, which largely assumes fault independence, independence is the exception rather than the rule in the multi-fault setting.

Fig. 2 provides a visual taxonomy of these fault interaction types and their respective impacts on the program repair process. As the figure illustrates, only fault independence allows for a straightforward repair using existing techniques. The other interaction types fundamentally challenge the core assumptions of traditional APR, demanding a more sophisticated, interaction-aware repair strategy.

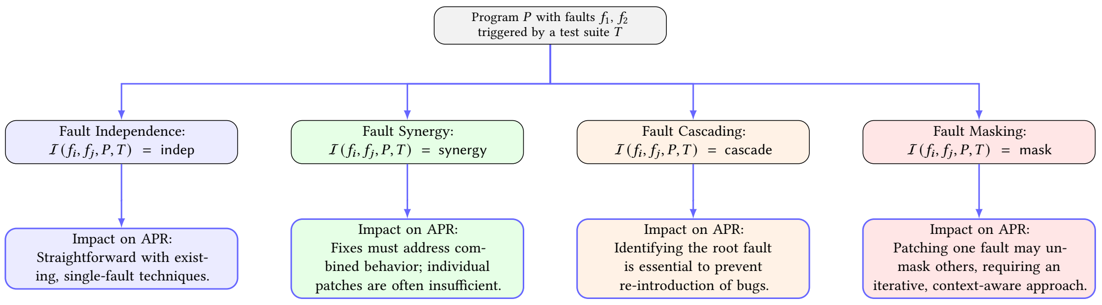
Program P with faults f_1, f_2 triggered by a test suite T
Fault Independence: $\mathcal{I}(f_i, f_j, P, T)$ = indep
Fault Synergy: $\mathcal{I}(f_i, f_j, P, T)$ = synergy
Fault Cascading: $\mathcal{I}(f_i, f_j, P, T)$ = cascade
Fault Masking: $\mathcal{I}(f_i, f_j, P, T)$ = mask
Impact on APR: Straightforward with existing, single-fault techniques.
Impact on APR: Fixes must address combined behavior; individual patches are often insufficient.
Impact on APR: Identifying the root fault is essential to prevent re-introduction of bugs.
Impact on APR: Patching one fault may unmask others, requiring an iterative, context-aware approach.

Fig. 2. A taxonomy of fault interaction types and their impact on automated debugging and repair. $\mathcal{I}(f_i, f_j, P, T)$ denotes the semantic interaction between two faults under a set of failing test cases T.

3.2 A Formal Perspective on Overfitting and Its Complication in Multi-Fault Repair

Overfitting is a central challenge in automated program repair, rooted in the fundamental disconnect between full semantic correctness and behavior on a finite test suite [24, 33, 37, 39]. In the single-fault setting, this problem is well-understood, but in the presence of multiple interacting faults it escalates to a new level of complexity. We formally deconstruct how multi-fault scenarios exacerbate this challenge.

Definition 3.1 (Single-Fault Overfitting)**.** Let P be a faulty program, P_{correct} the corresponding correct program, and T a finite test suite. A patch π yielding $P' = \text{apply}(P, \pi)$ is an *overfitting patch* if $\forall t \in T: \text{run}(P', t) = \text{run}(P_{\text{correct}}, t)$, yet $P' \not\equiv P_{\text{correct}}$ (i.e., P' is not semantically equivalent to the correct program).

In traditional APR, overfitting is an unavoidable consequence of using the test suite as a partial oracle. Thus, a patch may satisfy T by coincidence, repairing the observed failures while introducing behavior that is correct only for the restricted set of inputs in T but fails in other contexts.

In the multi-fault setting, overfitting becomes more insidious. Beyond fitting a partial specification, a patch may also fit a misleading or ambiguous fault model. Because failing tests may not accurately reflect the true root causes of failure, due to masking and synergistic interactions, the repair system may target symptoms rather than causes.

Definition 3.2 (Multi-Fault Overfitting)**.** Let P be a program with fault set F and correctness specification S (e.g., a test suite). A composite patch $\pi_{\text{composite}}$ applied to P is an *overfitting multi-fault patch* if it satisfies S but:

(1) fails to repair one or more faults in F that are masked by others; or
(2) introduces regressions that are not detected by S; or
(3) semantically overfits to the observed fault interactions, failing when context or interaction patterns change.

This form of overfitting is particularly subtle. A naive APR tool, unaware of the fault interaction graph $\mathcal{I}$, may repair a visible symptom without addressing its underlying cause. For example, a synergistic interaction between faults f_i and f_j may be superficially "fixed" by a patch that suppresses the combined failure but leaves both f_i and f_j intact. Such a patch will pass all existing tests, yet it fails once future inputs or changes separate the two faults. This is a form of *deep overfitting*: the patch addresses an observed symptom while the semantic flaws remain.

Similarly, repairing a masking fault f_i without subsequently unmasking and correcting the hidden fault f_j also constitutes multi-fault overfitting. In effect, the tool overfits to the single-fault hypothesis despite the program's genuinely multi-fault nature.

Our proposed procedure, OrchestratedRepair (detailed in Section 6.3), is explicitly designed to address these deeper forms of overfitting. By deriving a repair order from $\mathcal{I}$ and refining the model when validation fails, the procedure moves beyond superficial, symptom-level fixes and aims instead for robust, semantically sound repairs.

3.3 Noisy and Misleading Fault Localization

Fault localization (FL) techniques, such as spectrum-based and mutation-based methods, are based on a fundamental assumption: that there is a direct correlation between a single fault and a failing test suite. In multi-fault settings, this assumption breaks down [42]. Specifically, a single failing test may be caused by a complex interplay of faults, making the observed failure an unreliable signal for localization. This produces noisy and misleading suspiciousness scores, undermining a key input to any APR pipeline.

Formally, let F denote the set of faults in a program P, and let $T_f \subset T$ be the set of failing tests. Traditional suspiciousness metrics, such as Ochiai, compute suspiciousness for a statement s as [1]:

$$\text{susp}(s) = \frac{\text{failingTestsCovering}(s)}{\sqrt{|T_f| \cdot \text{allTestsCovering}(s)}}.$$

In a multi-fault scenario, a failing test $t \in T_f$ may be triggered by a fault set $F_t \subseteq F$. The numerator, failingTestsCovering(s), counts all failures traversing s, regardless of whether s actually contains the fault responsible for the failure.

This gives rise to two key problems: *Misattribution* and *Dilution*. Misattribution occurs when a test fails due to fault $\mathbf{f}_1$ but traverses a clean region s_c near fault $\mathbf{f}_2$. The failure signal from $\mathbf{f}_1$ is incorrectly assigned to s_c, increasing its suspiciousness and masking the true fault location. Dilution occurs because the failure signal ($|T_f|$) is spread across multiple independent fault sites, lowering scores for actual fault locations and hindering prioritization. This corruption of the suspiciousness ranking is a fundamental challenge for multi-fault APR.

3.4 The Multi-Fault Oracle Problem

The limitations of traditional test-suite oracles are not mere inconveniences; they point to a fundamental theoretical challenge. To enable effective multi-fault repair, it is essential to define the properties of an *ideal multi-fault oracle.*

Formally, let P be a program with a set of faults F, and let T be a test suite. Applying a patch $\mathcal{P}$ to P produces a patched program P'. A traditional test-suite oracle, $\mathcal{O}_{\text{trad}}$, returns a binary result:

$$\mathcal{O}_{\text{trad}}(P', T) = \begin{cases} \text{correct}, & \text{if } \forall t \in T,\ P' \text{ passes } t, \\ \text{incorrect}, & \text{otherwise.} \end{cases}$$

While sufficient for single-fault repair, this binary signal is inadequate for multi-fault scenarios. A *partial patch* that fixes only some faults may still fail $\mathcal{O}_{\text{trad}}$, causing the repair process to discard useful partial fixes.

We propose that a multi-fault APR oracle must be *interaction-aware* and produce richer feedback. An ideal multi-fault oracle, $\mathcal{O}_{\text{MF}}$, should satisfy the following properties:

(1) *Partial Correctness Signal:* $\mathcal{O}_{\text{MF}}$ must produce a non-binary output indicating which faults or interactions have been addressed by a patch. This is essential for guiding sequential repair and validating partial fixes.
(2) *Monotonicity:* Applying a valid partial patch for a subset of faults $F' \subset F$ should not cause the oracle to signal regression unless it unmasks a hidden failure. This ensures measurable, non-negative progress during repair.
(3) *Interaction Inference:* The oracle must detect fault interactions. For example, if a patch fixes a failure but causes a previously passing test to fail, the oracle should identify a masking relationship rather than a simple regression. This feedback is vital for orchestrating the repair process.

Designing such an oracle is a substantial research challenge. It requires advances in fault localization, dynamic analysis, and potentially formal methods to go beyond simple pass/fail checks. This oracle is the missing link between the theory of fault interactions and a practical, scalable multi-fault APR framework. Without it, multi-fault repair remains fundamentally constrained by incomplete and potentially misleading guidance.

3.5 Benchmarking Gaps and Evaluation Challenges in Multi-Fault Repair

Automated program repair has advanced significantly in recent years, supported by widely used benchmarks such as Defects4J [22] and ManyBugs [18]. These resources, however, were designed and validated under the assumption that each program version contains a *single* fault. As a result, the community currently lacks a standardized benchmark suite for evaluating APR techniques on programs with *multiple interacting* faults.

This absence creates significant obstacles for evaluation. In the absence of standard benchmarks for multi-fault programs, each study must define its own scope of the problem and validation procedure. Such ad hoc choices hinder reproducibility and make it difficult to compare results across approaches in a systematic way. As a consequence, progress in multi-fault repair remains difficult to measure at the community level.

From a testing perspective, the limitations are even more pronounced. Multi-fault programs pose distinctive challenges: a patch addressing one fault may interact with, obscure, or even exacerbate another. The absence of standardized multi-fault test cases and associated oracles prevents researchers from systematically assessing whether an APR tool can robustly address fault interactions or avoid regressions once a partial fix is applied.

Current evaluation practices also fall short conceptually. Widely used measures such as *correct patch rate* provide limited insight in multi-fault scenarios. They do not capture the nuances of *partial repairs* or the incremental progress an APR technique may achieve when some, but not all, faults are resolved. Without metrics that quantify the degree of repair, such as the number or proportion of faults addressed, it is difficult to establish common ground for assessing progress or to justify claims about the effectiveness of multi-fault repair techniques.

3.6 Patch Interference and Dependency

In the single-fault setting, candidate patches can usually be evaluated independently. In contrast, multi-fault APR introduces interdependencies: applying one patch may alter the program state in ways that are necessary for another patch to succeed, or two patches may conflict, leading to regressions. Consequently, the order in which patches are applied becomes crucial: an inappropriate sequence can obscure faults or even prevent successful repair.

To reason about these interactions, we formalize the notion of *patch interference*. This definition captures dependencies and potential conflicts between patches, providing a foundation for understanding the dynamics of multi-fault repair.

Definition 3.3 (Patch Interference)**.** Let $\boldsymbol{P}_0$ be the original faulty program with fault set $F = \{f_1, \ldots, f_n\}$. We define $\boldsymbol{\Pi} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_k \rangle$ as an ordered sequence of subpatches, where $\mathcal{P}_i$ is the patch intended for fault f_i. The sequential application of this sequence results in the program state $\boldsymbol{P}_k = \text{apply}(\mathcal{P}_k, \ldots, \text{apply}(\mathcal{P}_2, \text{apply}(\mathcal{P}_1, \boldsymbol{P}_0)) \ldots)$. A sequence $\boldsymbol{\Pi}$ is a valid repair sequence for a subset of faults $F' \subseteq F$ if the resulting program $\boldsymbol{P}_k$ semantically corrects all faults in F'. A patch $\mathcal{P}_i$ from a sequence $\boldsymbol{\Pi}$ is said to interfere with a patch $\mathcal{P}_j$ (with $i \neq j$) if the sequence $\langle \mathcal{P}_j, \mathcal{P}_i \rangle$ is not a valid repair sequence for faults $\{f_i, f_j\}$, but the sequence $\langle \mathcal{P}_i, \mathcal{P}_j \rangle$ is.

This definition highlights that in multi-fault repair, patches are not independent variables but elements of a complex system, where the order of application matters as much as the content of the patches themselves. To illustrate this non-commutativity, consider a program with two faults: missing null check (f_1) and a faulty lookup function that occasionally returns a null value (f_2). Let $\mathcal{P}_1$ be the patch to add the null check and $\mathcal{P}_2$ be the patch to fix the lookup function. The sequence $\langle \mathcal{P}_1, \mathcal{P}_2 \rangle$ would be a valid repair sequence: the null check is added first, and then the faulty lookup is fixed, resulting in a semantically correct program. However, the sequence $\langle \mathcal{P}_2, \mathcal{P}_1 \rangle$ could fail. If we first apply $\mathcal{P}_2$ to fix the lookup, the program may no longer produce null values. In this new context, applying $\mathcal{P}_1$ (the null check) might be considered redundant or even incorrect by a sophisticated repair engine, especially if the patch introduces an unintended side effect. This example demonstrates how the application of one patch ($\mathcal{P}_2$) can fundamentally alter the context for another ($\mathcal{P}_1$), causing the second patch to fail and leading to a non-valid repair sequence.

3.7 Undefined Stopping Conditions

In single-fault APR, there is a clear stopping criterion: the repair is considered complete once the program passes all tests in the validation suite. In the multi-fault scenario, this stopping criterion no longer applies, and the problem of patch validation becomes significantly more complex.

One may argue that a multi-fault program can be treated as a single, complex problem and that only the final, composite patch needs to be validated. However, this approach is fundamentally flawed for two systematic reasons. First, it creates a *black box validation problem*: if a composite patch fails, the test suite provides no information about which of the constituent edits or underlying faults is responsible. This lack of diagnostic feedback makes it impossible to systematically debug or refine the patch, reducing the repair process to a series of blind, expensive attempts.

Second, this perspective *ignores the value of partial progress.* As we have argued, sequential repair strategies might produce patches that fix a subset of faults but leave others unaddressed. These partial patches will inevitably cause the program to continue failing some tests. A naive oracle would incorrectly label these as "incorrect", leading the repair process to discard a valid, incremental step towards a complete solution. Therefore, a successful multi-fault repair strategy must be able to reason about and validate partial patches.

This reveals a deeper limitation that extends beyond traditional pass/fail oracles: repair processes require oracles that can offer richer, incremental feedback. To capture this idea, we introduce two complementary notions of correctness oracles. A *partial correctness oracle* assesses the incremental effect of a patch on individual faults, while a *composite correctness oracle* evaluates the correctness of the overall, fully composed fix. Together, these two oracles form the backbone of our formal framework for analyzing progress and completeness in multi-fault repair.

Definition 3.4 (Partial Correctness Oracle)**.** Let F be the set of faults and $\boldsymbol{P}_{\text{old}}$ denote the current partially-patched program state (i.e., the state before the current orchestration step). The oracle evaluates a candidate patch $\mathcal{P}_k$, which is intended to fix a specific fault $f_k \in F$. Let $\boldsymbol{P}'$ be the resultant program after applying the patch, defined as $\boldsymbol{P}' = \boldsymbol{P}_{\text{old}} \circ \mathcal{P}_k$. Let $\mathcal{S}$ denote the set of correctness properties. A partial correctness oracle is a function O_{partial} : Program × Program × $F \times \mathcal{S} \to (\Delta\mathcal{S}, F_{\text{rem}})$ that maps the two program states, the fault set, and the properties $\mathcal{S}$ to the output tuple:

(1) **Net Satisfaction Change ($\Delta\mathcal{S}$):** The set of newly satisfied properties, calculated as: $\mathcal{S}_{\text{satisfied}}(\boldsymbol{P}') \setminus \mathcal{S}_{\text{satisfied}}(\boldsymbol{P}_{\text{old}})$.
(2) **Remaining Faults (F_{rem}):** The subset of faults from F that are still deemed present in $\boldsymbol{P}'$.

A partial patch $\mathcal{P}_k$ is considered successful if $\Delta\mathcal{S} \neq \emptyset$. The orchestrated repair procedure utilizes this oracle to guarantee monotonicity, ensuring that the set of satisfied properties never decreases across successive patch applications.

Definition 3.5 (Composite Correctness Oracle)**.** Let $\boldsymbol{P}$ be the original program with the initial fault set F, and $\boldsymbol{P}''$ the final program obtained by applying a composite patch $\pi_{\text{composite}}$. Let $\mathcal{S}$ denote the set of correctness properties (e.g., test suite). A composite correctness oracle is a function $O_{\text{composite}}$: Program × $F \times \mathcal{S} \to$ (Verdict, $F_{\text{unresolved}}$, $\boldsymbol{\Psi}$) that maps the final patched program $\boldsymbol{P}''$, the original fault set F, and $\mathcal{S}$ to the following output tuple:

(1) **Verdict:** The oracle returns one of the following outcomes:
   - *Correct:* All properties are satisfied ($\mathcal{S}_{\text{satisfied}} = \mathcal{S}$), and all faults are resolved ($F_{\text{unresolved}} = \emptyset$).
   - *Incorrect:* Some properties remain unsatisfied ($\mathcal{S}_{\text{satisfied}} \subset \mathcal{S}$), or new faults (regression) have been introduced.
(2) **Unresolved Faults ($F_{\text{unresolved}}$):** The set of faults from F that remain uncorrected in $\boldsymbol{P}''$.
(3) **Diagnostic Feedback ($\boldsymbol{\Psi}$):** A structured explanation of the failure, identifying remaining fault interactions or patch components that caused the failure/regression.

This oracle thus serves as the final arbiter of correctness, requiring both functional satisfaction ($\mathcal{S}_{\text{satisfied}} = \mathcal{S}$) and structural assurance ($F_{\text{unresolved}} = \emptyset$).

These two notions clarify the dual role of oracles in multi-fault repair. Oracles for *partial patches* must recognize and reward incremental progress, accepting patches that correctly address some faults even if others persist. This requires the ability to associate failing tests with specific faults, a capability that conventional test oracles typically lack. In contrast, oracles for *composite patches* are responsible for diagnosing failures in fully integrated repairs, identifying which component or interaction caused incorrect behavior. By distinguishing these two roles, our framework enables both fine-grained progress evaluation and principled diagnosis of composite patch failures.

Table 2 summarizes the contrast between single-fault and multi-fault program repair. The comparison shows that multi-fault APR is not merely more difficult but structurally distinct: each stage of the pipeline, from fault localization

Table 2. Comparison of challenges in Single-Fault vs. Multi-Fault Automated Program Repair (APR).

Challenge	**Single-Fault APR**	**Multi-Fault APR**
Fault Localization	Focused to a single line or statement.	Noisy, interdependent, and ambiguous. One fault's failure signal can mislead the localization of others.
Patch Validation	Clear (binary outcome). A test suite serves as a definitive pass/fail oracle.	Ambiguous. A patch may fix one fault but fail others due to interactions. Partial passes provide limited signal.
Test Suite Oracle	A single test suite provides a coarse, monolithic oracle.	Fragile, incomplete; fault interactions can mislead the suite, requiring partial oracles.
Fault Interactions	Rare: Faults are assumed independent.	Frequent and complex, with masking, synergy, and cascading dependencies ($\mathcal{I}$). *Example:* a masking fault f_1 prevents fault f_2 from failing a test until f_1 is fixed.
Patch Overfitting	Inherent but contained due to partial test coverage ($T \not\equiv \mathcal{S}$).	Magnified. Overfitting can occur not just to partial tests, but to a misleading fault model from interacting failures.
Repair Expressiveness	Patches target a single logical fault, possibly multi-location.	Composite patches are required; order and interactions must be carefully managed.
Benchmarks	Standardized and widely used benchmark suites (e.g., Defects4J).	Scarce and incomplete; absence of multi-fault benchmarks hampers reproducible, scalable tools and hinders fair comparison.
Stopping Condition	Clear: stops when all tests pass.	Ambiguous; requires a reliable signal that all faults in F are resolved, complicated by masking and oracle limitations.

through patch synthesis to validation, becomes more complex, less deterministic, and less well-defined. These challenges are interrelated and stem from the inherent nature of interacting faults. We argue that they form the central motivation for the research agenda presented in the remainder of this paper, which aims to address these fundamental limitations.

4 The Role of Large Language Models: A Synthesis Engine Within a Principled Framework

Recent advances in Large Language Models (LLMs) such as GPT-4 [35], Codex [8], and CodeLlama [38] have raised expectations about their potential to transform automated program repair. Their ability to generate syntactically correct, context-aware code snippets, often guided only by a prompt and failing tests, has already yielded impressive results on widely used single-fault benchmarks like Defects4J. This progress has, however, encouraged a misconception: that LLMs, by themselves, are capable of resolving the broader and more intricate challenge of multi-fault repair.

We emphasize that this is not the case. LLMs are best understood as powerful *synthesis engines*: tools capable of producing candidate code patches with remarkable fluency, but they do not resolve the fundamental theoretical and practical challenges of program repair. *Multi-fault repair requires more than plausible patch generation*: it requires principled reasoning about fault interactions, systematic validation, and guarantees about correctness and termination that lie outside the scope of current LLM capabilities. In the remainder of this section, we clarify these limitations and outline how LLMs may nevertheless play a constructive role within structured, feedback-driven frameworks.

4.1 Why LLMs Alone Struggle with Multi-Fault Repair

LLMs bring strong synthesis power to APR, but their contributions must be understood in context: they remain fundamentally heuristic and do not provide semantic guarantees. In multi-fault scenarios, this limitation is especially pronounced, as the interaction between faults amplifies the ambiguity of validation and the risk of incomplete fixes.
Heuristics, Not Proof: An LLM-generated patch is a heuristic proposal: it may plausibly satisfy failing tests or even appear structurally correct, but it carries no assurance of semantic validity. Overfitting remains a central risk, as LLMs can produce patches that superficially repair observed behaviors while introducing regressions elsewhere. This issue is not unique to LLMs but is magnified by their tendency to prioritize plausible syntax over formal correctness.
The Oracle Problem and Interaction Ambiguity: The effectiveness of an LLM-generated patch ultimately depends on its oracle, typically a test suite. In the single-fault setting, a passing suite often correlates with correctness. In multi-fault programs, however, the oracle is far less precise: a patch that addresses one fault may still fail due to another (masking), or may only function when paired with another fix (synergy). From the LLM's perspective, all such failures collapse into the same coarse signal—"fail", making it difficult to refine the repair process based on *interaction type*.
Structural Blindness: The Non-Compositional Fix: Perhaps most critically, LLMs lack a structural model of the problem spac. They treat patch generation as an isolated act of prediction rather than part of a structured repair process. Multi-fault repair, however, requires reasoning about compositional dependencies and repair sequence optimization, the very information captured by the fault interaction graph (FIG). An LLM cannot inherently determine:

- The correct order of repair required by a cascading fault chain.
- Whether two seemingly independent faults are actually synergistic and require a single, composite fix.
- If a fix for f_i will unmask a hidden fault f_j.

The LLM can synthesize a patch, but it cannot perform the necessary graph-based reasoning to ensure the fix is structurally sound and globally correct, demonstrating the need for an external orchestrator guided by $\mathcal{G}_I$.
Additional Practical Limitations: Beyond the semantic and structural challenges, LLMs face practical issues. Their outputs are inherently non-deterministic, with successive runs often producing different candidate patches, which makes reproducibility and systematic evaluation difficult. They are also highly sensitive to context: even small variations in prompts or surrounding code can lead to drastically different outputs, undermining stability.

4.2 LLMs as a Component, Not a Complete Solution

Multi-fault repair is inherently a structured process: multiple interacting faults must be localized, disentangled, and incrementally repaired with systematic feedback. LLMs, though highly capable at generating candidate code, are not sufficient as standalone solvers for this task. A more promising perspective is to integrate them as one component within a principled repair framework that orchestrates synthesis, validation, and refinement.

Concretely, one may envision an orchestrated workflow in which an LLM acts as the primary synthesis engine, guided by external analysis. The process might begin with fault localization, using either traditional or LLM-assisted techniques to identify suspicious regions that may harbor multiple defects. Once these regions are identified, the LLM can be prompted with the relevant code context and failing tests to generate candidate patches. These candidates are then subjected to systematic validation and refinement, drawing on complementary mechanisms such as test suites, fuzzing to uncover hidden fault interactions, and, where feasible, formal verification for critical components. In this way, the LLM's generative strength is harnessed not in isolation but as part of a structured loop that incrementally steers repairs toward correctness. In such a design, the orchestrator, not the LLM alone, manages the decomposition of multi-fault repairs, the sequencing of sub-patches, and the systematic validation of results. LLMs thereby function as powerful assistants, extending the space of candidate repairs, but always under the control of rigorous analysis.

5 Formal Model of Fault Interactions

Traditional testing techniques encounter significant challenges when applied to multi-fault programs. First, they often fail to accurately differentiate between multiple faults occurring simultaneously, which makes isolating individual issues more difficult. Second, they are inadequate at managing masking faults, where one fault conceals or prevents the activation of another, resulting in incomplete diagnostics and misplaced confidence in repairs. Third, estimating the total number of faults becomes unreliable when fault interactions alter observable behaviors, complicating efforts to evaluate test completeness or fault coverage. Finally, the computational complexity involved in analyzing multi-fault programs escalates quickly, as the number of potential fault interactions increases exponentially, making it hard to scale existing test generation or analysis techniques. These limitations reveal the need for a deeper, formal understanding of how faults interact and how such interactions fundamentally undermine current testing, localization, and repair.

In this section, we formalize a set of fault interaction types that provide the theoretical foundation for our analysis. These definitions are crucial for understanding why multi-fault programs are not merely a scaled-up version of the single-fault problem, but rather a qualitatively distinct one. Our primary goal is to formalize how these interactions impact observability and interfere with automated reasoning about faults.

5.1 Definitions and Setup

Let P be a program with a set of faults $F = \{f_1, f_2, \ldots, f_n\}$, and let T be a test suite that exposes each fault $f_i \in F$. To reason about how faults behave and interact under different settings, we define the following functions:

- **Execution:** $\mathrm{exec}(f_i, P[C], T) \in \{\mathrm{false}, \mathrm{true}\}$ denotes whether fault f_i is executed under any test in T, when program P is instrumented with the fault set $C \subseteq F$.
- **Visibility:** $v(f_i, P[C], T) \in \{\mathrm{false}, \mathrm{true}\}$ indicates whether fault f_i causes an observable failure in at least one test from T when evaluated in the context of fault set C.
- **Behavior:** $\mathsf{beh}(f_i, C) \in \{\mathsf{C}, \mathsf{H}, \mathsf{E}, \mathsf{L}, \mathsf{N}\}$ represents the failure mode caused by fault f_i when active with fault set C. The values are: C for crash, H for hang, E for uncaught exception, L for logic error, and N for no observable failure. We define the set of disruptive behaviors as $\mathsf{D} = \{\mathsf{C}, \mathsf{H}, \mathsf{E}\}$.

These functions capture fault execution, observability, and behavior, allowing us to formalize the interactions. We assume, for simplicity, that faults f_i, f_j are distinct and coexist in the program.

5.2 Fault Interaction Types

We define an interaction relation $\mathcal{I}(f_i, f_j, P, T)$ that classifies the relation between two faults f_i and f_j in a program P under a test suite T. This relation is central to our formal understanding of multi-fault repair.

- **Independence.** Faults f_i and f_j are *independent* if their effects are isolated, each fault is both visible and functionally stable, whether it appears alone or together with the other.

$$\begin{aligned}\mathcal{I}(f_i, f_j, P, T) = \mathsf{indep} \iff\ & v(f_i, P[\{f_i\}], T) \wedge v(f_j, P[\{f_j\}], T) \\ & \wedge\ v(f_i, P[\{f_i, f_j\}], T) \wedge v(f_j, P[\{f_i, f_j\}], T) \\ & \wedge\ \mathsf{beh}(f_i, \{f_i\}) = \mathsf{beh}(f_i, \{f_i, f_j\}) \\ & \wedge\ \mathsf{beh}(f_j, \{f_j\}) = \mathsf{beh}(f_j, \{f_i, f_j\}).\end{aligned} \tag{1}$$

  Fault independence reflects the ideal scenario for debugging and repair tools. In this setting, the presence of one fault neither masks nor alters the observable effect of another, ensuring that each fault can be localized, understood, and addressed in isolation. This idealized case is the implicit assumption underlying many single-fault APR systems and serves as a clear baseline for reasoning about the more complex, non-ideal scenarios encountered in multi-fault programs, where interactions complicate fault localization and repair.

- **Masking.** Fault f_i *masks* fault f_j if it causes a failure that suppresses f_j's observability when both are present.

$$\mathcal{I}(f_i, f_j, P, T) = \mathsf{mask} \iff v(f_i, P[\{f_i\}], T) \wedge \mathsf{beh}(f_i, \{f_i\}) \in \mathsf{D} \Rightarrow \neg v(f_j, P[\{f_i, f_j\}], T) \tag{2}$$

  Masking occurs when a dominant fault produces a disruptive failure, hiding the presence of another co-occurring fault. This directly explains Noisy and Misleading Fault Localization: a tool might correctly localize and fix the masking fault f_i but be unaware of the masked fault f_j, leading to an incomplete diagnosis and a brittle patch.

- **Synergy.** Faults f_i and f_j exhibit *synergy* if their individual, non-disruptive behaviors combine additively to produce a new, emergent, and observable failure.

$$\begin{aligned}\mathcal{I}(f_i, f_j, P, T) = \mathsf{synergy} \iff\ & \neg v(f_i, P[\{f_i\}], T) \wedge \neg v(f_j, P[\{f_j\}], T) \\ & \wedge\ \big(v(f_i, P[\{f_i, f_j\}], T) \vee v(f_j, P[\{f_i, f_j\}], T)\big)\end{aligned} \tag{3}$$

  Fault synergy arises when the combined effect of f_i and f_j produces a failure, even though each fault is harmless or non-disruptive when considered on its own. This additive interaction complicates both patch synthesis and patch validation, as a repair tool cannot simply address one fault at a time in isolation. Instead, a composite patch is required to resolve the combined effect, and a traditional test-suite oracle would only report failure, offering no insight into the underlying synergistic relationship between the faults.

- **Cascading.** Fault f_i *cascades* to f_j if it alters the program's control or data flow such that f_j, previously dormant, becomes executed and observable as a result of f_i's presence.

$$\mathcal{I}(f_i, f_j, P, T) = \mathsf{cascade} \iff \neg\mathsf{exec}(f_j, P[\{f_j\}], T) \wedge \mathsf{exec}(f_j, P[\{f_i, f_j\}], T) \tag{4}$$

  Fault cascading occurs when one fault, f_i, triggers a chain reaction that activates another fault, f_j, which was previously dormant or unobservable. This highlights the transitive nature of fault propagation. It directly relates to the Combinatorial Growth problem: a repair for f_i might expose a new failure for f_j in a different part of the program, increasing the search space. Cascading also complicates the oracle problem, as a partially fixed program may exhibit new failures, leading a naive oracle to reject a valid partial patch.

This formal model provides a vocabulary for our main thesis: that multi-fault APR is not a mere extension of single-fault repair. The existence of these interaction types proves that the problem is not simply about fixing more bugs but about managing their complex interdependencies. This formalization provides the foundation for building

Attribute	Independence	Masking	Synergy	Cascading
Manifestation	Both faults visible.	Dominant masks other.	Neither fault visible alone.	One activates another.
Cause of Failure	Individual faults.	Single dominant fault.	Combined effect.	Transitive activation.
Repair Complexity	Low (sequential).	Moderate (re-localization).	High (composite patch).	Moderate (order-sensitive).
Impact on Localization	Straightforward.	Misleading (hidden faults).	Ill-defined (no single point).	Complicates root cause analysis.
Impact on Oracle	Clear pass/fail.	Incomplete.	Ambiguous (only combined failure).	Ambiguous (new failures).
Impact on APR	Single-fault tools work.	Requires iterative loops.	Breaks single-fault assumptions.	Repair order critical.

Table 3. Comparison of Fundamental Fault Interaction Types

new theoretical frameworks and engineering novel repair techniques that can orchestrate dynamic analysis, formal reasoning, and multi-fault synthesis.

5.3 The Utility of Formal Fault Interaction Modeling

The formalization of fault interactions $\mathcal{I}$ is far from a purely theoretical exercise; it establishes the *principled foundation* for guiding practical, large-scale multi-fault repair. By explicitly modeling interaction types (indep, mask, synergy, cascade), we gain two intertwined benefits: enabling principled orchestration and supporting rigorous formal reasoning.

1. *Enabling Principled Orchestration (Engineering Utility)*: The $\mathcal{I}$ relation acts as the control input for the orchestrator framework, transforming the repair process from a naive search into an informed sequence of decisions:
   - Optimal Repair Sequencing: The model determines the order in which repairs should be attempted. For example, a fault f_i that cascades to f_j should generally be addressed before f_j. Similarly, a masking fault f_i must be fixed first to reveal the true set of active faults, preventing incomplete or misleading patches.
   - Targeted Validation: The interaction type guides the design of the validation process. For synergistic faults, the orchestrator recognizes the need to search for a composite patch $\pi_{i,j}$, treating any single-fault patch π_i that does not fully resolve the failure as a partial success rather than a failure to discard. This approach mitigates the inherent ambiguity of traditional test-suite oracles.
2. *Informing Formal Reasoning (Theoretical Utility)*: The formal definitions create a concrete bridge between program defects and formal methods, enabling advanced reasoning beyond simple test-suite verification:
   - Temporal Reasoning with LTL: Interaction types can be formally expressed as temporal properties using Linear Temporal Logic (LTL), thereby enabling rigorous formal verification. For instance, masking can be precisely defined as a requirement that a fix for f_i must ensure the observability of f_j whenever f_j remains present in the system. This ensures that a patch actually resolves a defect, rather than merely masking it.
   - Structuring LLM Learning: The defined $\mathcal{I}$ types can serve as structured, temporal training patterns for LLMs. Rather than training them purely on code diffs, we can train on decision-making tuples (e.g., $\langle P, \mathcal{I} = \text{mask}, f_i, f_j, \text{Action} = \text{Fix } f_i \text{ First}\rangle$). This approach trains LLMs not only on synthesis but on principled decision-making informed by the fault structure, moving them closer to predictable, reliable components within the Orchestrator.

In essence, by formally modeling fault interactions, we establish the external scaffolding that enables the powerful but inherently unreliable LLM synthesis engine to operate within logically sound boundaries enforced by the orchestrator.

6 Formalizing Multi-Fault Patch Generation and Validation

Having established our conceptual model for fault interactions, we now formalize how these interactions guide the generation and validation of patches. Unlike the single-fault paradigm, which often assumes a monolithic patch and a binary oracle, multi-fault repair requires a more sophisticated approach that reasons about partial patches, their composition, and a dynamic validation process.

We rely on the formal definitions from Subsection 2.5, where we introduced the set of faults F, the program P, and the fault interaction relation $\mathcal{I} \subseteq F \times F \times \mathcal{R}$, which models interaction types such as masking and synergy. This model is crucial for establishing a *correct repair order* and for interpreting the *validity of partial patches*.

6.1 Patches and Composition Semantics

Fixing a multi-fault program is not simply the sum of fixing individual faults; it requires the coherent integration of multiple repairs. To capture this formally, we define a set of candidate *subpatches*, $\mathcal{P} = \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, where each subpatch $\mathcal{P}_i$ is a localized change targeting a single fault f_i. These subpatches have no standalone meaning: their correctness emerges only through composition. We introduce an operator $\circ$ to merge subpatches into a unified patch:

$$\pi_{\text{composite}} = \mathcal{P}_1 \circ \mathcal{P}_2 \circ \cdots \circ \mathcal{P}_n \tag{5}$$

The semantics of $\circ$ depend on the patch representation (e.g., AST rewrites or instruction-level edits) and govern how and whether one subpatch may interfere with another during composition.

The core challenge of this composition lies in managing patch interactions, which are direct consequences of the underlying fault interactions. For example, a fix for a masking fault must be applied first to unmask the hidden fault; applying the fix for the hidden fault prematurely would fail. This implies that the composition process cannot be arbitrary. A naive parallel composition (e.g., merging all subpatches at once) would be effective only for independent faults. For cascading or masking faults, a sequential composition strategy is required to preserve correct repair order.

In our framework, the fault interaction model $\mathcal{I}$ directly informs the composition strategy. It provides a partial order on the faults, which we formalize in the next section, that dictates the correct sequence for generating and composing subpatches. This ordered, incremental approach avoids the pitfalls of monolithic patch generation and ensures that each repair step is semantically sound within the context of the remaining faults. The objective is to produce a coherent $\mathcal{P}_{\text{composite}}$ that not only passes all tests but also semantically corrects the entire program.

6.2 The Multi-Fault Repair Process and Validation Oracle

Multi-fault repair is inherently an incremental process. A repair system R proceeds in phases, generating and validating subpatches, each targeting a specific fault $f_i \in F$. A key challenge is the *oracle problem*: a simple pass/fail verdict from a test suite T is insufficient when a program still contains unresolved, interacting faults. As we have argued, a successful multi-fault repair strategy must be able to reason about and validate partial patches, which necessitates a more sophisticated oracle mechanism.

To enable sound validation during this progressive repair, we rely on the notions of partial and composite correctness oracles defined earlier. A subpatch is evaluated by a partial oracle, O_{partial}, which assesses whether it constitutes a valid intermediate fix. This oracle provides nuanced, diagnostic feedback essential for guiding the incremental repair process.

This process requires a formal notion of a *valid partial patch* that is grounded in fault-aware semantics. We assume the repair process respects a partial order $\prec_{\mathcal{I}}$ over F, where $f_j \prec_{\mathcal{I}} f_i$ implies that f_j should be addressed before f_i to avoid semantic interference. This ordering is crucial for handling interactions like masking and for ensuring the correctness of intermediate steps.

Definition 6.1 (Valid Partial Patch under Interaction-Aware Ordering)**.** Let $\prec_{\mathcal{I}}$ be a partial order over faults $F = \{f_1, \ldots, f_n\}$ induced by the interaction relation $\mathcal{I}$. A subpatch pt_i is considered *valid* for fixing fault f_i in a program P if:

(1) It eliminates the faulty behavior directly attributed to f_i.
(2) It does not adversely interfere with any previously validated subpatch pt_j for faults f_j such that $f_j \prec_{\mathcal{I}} f_i$.
(3) It preserves the program state so that it does not hinder the detection or repair of any fault f_k such that $f_i \prec_{\mathcal{I}} f_k$.

This definition ensures that each repair step contributes to the overall goal without regressing earlier fixes or compromising future ones. The interaction-aware ordering $\prec_{\mathcal{I}}$ guarantees that fault interdependencies, such as masking and synergy, are respected in both validation and repair sequencing. It is essential to note that a valid partial patch does not need to pass all failing tests, as remaining faults may cause some; however, it must not introduce new regressions or mask other issues. The correctness oracle then handles the final verdict on the entire repair, $O_{\text{composite}}$.

6.3 A High-Level Procedure for Orchestrated Repair of Multi-fault Programs

Building on the formalisms introduced in this paper, particularly the $\mathcal{G}_I$ model, we present a high-level procedure for orchestrating multi-fault APR. This procedure provides a blueprint for future tools, integrating fault interaction analysis with formal validation oracles to guide the repair process. We call this procedure OrchestratedRepair; it takes as input a faulty program P, a set of faults F, and a set of correctness properties S (e.g., a test suite). To make the process concrete, we formalize it in Algorithm 1, detailing the inputs, outputs, and iterative, feedback-driven logic that directs repair.

6.3.1 Principles of OrchestratedRepair. The logical steps of Algorithm 1 are governed by the $\mathcal{G}_I$ structure. The generalized oracles (O_{partial} and $O_{\text{composite}}$) introduced in Subsection 4.2 are instantiated as specialized suboracles: the halting oracle O_{halt}, which checks normal program termination (resolving crashes or infinite loops/hangs), and the output oracle O_{output}, which checks functional correctness. These oracles underpin three core algorithmic principles.

Principle 1: Sequence Optimization via $\mathcal{G}_I$-Guided Topological Sort. The orchestrator defines the repair sequence Σ by ensuring all strict dependencies are respected. This is achieved via a modified *topological sort* over the directed dependency sub-graph $\mathcal{G}_D = (F, \mathcal{I}_{\text{Mask}} \cup \mathcal{I}_{\text{Cascade}})$. Faults involved in Synergy ($\leftrightarrow$) are excluded from this sort and treated as a single compound repair task.

(1) *Priority Determination:* Faults f_i with an in-degree of zero in $\mathcal{G}_D$ are prioritized as root causes.
(2) *Parallelization of Independence:* The Independence relation $f_i \leftrightharpoons f_j$ is explicitly excluded from the dependency sort. All independent faults are flagged for concurrent, parallel repair (as checked by DependenciesResolved in the algorithm), maximizing efficiency by leveraging the verified non-interaction.
(3) *Strict Sequential Fixes:* The sequence Σ mandates that if $f_i \longrightarrow f_j$ or $f_i \rightharpoonup f_j$, the patch $patch_i$ must be successfully validated (according to the criteria defined in Principle 3) before $patch_j$ is attempted.

Principle 2: Dynamic Suboracle Refinement and Isolation. To address fault interference, the orchestrator avoids relying on a fixed, static partitioning of the test suite T. Instead, it constructs a *dynamic suboracle* tailored to each target fault f_i, inferred at repair time through targeted fault isolation. This approach ensures that validation focuses precisely on the fault under repair, minimizing noise from unrelated failures.

- *Isolation Context:* For a given fault f_i, the orchestrator executes P in the context where all partial patches Π_{partial} for faults $f_j \prec_{\mathcal{I}} f_i$ have been applied. This execution effectively resolves upstream dependencies, isolating f_i and thereby *unmasking* its genuine fault symptoms.

Algorithm 1 Orchestrated Multi-Fault Program Repair

```
Require: Program $P$, Fault set $F$, Correctness properties $S$
Ensure: A composite patch $\pi_{\text{composite}}$ or failure
 1: $P' \leftarrow P$
 2: $\Pi_{\text{partial}} \leftarrow \emptyset$                                      ▷ Set of partial patches
 3: $F_{\text{resolved}} \leftarrow \emptyset$                                   ▷ Set of faults fixed by partial patches
                                                          ▷ Step 1: Infer Interactions and Ordering
 4: $\mathcal{I} \leftarrow$ InferInteractionRelation$(P, F, S)$
 5: $\prec_{\mathcal{I}} \leftarrow$ DerivePartialOrder$(\mathcal{I})$
 6: $F_{\text{queue}} \leftarrow$ TopologicalSort$(F, \prec_{\mathcal{I}})$
                                                          ▷ Step 2: Synthesize Partial Oracles
 7: $O_{\text{partial}} \leftarrow$ SynthesizePartialOracles$(F, \mathcal{I}, S)$
                                                          ▷ Step 3: Iterative Patching
 8: while $F_{\text{queue}} \neq \emptyset$ do
 9:     $f_i \leftarrow$ PeekNext$(F_{\text{queue}})$
10:     if DependenciesResolved$(f_i, F_{\text{resolved}}, \prec_{\mathcal{I}})$ then
11:         $\pi_{\text{candidate}} \leftarrow$ GenerateSubpatch$(P', f_i)$
12:         if Validate$(\pi_{\text{candidate}}, O^i_{\text{partial}})$ then
13:             $P' \leftarrow$ ApplyPatch$(P', \pi_{\text{candidate}})$
14:             $\Pi_{\text{partial}} \leftarrow \Pi_{\text{partial}} \cup \{\pi_{\text{candidate}}\}$
15:             $F_{\text{resolved}} \leftarrow F_{\text{resolved}} \cup \{f_i\}$
16:             Dequeue$(F_{\text{queue}}, f_i)$
17:         else                                          ▷ Refinement Loop
18:             $\mathcal{I} \leftarrow$ RefineInteractionModel$(\mathcal{I}, f_i, \pi_{\text{candidate}})$
19:             $F_{\text{queue}} \leftarrow$ Re-Sort$(F, \mathcal{I})$
20:             continue
21:         end if
22:     end if
23: end while
                                                          ▷ Step 4: Final Composite Validation
24: $\pi_{\text{composite}} \leftarrow$ Combine$(\Pi_{\text{partial}})$
25: $O_{\text{composite}} \leftarrow$ SynthesizeCompositeOracle$(S)$
26: if Validate$(\pi_{\text{composite}}, O_{\text{composite}})$ then
27:     return $\pi_{\text{composite}}$
28: else
29:     return 'Failure'
30: end if
```

- *Dynamic Test Set T'_{f_i}:* The resulting *dynamic suboracle* T'_{f_i} is defined as the subset of T whose failures persist when executed in the *Isolation Context*. Dynamic slicing is then applied to the execution traces of T'_{f_i} to verify that the observed failure origin correlates strongly and uniquely with the location of f_i, confirming precise fault isolation and localization for the repair task.
- *Assertion Integration:* Formal assertions extracted from bug reports and specifications are incorporated into T'_{f_i}, serving as high-precision validation points that guide and strengthen repair correctness.

Principle 3: Interaction-Aware Partial Patch Acceptance ($\mathcal{A}$). The acceptance criterion for a partial patch $patch_{f_i}$ explicitly accounts for predictable failures caused by known, unpatched downstream faults f_j. We define $\mathcal{A}$ as a conditional oracle, taking the fault interaction graph $\mathcal{G}_I$ as an explicit parameter, thus formalizing its dependency on

Fault Type	**Subpatch Strategy**	**Full-Patch Strategy**	**Validation Purpose**	**Oracle Source(s)**
Disruptive	Halting checks (timeouts, crash monitors)	Combined $O_{\text{halt}} + O_{\text{output}}$	Ensure termination and functional correctness	Termination provers, test outputs, crash reports
Independent	One oracle per fault (assertion or output-based)	O_{output}	Validate fault-local fixes and isolate behavior	User-provided test suite, inferred postconditions
Synergistic	Compound sub-oracle over joint fault group	$O_{\text{output}} + O_{\text{halt}}$	Detect combined behavior or non-decomposable fixes	Test outcomes, user assertions, joint failure analysis
Masked	Oracle refinement guided by masking; dynamic toggling	O_{output}	Reveal suppressed behavior due to other faults	Test logs, bug traces, patch experiments
Cascading	Oracle refinement staged with patch order	Ordered composite validation using both oracles	Reveal dependent or emergent faults post-fix	Patch ordering traces, dynamic monitors

Table 4. Validation strategies across fault types, patch levels, and oracle sources.

the fault structure. $\mathcal{A}$ is the formal condition required by the Validate function in the orchestrator:

$$\mathcal{A}(\mathcal{P}_{f_i}, T, \mathcal{G}_I) \iff O_{\text{halt}}(\mathcal{P}_{f_i}, T) \ \wedge\ \forall t \in T, \big(O_{\text{pass}}(\mathcal{P}_{f_i}, t) \iff O_{\text{pass}}(P^\star, t)\big) \ \vee\ O_{\text{unmasked}}(\mathcal{P}_{f_i}, t, \mathcal{G}_I).$$

This criterion holds when the patch eliminates non-termination (property O_{halt} holds) and, for every test t, the outcome either matches that of the ground-truth program $P^\star$, indicating a correct fix for an isolated fault, or it satisfies the O_{unmasked} property. The $\mathcal{G}_I$ parameter is essential here: it determines the activation condition for O_{unmasked}, which allows $\mathcal{A}$ to accept $\mathcal{P}_{f_i}$ if and only if f_i is a root cause whose fix intentionally reveals symptoms of a known, unpatched downstream fault f_j as dictated by a $\mathcal{G}_I$-edge ($\rightharpoonup$ or $\leftrightharpoons$). This explicitly distinguishes correct intermediate fixes from predictable failures in a unified, interaction-aware validation framework.

This formulation turns patch validation from a pass/fail measure into an *interaction-aware consistency check*, where acceptance depends not only on passing tests but also on *failing correctly* according to the predictions of the $\mathcal{G}_I$ model.

The formal components comprising the $\mathcal{A}$ criterion are defined as follows:

- $O_{\text{halt}}(\mathcal{P}_{f_i}, T)$: Ensures that applying $\mathcal{P}_{f_i}$ eliminates abnormal termination (crashes) and non-termination (infinite loops or hangs) on all test cases in T. The validity of this check is assessed via a composite approach: (1) dynamic analysis (observing timeouts and crash monitors) and (2) static/formal analysis (leveraging techniques like termination provers for decidable program fragments).
- $O_{\text{pass}}(P^\star, t)$: Denotes that test t passes on the ground-truth program $P^\star$. $P^\star$ is defined as the program version that satisfies the overall correctness properties S (the functional specification) and serves as the gold standard for oracle comparison. In benchmarking contexts, $P^\star$ is typically the developer-fixed version of the faulty program.
- $O_{\text{unmasked}}(\mathcal{P}_{f_i}, t, \mathcal{G}_I)$: Holds when test t fails with an O_{output} failure that is directly attributable to a still-unpatched fault f_j, and this failure is predicted by the $\mathcal{G}_I$ structure (i.e., the presence of a $\mathcal{G}_I$-edge like $f_i \rightsquigarrow f_j$ or $f_i \rightharpoonup f_j$). This confirms that $\mathcal{P}_{f_i}$ has successfully revealed or isolated the subsequent fault, thereby mitigating overfitting.

The acceptance criterion $\mathcal{A}$ provides the formal foundation for partial patch validation. Table 4 serves as the implementation blueprint for the orchestrator's validation step, detailing how the interaction-aware acceptance logic (criterion $\mathcal{A}$) must be practically instantiated across the different fault roles defined by the $\mathcal{G}_I$ model, specifying required oracle sources and necessary patch strategies.

This OrchestratedRepair framework (see Fig. 3) is designed to guarantee robust termination and maximized utility under practical constraints. The procedure is inherently iterative, but its primary, desired termination condition is the successful validation by the final composite correctness oracle $O_{\text{composite}}$. In practice, the framework is also configured

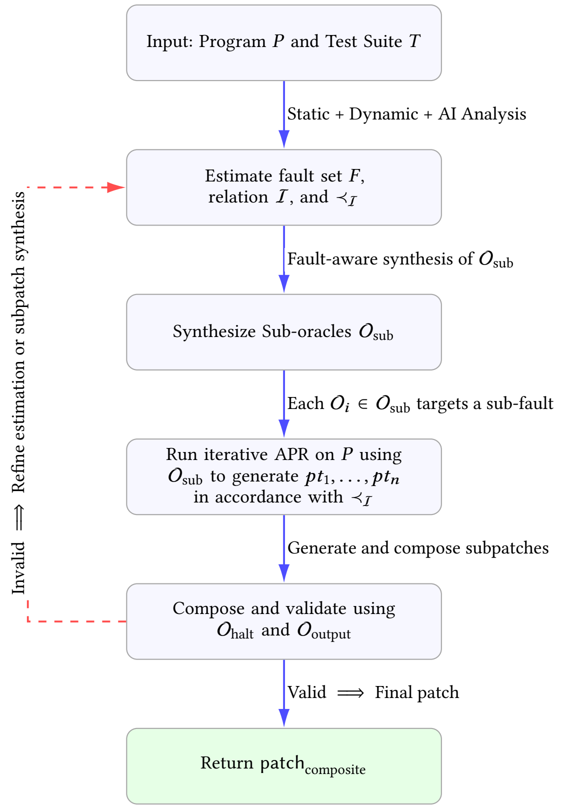
Input: Program P and Test Suite T
Static + Dynamic + AI Analysis
Estimate fault set F, relation $\mathcal{I}$, and $\prec_{\mathcal{I}}$
Fault-aware synthesis of O_{sub}
Synthesize Sub-oracles O_{sub}
Each $O_i \in O_{\text{sub}}$ targets a sub-fault
Run iterative APR on P using O_{sub} to generate $pt_1, \ldots, pt_n$ in accordance with $\prec_{\mathcal{I}}$
Generate and compose subpatches
Compose and validate using O_{halt} and O_{output}
Invalid $\implies$ Refine estimation or subpatch synthesis
Valid $\implies$ Final patch
Return patch$_{\text{composite}}$

Fig. 3. Modular patch synthesis and validation workflow. Subpatches are generated iteratively based on $\prec_{\mathcal{I}}$ and validated in composition. Dashed arrow denotes refinement upon validation failure.

with a finite time budget, serving as a secondary termination condition. Should the time budget be exhausted before a fully valid composite patch is found, the system is designed to report the complete, ordered sequence of validated partial patches (Π_{partial}). This capability offers a critical form of 'best-effort' repair, providing engineers with traceable, non-regressing fixes for a subset of faults. By leveraging the $\mathcal{G}_I$ model to sequence and validate incremental repairs, this framework elevates multi-fault APR from an intractable heuristic search problem to a structured, model-guided engineering process, offering a path toward resilience and diagnosability in complex repair tasks.

6.4 Computational Feasibility and Design Rigor

The viability of the orchestrated repair framework mainly depends on ensuring the computational tractability of its core operations, in particular, the inference of the interaction graph $\mathcal{G}_I$ and the rigorous instantiation of the interaction-aware validation oracle. We demonstrate both the feasibility and the methodological rigor of these components, illustrating how the abstract formalism developed in Sections 3.7 and 5 can be systematically instantiated in practice.

6.4.1 Tractability of Interaction and Partial Order Inference. The operation InferInteractionRelation(P, F, S) aims to derive the complete interaction relation $\mathcal{R}$, consisting of Mask, Cascade, Synergy, and Independence. While the full graph $\mathcal{G}_I = (F, \mathcal{R})$ is theoretically desirable, enumerating all interactions incurs exponential blow-up in the worst case. For practical orchestration, only those interactions that impose a strict temporal dependency are required. This motivates a constrained inference procedure.

Definition 6.2 (Constrained Interaction Graph $\mathcal{G}_{I,sub}$)**.** Let $\mathcal{R}_{sub} = \{\rightsquigarrow, \rightharpoonup\}$ denote the subset of Mask and Cascade relations. We define the *constrained interaction graph* as $\mathcal{G}_{I,sub} = (F, \mathcal{R}_{sub})$. The induced partial order $\prec_I$ is the minimal transitive closure of $\mathcal{R}_{sub}$. This restriction is sufficient for sequencing because only Mask and Cascade interactions introduce genuine precedence constraints; Synergy and Independence admit concurrent treatment during orchestration.

Proposition 6.1 (Polynomial-Time Constructability of $\mathcal{G}_{I,sub}$)**.** *Let F be a set of N suspected faults. The constrained interaction graph $\mathcal{G}_{I,sub}$ can be inferred in time $\mathbf{O}(N^2 \cdot |T| \cdot Time_{Test})$, where $|T|$ is the test suite size.*

Sketch. For each ordered pair (f_i, f_j), we test whether repairing f_i alters the observability of f_j under the test suite T. This yields a dependency iff either a Mask or Cascade relation holds. The pairwise enumeration contributes the $\mathbf{O}(N^2)$ term, while test execution introduces the $|T| \cdot \text{Time}_{\text{Test}}$ factor. Crucially, the procedure remains polynomial in N and linear in $|T|$, in sharp contrast to the exponential space of all interaction subsets. In practice, multi-fault benchmarks have small N, and modern FL techniques (e.g., FLITSR) allow pruning of implausible pairs, further mitigating cost. □

6.4.2 Rigorous Design and Practical Realization of the Partial Validation Oracle. Recall from Definition 3.4 that a *partial correctness oracle $O_{partial}$* is essential for assessing the incremental validity of a candidate patch in the presence of remaining faults. We now formalize its operational realization within the orchestrated repair framework.

Definition 6.3 (Operational Partial Validation Oracle $O_{partial}$)**.** Let $\mathcal{U} \subseteq F$ denote the current set of unfixed faults. Given a candidate patch $\mathcal{P}_k$ for some $f_k \notin \mathcal{U}$, the oracle $O_{partial}$ returns **True** iff the residual failures of the patched program $(P \circ \mathcal{P}_k)$ can be attributed solely to $\mathcal{U}$. Formally,

$$O_{partial}(P \circ \mathcal{P}_k, \mathcal{U}, T) = \textbf{True} \quad \Leftrightarrow \quad \begin{cases} \text{susp}'(f_k) = 0, \\ \forall f \notin \mathcal{U} : \text{susp}'(f) < \epsilon, \end{cases}$$

where susp′ is a multi-fault-aware suspiciousness measure and $\epsilon \geq 0$ is a tolerance bound capturing the maximal noise level admissible for non-faulty locations. In the strict setting, $\epsilon = 0$; in practical instantiations, ϵ may be derived empirically (e.g., from the distribution of susp′ over known non-faulty statements).

Proposition 6.2 (Algorithmic Realization of $O_{partial}$)**.** *The oracle $O_{partial}$ is algorithmically realizable by combining (i) multi-fault localization metrics that minimize the misattribution of failing tests [42], and (ii) causal attribution of execution traces. The operational realization yields a sound instantiation of the partial correctness oracle defined in Definition 3.4.*

Sketch. Executing the patched program $\boldsymbol{P} \circ \mathcal{P}_k$ against T yields updated suspiciousness scores susp′. If all high-suspicion locations correspond exclusively to faults in $\mathcal{U}$, and the location of f_k is deemed resolved (susp′$(f_k) = 0$), then residual failures are attributable only to $\mathcal{U}$. This state guarantees that the Net Satisfaction Change ($\Delta\mathcal{S}$) is non-empty, thereby satisfying the success criterion of the oracle and adhering to the monotonicity requirement of the overall repair procedure (Definition 3.4). Recent AI-based approaches can further refine fault attribution by reasoning over execution traces, offering auxiliary disambiguation. This ensures that $O_{partial}$ is not merely a theoretical construct but an implementable mechanism. □

The two results above jointly demonstrate that (i) dependency inference can be achieved within polynomial bounds, and (ii) partial validation can be grounded in concrete heuristics without departing from the formal requirements. Together, they provide the computational rigor necessary for orchestrated repair to be practically realizable.

6.5 Theoretical Properties and Open Challenges

Our proposed procedure provides a structured, model-guided framework for multi-fault program repair. Its practical effectiveness relies on both foundational theoretical guarantees and inherent implementation constraints. We next highlight the core theoretical limits that any concrete realization of OrchestratedRepair must still address.

6.5.1 Soundness and Completeness. Within our framework, soundness refers to the guarantee that a patch accepted as a valid fix is indeed correct with respect to the full specification. Completeness refers to the ability of the repair system to find a correct patch if one exists.

Proposition 6.3 (Soundness of OrchestratedRepair)**.** *The soundness of the* OrchestratedRepair *procedure is directly dependent on the soundness of its oracles. If the composite oracle $O_{composite}$ and the partial oracles $O_{partial}$ are sound (i.e., they encode the full and partial specifications, respectively), then the entire orchestrated procedure is sound.*

Corollary 6.1 (Undecidability of Perfect Soundness)**.** *Given that test suites provide only a partial specification and program equivalence is undecidable, a perfectly sound oracle is generally impossible to achieve. Thus, our procedure, like any other APR method, cannot guarantee a sound repair for all possible inputs.*

The completeness of our procedure is even more complex. It depends on several factors: the completeness of the bug report analysis for inferring F and $\mathcal{I}$, the completeness of the subpatch generation algorithm, and the termination of the iterative search. Since bug reports are inherently incomplete and patch generation is a search over a potentially infinite space, achieving completeness is not guaranteed.

6.5.2 Complexity and Termination. The procedure's termination is not trivial. While a time budget can be used as a hard limit, a more principled approach requires analyzing the computational cost of the core steps. As demonstrated in Section 6.4 (Proposition 6.1), the constrained inference of the sequencing partial order $\prec_I$ is successfully achieved in polynomial time ($\mathbf{O}(N^2 \cdot |T| \cdot \text{Time}_{\text{Test}})$), resolving the feasibility concern.

However, the complexity of inferring the full, unconstrained interaction graph $\mathcal{G}_I$ (including all Synergy and Independence relations) remains a significant open challenge. Characterizing all potential $\mathcal{R}$ interactions is computationally intractable for even a moderate number of faults. This highlights a critical need for efficient heuristics and AI-driven techniques to fully characterize $\mathcal{I}$ without resorting to the brute-force approach. Furthermore, the overall complexity of OrchestratedRepair is ultimately dominated by the iterative process of patch synthesis and validation, which remains exponential in the worst case due to the underlying search space of possible patches.

7 Principles for Evaluating Multi-Fault APR

The shift from single- to multi-fault automated program repair requires a fundamental rethinking of how success is measured. Metrics commonly used for single-fault repair, such as patch correctness or plausibility based on a binary pass/fail test suite, are inadequate for capturing the complexity of multi-fault scenarios. In this setting, a successful patch may address only a subset of existing faults, interact with other faults, or introduce new, subtle regressions. To account for these challenges, we propose a *multi-dimensional evaluation framework* that moves beyond a single notion of success, providing a more nuanced and comprehensive assessment of multi-fault APR tools.

Developing such metrics is a critical prerequisite for future empirical work. By establishing this framework, we provide a common language for the community to measure and compare progress, ensuring that empirical studies are grounded in a precise understanding of the complexities inherent to multi-fault program repair.

7.1 Evaluation Setting and Formal Notation

To ground our metrics in a realistic evaluation context, we formalize the setting in which a multi-fault APR tool is assessed. We assume a *multi-fault benchmark* of programs with known ground-truth faults. Such a benchmark provides the oracle needed to evaluate repair effectiveness beyond simple test-suite success.

Let $\mathcal{D}$ denote the benchmark, a finite set of faulty programs. For each program $P \in \mathcal{D}$, the associated ground-truth fault set is $\mathcal{F}(P) = \{f_1, f_2, \ldots, f_n\}$. Across the benchmark, the complete fault set is $\mathcal{F}_{\text{total}} = \bigcup_{P \in \mathcal{D}} \mathcal{F}(P)$. A successful repair of a program P yields a corrected program P' in which all faults in $\mathcal{F}(P)$ are eliminated.

When an APR tool is applied to $\mathcal{D}$, its results can be analyzed at the fault level, giving rise to two central sets:

- $\mathcal{F}_{\text{patched}}$: faults for which the tool generated a patch.
- $\mathcal{F}_{\text{fixed}}$: faults correctly repaired, where correctness means the fault is eliminated without introducing regressions, as verified by a comprehensive oracle (e.g., test suite and human inspection).

These definitions provide the measurable foundation for our proposed metrics, ensuring they are both precise and practically grounded, while enabling a rigorous and reproducible evaluation framework.

7.2 Proposed Multi-Dimensional Metrics

Our framework introduces a layered set of metrics to capture the nuanced performance of multi-fault APR tools. Unlike conventional single-fault metrics, it moves beyond binary pass/fail assessments to measure success across multiple dimensions: fault-level granularity, holistic program-level performance, handling of fault interactions, and tool efficiency. These metrics are empirically grounded, relying on information naturally available from multi-fault, such as the ground truth of faults and the outputs of repair tools. By adopting this framework, the research community can perform more meaningful and rigorous comparisons, fostering deeper insight into the capabilities of repair tools. Table 5 presents a complete overview, with formal definitions of each metric organized into distinct categories detailed below.

7.2.1 Fault-Level Repair Accuracy. These metrics provide a fine-grained evaluation of a tool's performance at the granularity of individual faults, which is crucial for distinguishing between a tool that partially fixes many programs versus one that fully fixes a few.

- **Fault Fix Rate (FFR):** The percentage of individual faults correctly fixed across a set of multi-fault programs. This metric serves as a more informative alternative to the traditional "correct patch" measure. For example, a tool that repairs 10 of 20 faults in a program would not be deemed a failure but would yield a 50% FFR, granting partial credit and rewarding tools that make substantial, if incomplete, progress. This measure is essential for guiding the development of tools that address more defects, even when a complete fix is not achieved.
- **Per-Fault Precision and Recall:** These metrics are adapted from information retrieval to assess the tool's effectiveness in addressing faults. Precision measures the proportion of correctly fixed faults among all faults for which the tool attempted a patch. A high precision indicates that the tool's localization and synthesis mechanisms are accurate, avoiding spurious patches. Recall measures the proportion of correctly fixed faults relative to the total number of existing faults in the benchmark. A high recall indicates a tool's comprehensiveness in finding and fixing all relevant defects. The trade-off between precision and recall can be used to distinguish between conservative tools (high precision) and more aggressive, comprehensive ones (high recall).
- **Fault Fix Coverage (FFC):** For a given program, this metric measures the fraction of distinct faults successfully repaired. This metric enables per-program analysis. For instance, on a program with five faults, success is no

longer binary: fixing one fault yields 20% FFC, fixing three yields 60%, and so on. This granular view supports more nuanced comparisons of tools that perform differently on the same program.

The granularity provided by these metrics is critical for understanding why tools succeed or fail in multi-fault settings. They allow for a more detailed analysis than a simple binary outcome.

7.2.2 Program-Level Patch Success. These metrics provide a holistic view of a tool's performance on a per-program basis, complementing the fault-level view by measuring the tool's ability to achieve a complete, working solution.

- **Full Repair Rate (FRR):** The percentage of multi-fault programs for which all faults were correctly fixed. This is the most stringent measure of success, reflecting the tool's ability to provide a complete and satisfactory solution to the developer. It is the direct equivalent of the single-fault "plausible patch" metric, but applied to the far more complex multi-fault scenario. A high FRR indicates a tool is not just finding isolated fixes but can manage the entire repair process for a multi-defect program.
- **Partial Repair Rate (PRR):** The percentage of programs with at least one fault correctly fixed. This metric measures the tool's overall reach. While a full repair is the ultimate goal, a partial repair can still be highly valuable, reducing the developer's workload and simplifying the remaining debugging process. A tool might have a low FRR due to the difficulty of certain faults, but a high PRR would still demonstrate its practical utility.
- **Patch Correctness:** This metric measures whether the final patch successfully fixes all existing faults without introducing regressions. Unlike the single-fault context where correctness is often implicitly defined by test-suite passing, in multi-fault repair, it must be explicitly defined to ensure the final program satisfies all original specification properties and does not introduce new bugs due to fault interactions. This metric is a direct response to the overfitting problem magnified in multi-fault contexts.

These metrics are essential for assessing a tool's capability to deliver a usable solution, going beyond isolated fault fixes to capture whether the repaired program can be relied upon as a correct, stable, and developer-ready outcome.

7.2.3 Interaction-Aware Metrics. These advanced metrics are crucial for evaluating tools that claim to handle fault interactions, moving beyond standard metrics to assess a tool's ability to reason about complex dependencies. These metrics are not feasible with single-fault benchmarks and are a core contribution of our framework.

- **Fault Interaction Index (FII):** This index quantifies how a patch for one fault affects the observability or fixability of another. A tool's FII can be measured empirically by comparing its performance across different repair sequences. For example, if a tool fails to fix a set of faults in one order but succeeds in another, it indicates that the tool's strategy is sensitive to fault interaction, suggesting a high FII. A tool with a low FII is more robust to these interactions, a desirable trait for any multi-fault repair system.
- **Repair Stability:** This metric measures whether the final program state is consistent regardless of the order in which interacting faults are repaired. A high stability indicates that a tool's repair strategy is robust and independent of the order in which faults are addressed. This is a critical property for tools that are intended for real-world use, where the order of bug reports or developer fixes may be non-deterministic.

These metrics provide unique and actionable insights into a tool's capacity to handle the most challenging aspect of multi-fault programs: the complex interaction between defects.

7.2.4 Execution Metrics. While not unique to multi-fault APR, these metrics are essential for comparing the efficiency and practicality of different approaches. They form a vital bridge between theoretical effectiveness and practical utility.

Table 5. A Multi-Dimensional Evaluation Framework for Multi-Fault APR

Metric	**Description**	**Formal Definition**
Fault-Level Repair Accuracy		
Fault Fix Rate (FFR)	The percentage of individual faults correctly fixed across all programs in a multi-fault benchmark.	$FFR = \frac{\lvert\mathcal{F}_{\text{fixed}}\rvert}{\lvert\mathcal{F}_{\text{total}}\rvert}$
Per-Fault Precision	The ratio of correctly fixed faults to all identified faults that are addressed by a generated patch.	$Precision_f = \frac{\lvert\mathcal{F}_{\text{fixed}}\rvert}{\lvert\mathcal{F}_{\text{patched}}\rvert}$
Per-Fault Recall	The ratio of correctly fixed faults to the total number of existing faults in the benchmark.	$Recall_f = \frac{\lvert\mathcal{F}_{\text{fixed}}\rvert}{\lvert\mathcal{F}_{\text{total}}\rvert}$
Program-Level Patch Success		
Full Repair Rate (FRR)	The percentage of multi-fault programs for which all faults are correctly fixed.	$FRR = \frac{\lvert\{P \in \mathcal{P}_{MF} \mid \text{All faults in } P \text{ are fixed}\}\rvert}{\lvert\mathcal{P}_{MF}\rvert}$
Partial Repair Rate (PRR)	The percentage of programs with at least one fault correctly fixed.	$PRR = \frac{\lvert\{P \in \mathcal{P}_{MF} \mid \text{At least one fault in } P \text{ is fixed}\}\rvert}{\lvert\mathcal{P}_{MF}\rvert}$
Interaction-Aware Metrics		
Fault Interaction Index (FII)	A measure of how fixing one fault affects the observability or fixability of another.	Quantified by observing changes in the failing test suite after a partial patch is applied, indicating masking or unmasking.
Repair Stability	Measures the consistency of a repair solution when faults are addressed in different orders.	Assessed by comparing final patch behavior across repair sequences.
Execution Metrics		
Avg Time to Repair	Average time to find a plausible (partial or full) patch.	$AvgTime = \frac{\sum_{P \in \mathcal{P}_{MF}} \text{Time}(P)}{\lvert\mathcal{P}_{MF}\rvert}$
Repair Attempts	The average number of candidate patches or iterations needed to find a successful fix.	$AvgAttempts = \frac{\sum_{P \in \mathcal{P}_{MF}} \text{Attempts}(P)}{\lvert\mathcal{P}_{MF}\rvert}$

- **Average Time to Repair:** Given the expanded search space, the time it takes for a tool to find a plausible patch (partial or full) becomes a primary concern. This metric allows for a direct comparison of the computational costs of different repair strategies, from brute-force search to more targeted, interaction-aware approaches.
- **Repair Attempts per Program:** This metric provides a proxy for the efficiency of the search algorithm. It measures the average number of candidate patches or iterations needed to find a fix. Tools that can converge on a solution with fewer attempts are more efficient and scalable. This metric is particularly useful for comparing generative approaches with search-based ones.

By adopting this multi-dimensional framework, the APR community can move beyond simplistic pass/fail criteria to enable more meaningful, rigorous, and nuanced comparisons of tools, ultimately advancing the state of the art and pushing the boundaries of what automated repair can achieve in real-world, multi-fault programs.

8 Demonstration: $\mathcal{G}_I$-Guided APR Orchestration

As a proof of concept, the core contribution of our research agenda is to show the conceptual advantage of an $\mathcal{G}I$*-guided* APR orchestrator over traditional heuristic-based approaches. Using a representative multi-fault program, we illustrate how the fault interaction graph $\mathcal{G}I$ offers a principled and verifiable basis for sequencing, generating, and validating patches. By structuring the repair process around this graph, the orchestrator achieves greater efficiency, enforces model-consistent correctness at each stage, and systematically mitigates overfitting that limits heuristic systems.

8.1 The Multi-Fault Program and Interaction Landscape

To illustrate the inherent complexity of multi-fault APR, we consider the target program P (Listing 2) alongside a failing test suite T. The effectiveness of our OrchestratedRepair framework rests on the integration of several supporting tools: (i) fault localization techniques (e.g., Ochiai, Zoltar, CFaults) to identify the initial fault set $\mathcal{F} = \{f_1, f_2, f_3\}$; (ii) formal verification tools, including termination provers (TP) such as AProVE or Ultimate Automizer, and assertion inference engines (e.g., CPAchecker, Frama-C) to construct interaction-aware suboracles; and (iii) heuristic assistants (e.g., CodeLlama, GPT-4) to generate initial candidate patches. With this setup, we show how $\mathcal{G}_I$ is inferred and how candidate patches are systematically validated against oracles. This example demonstrates how our framework transforms the multi-fault APR challenge into a structured, verifiable, and principled repair process.

```c
// Initial Input: item_counts = [10, -1, 20], length = 3
int calculate_inventory_value(int* item_counts, int length) {
  // f1 (Dual Role): Off-by-one array access (<= length instead of < length)
  // This single fault has two effects: (1) OOB write at i==3 (Masking/Crash),
  // and (2) Data corruption (item_counts[1] from -1 -> 0) which Cascades to f2.
  for (int i = 0; i <= length; i++)
    item_counts[i] = item_counts[i] + 1;

  // f2 (Cascading): Division by corrupted value
  int total_value = 0;
  for (int i = 0; i < length; i++)
    if (item_counts[i] % 5 == 0) // triggered because f1 corrupted item_counts[1] to 0
      total_value += 100 / item_counts[i]; // potential division by zero

  // f3 (Semantic): Incorrect default initialization for large, zero-valued inventory
  if (length > 2 && total_value == 0)
    total_value = -1;

  return total_value;
}
```

Listing 2. Multi-fault snippet illustrating a dual-role fault f_1 (Masking/Cascade), a Cascading fault f_2, and a masked semantic fault f_3.

Table 6. Fault taxonomy and interactions (as used in $\mathcal{G}_I$).

Fault	Symptom	Notes on interaction / manifestation
f_1	*OOB / Corruption*	Off-by-one write: (1) Cascades to f_2 via data corruption. (2) OOB crash Masks f_2 and f_3.
f_2	*Division-by-Zero*	Triggered by f_1's corruption; when triggered it aborts execution and masks f_3.
f_3	*Semantic Error*	Does not influence f_1, f_2, but manifests only if execution reaches its check (i.e., not masked).

The interaction analysis (the InferInteractionRelation primitive in Algorithm 1) yields the fault interaction graph $\mathcal{G}_I$ shown in Fig. 4. In practice this inference combines differential execution, dynamic slicing, and targeted checks

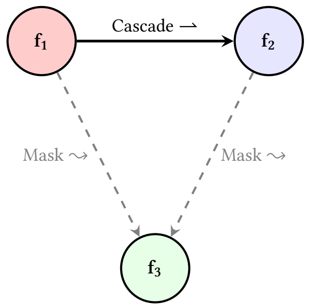

Fig. 4. Fault interaction graph $\mathcal{G}_I$: fault f_1 cascades to f_2, while f_1 and f_2 mask f_3 via early termination. Cascade interactions use solid arrows ($\rightharpoonup$), mask interactions use dashed gray arrows ($\rightsquigarrow$).

of temporal/assertional dependencies to confirm whether one fault's execution or data flow influences another's manifestation. The resulting $\mathcal{G}_I$ is the central artifact guiding our repair orchestration strategy.

8.2 Contrasting Heuristic and Orchestrated Repair

Contemporary LLMs tools such as GPT-4, Codex, and CodeLlama have shown impressive results in single-fault APR. While LLMs can sometimes generate plausible patches for small multi-fault programs, their success is heuristic and not systematic. LLMs are not trained on $\mathcal{G}_I$-aware fault structures and lack the formal machinery to perform interaction analysis or use criterion $\mathcal{A}$. Their reliance on test-passing as the primary objective renders them prone to accidental overfitting, reinforcing the necessity of our structured, model-guided orchestration approach.

8.2.1 Heuristic Analysis and the Overfitting Trap. Conventional APR tools such as `GenProg`, `TBar`, or `Anglix` are inherently unaware of fault interactions. These tools typically treat the entire failing test suite T as a monolithic entity and optimize for a single test-passing objective. Consider the program in Listing 2: here, fault $\mathbf{f_1}$ (an off-by-one or out-of-bounds write) causes an immediate program crash or abort. As a result, the initial test suite T reports only this crash symptom, effectively masking subsequent faults. The tool therefore begins by generating a candidate subpatch $\mathcal{P}_{f_1}$, for example, correcting the loop bound to $i <$ `length`. Once $\mathcal{P}_{f_1}$ is applied, the program advances beyond the crash point, which in turn reveals previously masked faults. Specifically:

- *Unmasking dependent faults:* Correcting $\mathbf{f_1}$ prevents its data corruption, thereby removing state dependencies that conceal $\mathbf{f_2}$. This exposes $\mathbf{f_2}$'s division-by-zero failure if $\mathbf{f_2}$ remains present.
- *Lifting termination masks:* Preventing $\mathbf{f_1}$'s premature termination removes masking effects on downstream code, thereby revealing $\mathbf{f_3}$'s semantic error, even in cases where $\mathbf{f_3}$ might still be masked by $\mathbf{f_2}$.

This interplay of faults demonstrates the core challenge of fault interaction: repairing one fault can fundamentally alter the symptom space. The heuristic validator, however, observes an increase in the total number of failure symptoms, from a single crash to multiple output failures, and, lacking a principled criterion such as $\mathcal{A}$, rejects $\mathcal{P}_{f_1}$ as introducing regressions. This rejection forces the repair process into an intractably large search for a single composite fix $\mathcal{P}_{\{f_1,f_2,f_3\}}$. The resulting search space is combinatorially complex and prone to producing high-scoring but incorrect overfitting patches, highlighting a fundamental limitation of conventional heuristic-based APR.

8.2.2 Orchestrated, $\mathcal{G}_I$-Guided Repair. Our orchestrator, guided by the dependency order $\prec_I$ from $\mathcal{G}_I$ (Fig. 4), reframes repair as structured optimization of sequential and parallel patching, with each candidate validated against a suboracle.

(1) *Root Cause Prioritization (Effectiveness):* The graph clearly dictates the priority. The cascading dependency $\mathbf{f_1} \rightharpoonup \mathbf{f_2}$ ensures that the root cause f_1 must be repaired first, ensuring its corrupted state is corrected before addressing f_2.

(2) *Interaction-Aware Validation (Criterion $\mathcal{A}$):* This criterion represents the key distinction of our approach: partial repairs are not discarded prematurely. For example, the fix $\mathcal{P}_{f_1}$ is accepted if it meets two conditions: (a) it satisfies its fault-specific halting oracle O_{halt}, and (b) it fulfills the unmasking requirement $\mathcal{A}$. The latter demands a change in failure mode: for any test case t that previously resulted in a non-terminating error (Crash/Abort), the patched program must now produce either a terminating failure (incorrect output) or a Pass. This shift ensures that the masking effect is removed, achieving the objective O_{unmasked} by converting a disruptive crash into an observable semantic failure.

(3) *Decoupling and Parallelization (Efficiency):* The causal independence of f_3 ($\mathbf{f_3} \leftrightharpoons \mathbf{f_1}, \mathbf{f_2}$) indicates that $\mathcal{P}_{f_3}$ can be synthesized in a parallel repair task, completely unaffected by the outcomes of the f_1 and f_2 fixes. However, due to the $\mathbf{f_1}, \mathbf{f_2} \rightsquigarrow \mathbf{f_3}$ masking edges, the final validation and integration of $\mathcal{P}_{f_3}$ must be deferred until f_1 and f_2 are verifiably repaired, as shown in §8.3.

8.3 Formalizing Patch Validation with Suboracles

To ensure verifiable correctness for both partial and composite repairs, the orchestrator constructs fault-specific suboracles derived from the initial test suite T and the interaction properties established during $\mathcal{G}_I$ inference. These suboracles allow the orchestrator to validate each repair incrementally, explicitly avoiding the false-rejection and overfitting errors that cripple heuristic APR in multi-fault scenarios.

8.3.1 Suboracle Construction for Partial Patches. The orchestrator tailors the validation strategy dynamically and systematically according to the specific type, severity, and interaction characteristics of the fault being addressed:

(1) *Halting oracle (O_{halt}) for disruptive faults (f_1, f_2):* The halting oracle ensures that a repaired program terminates normally, even if the final output remains incorrect due to remaining faults. This is a prerequisite for accepting any partial repair for disruptive faults. For the root-cause OOB fault f_1, a candidate patch $\mathcal{P}_{f_1}$ must satisfy:

$$O_{\text{halt}}(\mathcal{P}_{f_1}, T_{f_1}) \wedge O_{\text{halt}}(\mathcal{P}_{f_1}, T_{f_2})$$

where T_{f_i} denotes the set of test cases that fail due to fault f_i's manifestation. This check verifies that $patch_{f_1}$ resolves both its direct crash symptom (f_1) and the cascading crash (f_2) that would occur if the data corruption persisted. Formal tools like termination provers or lightweight assertion checkers (e.g., Frama-C, using inferred assertion `assert(i < length);`) formally confirm that the program state leading to the crash is now unreachable.

(2) *Unmasking oracle ($O_{unmasked}$) for masking interactions ($f_1 \rightsquigarrow f_3$):* The core function of the $\mathcal{A}$ criterion is implemented via O_{unmasked}. We define the program status function $\text{Status}(P, t)$ for a test case t as returning one of three outcomes: Crash (non-terminating failure), OutputFail (terminating failure with incorrect output), or Pass (terminating success). A patch for a masking fault (like $\mathcal{P}_{f_1}$) is accepted if it resolves the disruptive masking behavior, even if the overall output remains incorrect due to unmasked faults. This is formally defined as:

$$O_{\text{unmasked}}(\mathcal{P}_{f_i}, t) \iff \big(\text{Status}(P \circ \mathcal{P}_{f_i}, t) \in \{\text{OutputFail}, \text{Pass}\}\big) \wedge \big(\text{Status}(P, t) = \text{Crash}\big)$$

That is, for any test case t that previously caused a Crash in the original program P, the patched program $P \circ \mathcal{P}_{f_i}$ must now yield a terminating status (OutputFail or Pass). For the f_1 fix, this oracle accepts $\mathcal{P}_{f_1}$ because it converts the initial single Crash symptom into observable OutputFail symptoms caused by f_2 and f_3.

(3) *Output oracle (O_{output}) for semantic faults (f_3):* The output oracle validates a patch by checking whether the program exhibits the expected input-output behavior defined by T. This oracle is applied locally and conditionally according to the $\prec_{\mathcal{I}}$ order. For the independent fault f_3, its validation is deferred until the preceding disruptive faults (f_1 and f_2) are neutralized. We first define the intermediate, partially corrected program P':

$$P' = P \circ \mathcal{P}_{f_1} \circ \mathcal{P}_{f_2}.$$

The patch $\mathcal{P}_{f_3}$ is accepted if applying it to P' satisfies its local output oracle:

$$O_{\text{output}}(P' \circ \mathcal{P}_{f_3}, T_{f_3}).$$

The $\mathcal{G}_I$ guarantee of independence $(\mathbf{f_3} \leftrightarrows \mathbf{f_1}, \mathbf{f_2})$ justifies this sequence: since f_3's fix is guaranteed not to re-introduce f_1 or f_2, its validation is sound when conducted on the stable, intermediate program P'.

This modular approach to suboracle construction enables the orchestrator to validate partial correctness for sub-patches, thereby preventing the false-rejection errors that cripple heuristic APR in multi-fault scenarios.

8.3.2 Integrated Validation for Composite Patches. Once all subpatches $\mathcal{P}_{f_1}$, $\mathcal{P}_{f_2}$, and $\mathcal{P}_{f_3}$ have successfully passed their corresponding suboracles, they are composed into the final repair:

$$\pi_{\text{composite}} = \mathcal{P}_{f_1} \circ \mathcal{P}_{f_2} \circ \mathcal{P}_{f_3}.$$

The final validation step applies an integrated oracle:

$$O_{\text{final}}(\pi_{\text{composite}}, T) = O_{\text{halt}}(\pi_{\text{composite}}, T) \wedge O_{\text{output}}(\pi_{\text{composite}}, T).$$

This ensures that the repaired program terminates correctly and produces the expected output for the entire test suite T. By building the final repair through validated subcomponents, the orchestrator avoids the combinatorial complexity of searching for a single composite patch $\pi_{\text{composite}}$, achieving repairs that are both efficient and sound.

This formalization highlights how the core contributions, the interaction graph $\mathcal{G}_I$ and the criterion $\mathcal{A}$, recast multi-fault APR as a structured and verifiable process. The interaction graph $\mathcal{G}_I$ provides the foundation for decomposing an otherwise intractable monolithic search space into an ordered set of manageable subpatch tasks. The criterion $\mathcal{A}$ acts as the guiding principle that enables the orchestrator to avoid the false-rejection trap of heuristic validation. Together, these components yield an orchestration process that produces efficient, verifiable, and non-overfitting repairs.

9 A Forward-Looking Research Agenda: The Collaborative Orchestrator Paradigm

We envision the future of multi-fault program repair as guided by a *collaborative orchestrator*, a unifying process that coordinates synthesis, validation, and analysis across multiple interacting components. This paradigm reframes PROG$_{\text{MF}}$ not as the pursuit of a single static patch, but as an *adaptive process of coordination and refinement* shaped by a central reasoning architecture. The orchestrator's goal is to increase the chances of producing correct and general repairs by managing fault dependencies, interpreting diverse validation feedback, and bringing together the strengths of different repair mechanisms.

The central contribution of this vision lies in shifting from isolated repair attempts toward structured collaboration. Rather than treating each repair tool as an independent solver, the orchestrator treats them as complementary participants in a coordinated process. It continuously integrates information from multiple oracles, analyses the structure of the fault interaction graph $\mathcal{G}_I$, and adapts repair strategies in response to emerging dependencies and partial successes. In doing so, it replaces ad hoc search with deliberate reasoning over the evolving space of possible repairs.

Conceptual Definition: The Neuro-Symbolic Engine. At the core of this paradigm lies the *neuro-symbolic engine*, a reasoning architecture that unites structured, formal analysis with adaptive synthesis. It provides a principled mechanism through which deductive constraints and exploratory reasoning operate in concert to guide repair.

(1) **Symbolic Component.** This layer realises the theoretical framework developed in this paper—the fault interaction graph ($\mathcal{G}_I$), the partial oracle (O_{partial}), and the composite oracle ($O_{\text{composite}}$). It governs dependency management, sequencing, and the verification of candidate patches against formal constraints. Through this layer, the orchestrator maintains soundness and ensures that repairs evolve consistently with interaction semantics.
(2) **Adaptive Component.** Complementing the symbolic layer, the adaptive component provides the flexibility required to explore and refine candidate solutions within large, uncertain repair spaces. It interprets behavioural traces, re-evaluates fault hypotheses, and proposes candidate patches under the guidance of the symbolic framework. Its purpose is not to replace formal reasoning, but to extend it—to operate effectively where the search space defies exhaustive analysis.

These components operate in a continuous reasoning cycle where the symbolic layer constrains and interprets adaptive exploration, while the adaptive layer extends the search frontier under symbolic guidance. This interaction defines the *neuro-symbolic engine* as the conceptual and architectural core of the collaborative orchestrator paradigm. It transforms multi-fault repair from isolated attempts into a coherent process of hypothesis, validation, and refinement, grounded in formal reasoning yet flexible enough to address the evolving complexity of real-world software.

9.1 The Collaborative Orchestrator Architecture

The foundation of this research agenda is the collaborative orchestrator architecture, the conceptual blueprint illustrated in Fig.5. This architecture formalizes the proposed multi-agent system, positioning a centralized, neuro-symbolic engine (the orchestrator) to manage the interplay between specialized components: *Formal Methods (FM)*, *Fuzzing*, and *APR*.

The orchestrator's intelligence is derived from its ability to use the $\mathcal{G}_I$ model as its decision logic, allowing it to dynamically translate abstract theoretical constraints (e.g., fault dependencies) into concrete execution guidance (e.g., targeted properties for FM or constrained templates for APR). This systematic integration defines the three core axes of our forward-looking research agenda, detailed in the following subsections.

9.2 Inferring and Characterizing the Fault Interaction graph $\mathcal{G}_I$

The orchestrator's viability is fundamentally dependent on an accurate, dynamic model of $\mathcal{G}_I$. Since $\mathcal{G}_I$ cannot be fully known a priori, its inference and characterization represent the first core research challenge. We outline a trajectory in which the orchestrator's neuro-symbolic reasoning core synthesizes $\mathcal{G}_I$ by integrating evidence from complementary analysis techniques to not only detect interactions but also classify their type.

9.2.1 Gathering Multi-Modal Evidence for Interaction. The orchestrator integrates partial insights from multiple specialized analyses to form a robust hypothesis for the existence of an edge $(f_i, f_j) \in \mathcal{G}_I$. Let P denote the multi-fault program, $P[pt]$ the program with partial patch pt applied, and $\mathsf{Fail}(P, t)$ the Boolean outcome of executing P on test t.

- **Dynamic Analysis via Incremental Test Outcomes.** This method detects fault interactions by applying a partial patch pt_i to fault f_i and observing changes in the failure profile of fault f_j. An edge $(f_i, f_j) \in \mathcal{G}_I$ is confirmed when such patch application alters f_j's failure outcomes, indicating interaction between faults:

$$\text{Evidence}_{\text{Dyn}}(f_i, f_j) \;\Leftarrow\; \exists t \in T : \mathsf{Fail}(P[pt_i], t) \neq \mathsf{Fail}(P, t). \tag{6}$$

This approach provides measurable evidence of fault interaction by quantifying changes in the failure spectrum. It constitutes a key foundation for $\mathcal{I}$ inference, enabling the orchestrator to prioritize repair sequencing.

- **Static Dependency and Symbolic Analysis.** Structural coupling is suggested by overlapping control-flow or data-flow dependencies between the program regions associated with faults f_i and f_j, reflecting potential semantic correlation. Let Def-Use(f) and Use-Def(f) be the set of program locations involved in the definition and use chains of f. Simultaneous activation is suggested when these dependency chains intersect:

$$\text{Evidence}_{\text{Static}}(f_i, f_j) \;\Leftarrow\; \text{Def-Use}(f_i) \cap \text{Use-Def}(f_j) \neq \emptyset. \tag{7}$$

This analysis provides a high-confidence structural prior for the orchestrator, linking static program structure to fault interaction by revealing shared definition–use chains that imply semantic and execution coupling.

- **Execution Trace Clustering.** This method performs lightweight failure analysis by clustering failing execution traces based on shared dynamic profiles, such as overlapping stack traces, control-flow paths, or contexts. Let $\Sigma(f)$ denote the set of failing traces covering the location of fault f. An interaction between f_i and f_j is supported when these faults co-occur within the same failure contexts, indicating potential temporal coupling:

$$\text{Evidence}_{\text{Trace}}(f_i, f_j) \;\Leftarrow\; \frac{|\Sigma(f_i) \cap \Sigma(f_j)|}{|\Sigma(f_i) \cup \Sigma(f_j)|} \geq \tau_{\text{co}}, \tag{8}$$

where τ_{co} is a configurable co-occurrence threshold. This technique provides robust empirical evidence of fault interaction by correlating failures that are proximate in execution space or time, enabling the orchestrator to infer causal relationships without exhaustive static analysis.

9.2.2 Characterizing $\mathcal{I}$ for Strategic Sequencing. Detection of fault interaction is insufficient; effective orchestration requires classifying the interaction type, which dictates optimal patch sequencing and resources. The orchestrator's neuro-symbolic core classifies each edge $(f_i, f_j) \in \mathcal{G}_I$ by determining which formal type (e.g., mask, synergy) in Section 5 matches the collected multi-modal evidence. This classification maps the observable dynamic consequences, captured by the set of failing tests $T_{\text{fail}}(P) = \{t \in T \mid \mathsf{Fail}(P, t)\}$, to the underlying properties (visibility, execution, behavior) established in Section 5. The key observable signatures for this classification are summarized in Table 7.

Formal Verification for Causal Inference. While dynamic analysis provides high-confidence evidence of correlation (T_{fail} exhibits differential behavior, a quantifiable alteration in the program's failure spectrum after applying a partial patch), *formal verification* is necessary to establish *causality* and achieve maximum inference accuracy. The formal definitions of masking and cascading can be directly translated into properties verifiable via **LTL** (Linear Temporal Logic) or other program verifiers [6, 9]. For example, masking ($f_i \to f_j$) can be verified by asserting:

$$\text{FM}_{\mathcal{I}}(f_i \to f_j) \iff \mathbf{V}\left(P[\{f_i, f_j\}], \neg\mathsf{AssertedFailure}(f_j) \mid \mathsf{AssertedFailure}(f_i)\right)$$

where $\mathbf{V}$ denotes a successful verification, and $\mathsf{AssertedFailure}(f_j)$ is a formal specification of the desired failure post-condition of f_j, typically derived from contextual artifacts such as the failing test oracle output, the bug report's expected behavior, or an explicit fault assertion model. When the verifier confirms causality, the orchestrator elevates the confidence score of the corresponding edge in $\mathcal{G}_I$ to its maximum value, overruling conflicting heuristic evidence.

Table 7. Classification of Fault Interaction Types and Required Orchestration Strategy.

Interaction Type	**Notation**	**Dynamic Inference Signature**	**Orchestrator Strategy**
Independence	$f_i \leftrightharpoons f_j$	Failures of f_j unchanged after $patch_i$	Concurrent repair; independent APR
Masking	$f_i \rightsquigarrow f_j$	$patch_i$ hides a subset of f_j's failures:	Prioritize f_i; $\mathcal{I}$ dictates sequence; re-localize f_j
Synergy	$f_i \leftrightarrow f_j$	Individual patches fail; joint succeeds	Mandate composite patch synthesis
Cascading	$f_i \rightharpoonup f_j$	$patch_i$ fixes f_i but introduces new failure f_j	Strict regression; re-localize and re-synthesize f_j

Inference Complexity and the Accuracy-Cost Trade-off. Achieving high-accuracy inference must be balanced against computational cost. As formally established in Section 6.4, the necessary pairwise execution cost for determining any potential interaction is bounded by the tractable polynomial $\mathbf{O}(N^2 \cdot |T|)$, where N is the number of faults and $|T|$ is the test suite size. However, accurately classifying the interactions, specifically disentangling synergy and independence, requires analysis whose complexity approaches undecidable or exponential bounds, a challenge noted in Section 6.5. Therefore, the orchestrator adopts a *resource-aware strategy*: lightweight heuristics such as trace clustering generate initial $\mathcal{G}_I$ priors and establish the partial order $\mathcal{G}_{I,sub}$. This enables the selective application of costly, high-accuracy verification to only those fault pairs with ambiguous or critical signatures (e.g., high potential for synergy). This explicit trade-off between classification accuracy and cost defines the core optimization problem addressed by the orchestrator's dynamic resource-allocation engine.

These classification rules, backed by the confidence scores derived from the multi-modal evidence and formal verification, provide the foundation for the orchestrator's sequence planner:

- **Rule-of-Priority (Masking):** If $f_i \rightarrow f_j$ is inferred, the Orchestrator enforces that pt_i must be generated and validated before any patch attempt for f_j, as fixing f_i is necessary to establish the correct context for localizing f_j.
- **Rule-of-Synthesis (Synergy):** If $f_i \leftrightarrow f_j$ is inferred, the Orchestrator constrains the generative APR agent to produce a single, composite patch $pt_{i,j}$ addressing both faults simultaneously, since the individual fixes are semantically incorrect or functionally incomplete.

The outcome of this stage is a weighted, directed graph $\mathcal{G}_I$ that encodes both the existence and type of fault interactions. This graph provides the critical domain knowledge for the orchestrator's reasoning, guiding the subsequent synthesis and validation phases with maximal efficiency.

9.3 Orchestrating Interaction-Aware Synthesis

A core principle of the orchestrator is the utilization of the model $\mathcal{G}_I$ as its decision logic, transforming repair from a blind search into a deliberate sequence planning problem.

- **Sequence Determination:** The orchestrator must infer the current graph $\mathcal{G}_I$ (potentially using initial multi-fault localization) to determine the optimal repair sequence. For example, if fault f_i masks fault f_j, the orchestrator should prioritise generating a patch for f_i first, thereby unmasking f_j and enabling its subsequent repair.
- **Targeted Synthesis:** Rather than permitting a high-expressiveness tool (such as an LLM or evolutionary APR) to generate patches indiscriminately, the orchestrator uses $\mathcal{G}_I$ semantics to constrain synthesis. For instance, in the case of synergy faults, it may instruct an LLM to produce a single composite patch addressing both faults simultaneously, instead of generating two distinct subpatches.
- **Agent Integration:** The orchestrator serves as the integrative core connecting specialised repair agents. It routes the problem: the $\mathcal{G}_I$-graph and partial test suite are passed to a Sequence Planner, which determines the

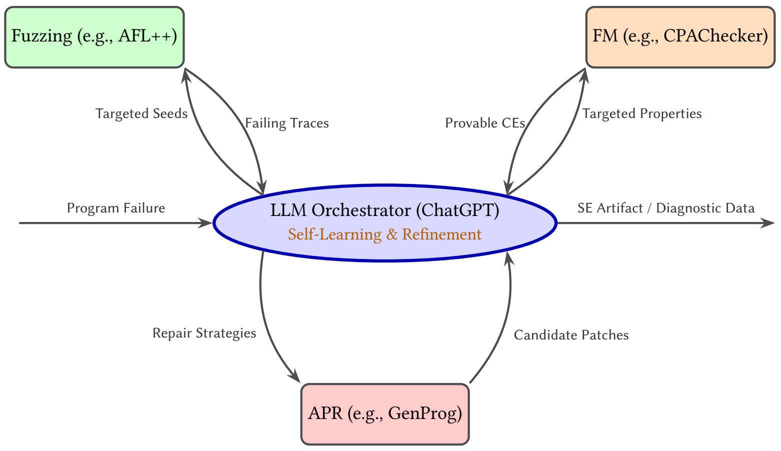

Fig. 5. The collaborative orchestrator paradigm: A conceptual architecture for multi-fault APR that utilizes the formal model $\mathcal{G}_I$ (Interaction) as its decision logic to manage repair sequencing and employs multi-modal validation (FM, Fuzzing) to constrain patch synthesis against fundamental limits (overfitting).

required patch type and fault location, then directs this to a Synthesis Engine (e.g., an LLM or traditional APR tool). This maximises the efficiency and focus of each agent.

9.4 Multi-Modal Validation Against Fundamental Limits

Our formal analysis reveals that overfitting and ambiguous oracle feedback are inherent and persistent challenges in multi-fault programs. The orchestrator addresses these challenges by supplanting the traditional single, weak test oracle with a *multi-modal, layered validation system*, a principled architecture that integrates complementary verification mechanisms to robustly guide patch synthesis and ensure correctness under complex fault interactions.

- **Bridging Synthesis and Verification:** This agenda calls for hybrid approaches that combine the expressiveness of generative tools (LLMs) with the rigour of structured validation (formal methods, FM). The LLM generates plausible patches, but FM components, guided by the model $\mathcal{G}_I$, filter them. For example, the $\mathcal{G}_I$ model can provide precise properties (e.g., LTL assertions) that must hold when f_i is fixed, ensuring f_j remains unbroken.
- **The Fuzzing/FM Synergy:** Patches passing FM checks are further tested via interaction-aware fuzzing. This is not a blind search; it is guided by formal counterexamples and $\mathcal{G}_I$-model knowledge of dependency hot-spots. This closed-loop (FM + Fuzzing) $\rightarrow$ Validation synergy yields the strongest available oracle, reducing the risk of overfit patches that satisfy only an initial, weak test suite.
- **Partial Patch validation:** The Orchestrator must manage the validation of subpatches or partial fixes, a central challenge we previously identified [5]. This demands assertion-based oracles and formal property checking to certify a subpatch's correctness relative to remaining faults, ensuring subsequent repair steps are not invalidated.

9.5 System Dynamics and Implementation Tenets

To address concerns regarding computational cost, we clarify the dynamic and collaborative nature of the orchestrator.

- **Collaboration over Replacement:** The orchestrator embraces collaboration rather than replacement. It orchestrates existing, highly optimized APR tools (e.g., GenProg, TBar, Angelix) and specialized verification engines (formal methods, fuzzing) as cooperative agents within a multi-fault environment. This strategy avoids the prohibitive cost of building a monolithic repair system and leverages decades of single-fault APR research.

- **Dynamic, Context-Aware Execution:** Although the architectural model depicts a complete feedback loop, execution is inherently dynamic. The orchestrator uses the inferred interaction graph $\mathcal{G}_I$ and contextual factors, such as fault criticality, resource budgets, and fault type, to determine the optimal execution path. For example, a low-criticality fault might trigger only a rapid LLM $\leftrightarrow$ APR $\leftrightarrow$ fuzzing sub-cycle, bypassing the costly FM step. Conversely, a synergy fault in a safety-critical system would require the full LLM $\leftrightarrow$ FM $\leftrightarrow$ APR $\leftrightarrow$ fuzzing loop.
- **Enabling Parallelism:** Centralized orchestration enables parallel execution of independent repair cycles. For instance, while the resource-intensive LLM $\leftrightarrow$ FM loop validates a patch for fault f_A, a less costly LLM $\leftrightarrow$ fuzzing sub-cycle can concurrently run targeted interaction tests to refine $\mathcal{G}_I$ for a separate, unmasked fault f_B. This parallelism is essential for managing computational complexity and ensuring practical viability.

This dynamic orchestration ensures computational resources are deployed efficiently, reserving high-assurance validation steps for contexts where fault criticality and interaction complexity justify the additional cost.

9.6 The Iterative Orchestration and Adaptive Control Loop

The orchestrator's primary function is to realize the monotonicity requirement (Definition 3.4) by operating as a continuous feedback system. This systemic approach is necessary to overcome the undecidable complexity of fault classification, ensuring that all decisions are dynamically anchored in the formal constraints of the graph $\mathcal{G}_I$.

The entire system operates in a continuous, monitored cycle designed to maximize data utility. This cycle is formally defined as: Discovery (Fuzzing) $\longrightarrow$ Diagnosis ($\mathcal{G}_I$ Inference) $\longrightarrow$ Synthesis (LLM/APR) $\longrightarrow$ Validation (FM/Fuzzing). Every outcome, including successful patches, failed attempts, and formal counterexamples, is captured and used to refine the orchestrator's planning heuristics and its knowledge base for future iterations.

This iterative process, governed by the established partial order $\prec_I$ and the stringent requirements of the partial and composite oracles, represents a principled architectural evolution for multi-fault APR. It is the logical successor to reductionist single-fault methods, leveraging our formal understanding of theoretical limits and interactions to define a rigorous, forward-looking research agenda for the field.

10 Related Work

Automated program repair builds on decades of research in fault analysis, fault localization, and program transformation. Each of these areas has contributed valuable methods for detecting faults, reasoning about their behavior, and constructing candidate repairs. Yet a persistent gap remains. Existing work often addresses only fragments of the problem, such as identifying fault interactions, isolating defects, or generating candidate fixes, without providing a principled way to integrate these elements into a coherent repair process. This gap becomes especially critical when faults interact, since such interactions can render isolated fixes ineffective or even detrimental. In this section, we review the most relevant efforts in these areas and show how their limitations, simplifying assumptions, and incomplete fault semantics motivate the need for a formal framework and the orchestrated repair paradigm developed in this paper.

10.1 Formal Models of Fault Interaction: The Missing Link for Repair

The complex phenomenon of fault interactions, where one defect's manifestation or repair influences others (e.g., masking or cascading), has been a significant and well-documented challenge in program analysis and dependability research [11, 12, 43]. Evidence strongly shows that disregarding these interdependencies leads to an incomplete understanding of system behavior and severely diminishes the efficacy of analysis techniques. However, previous formal analyses and empirical studies have focused primarily on the detection or categorization of these interactions. Crucially, they have not offered a structured, actionable framework to specifically manage their profound impact on orchestrating multi-fault repairs. That is, they establish that f_i affects f_j, but do not provide the logic for a repair engine to decide when to fix f_i or how to validate the fix for f_j. Our work fills this theoretical void by introducing the formal relation $\mathcal{I}$, a mechanism designed not merely for observation, but to provide precise, repair-guided semantics that inform sequence planning and validation, thereby bridging the gap between interaction analysis and automated repair orchestration.

10.2 Multi-Fault Localization: Providing Locations, Missing Semantics

Considerable research efforts have extended fault localization (FL) from single-fault scenarios to multi-defect configurations, utilizing technologies such as logical inference [2, 36], genetic algorithms [44], and linear programming [10]. These advancements in Multi-Fault Localization (MFL) are crucial for identifying the set of suspicious entities $F = \{f_1, f_2, \ldots, f_n\}$. However, they typically output an unordered set or ranked list of locations without characterizing the semantic relationships between these faults. The output of MFL methods (the nodes of the $\mathcal{I}$ graph) is thus necessary but fundamentally insufficient for multi-fault repair. Repair requires deciding which fault to fix first and what type of patch is required (e.g., a composite fix for synergy). By integrating MFL results as the initial input for constructing our formal graph of fault interactions, our framework can semantically enrich the understanding of these localized faults, transforming a mere list of locations into a principled repair sequence plan.

10.3 Conventional APR: The Failure of the Reductionist Paradigm

Conventional APR tools, such as GenProg [26], Angelix [30], and TBar [28] rely on various techniques (e.g., genetic algorithms, SAT/SMT solving, templates). The unifying and critical assumption among these methods is the reductionist view that multi-fault programs can be fixed by repeatedly applying single-fault repair techniques, effectively assuming fault independence and compositionality. As we formally argue in Section 2.3, this assumption is invalid for the general case of PROG_{MF}. When faults exhibit masking or synergy, a patch for f_i may invalidate the localization for f_j, or a correct partial patch is rejected by the oracle, leading to infinite loops or incorrect termination. Thus, these tools fail on principle for interacting faults, not due to implementation flaws. Our framework is specifically designed to overcome this inherent limitation by introducing an Orchestrator layer that is founded on the $\mathcal{I}$ model, thereby managing the non-compositional behavior of interacting faults.

10.4 LLM-Based Repair: The Necessity of Principled Orchestration

The advent of Large Language Models and pre-trained code models (e.g., CodeBERT [13], DeepFix [19], and generative models) has demonstrated a powerful, flexible capability for code synthesis. These tools often generate fixes without explicit fault localization or templates, and their impressive contextual understanding makes them capable synthesis engines for complex edits. However, as noted in our analysis of fundamental limits, LLMs are statistical models that operate as "black-boxes", lacking a formal mechanism for: (1) Verifying a patch against formal properties (like program equivalence, termination, or semantic correctness), and (2) Systematically guiding the multi-fault repair process based on the logic of fault interaction $\mathcal{I}$. This critical lack of principled control is what necessitates our orchestrator paradigm. Our framework does not seek to replace LLMs but to leverage their high-expressiveness synthesis capability as a component within a structured loop. The orchestrator applies the formal logic derived from $\mathcal{I}$ to manage the LLM's inputs and outputs, providing the necessary semantic scaffold, reasoned sequence planning, and formal assurance that are currently absent in purely data-driven APR methods for complex multi-fault scenarios.

10.5 Formal Methods in Single-Fault APR: Constraints of Specification Availability

The inherent limitations of purely test-driven validation in APR, particularly the risks of overfitting and the absence of strong correctness guarantees, have long motivated the integration of Formal Methods (FM). Prior research has explored the use of FM to constrain the repair search space or enhance validation, primarily by incorporating specifications beyond finite test suites. However, these applications remain fundamentally constrained by two factors: their restriction to the single-fault paradigm and their dependence on the availability of suitable formal input.

- *Reliance on explicit specifications.* Approaches such as [14] employ pre-existing assertions or contract specifications to narrow the search space. While appealing in principle, their practical utility is severely limited by the scarcity and incompleteness of high-quality specifications in real-world codebases.
- *Heuristically inferred invariants.* Other works, including [3, 21], rely on dynamic analysis (e.g., Daikon) to infer likely program invariants. Although this reduces the need for manual specifications, it remains heuristic and incomplete: inferred invariants capture only observed behavior and cannot guarantee the absence of faults on unexecuted paths.
- *Specification-driven repair.* More ambitious methods [34, 40] explore guiding repair directly from formal specifications such as LTL. While this offers the strongest theoretical guarantees, the intractability and cost of inferring or writing full formal specifications for complex legacy code render this approach largely impractical.

In contrast to these efforts, our approach does not assume the existence of comprehensive specifications. Instead, the formal model $\mathcal{G}_I$ functions as a specification derivation engine. It dynamically generates precise, interaction-aware constraints (e.g., LTL patterns for checking masking and unmasking) to guide the orchestrator's validation component. This enables FM techniques to be applied in a targeted, resource-efficient, and effective manner within the $\mathcal{P}_{\text{MF}}$ setting.

11 Conclusion

This work presents the first thorough formal study of automated program repair in the multi-fault setting, framing it as a distinct and principled area of research within software engineering. Our analysis shows that moving beyond single-fault repair is not an incremental extension, but a fundamental shift, bounded by undecidability, shaped by fault interactions, and vulnerable to overfitting.

We contribute three foundational elements to advance this domain. First, a shared formal vocabulary for describing and reasoning about interacting faults. Second, a unified model capturing the space of partial patches and intermediate fixes. Third, a set of evaluation criteria that transcends the narrow question of "does it pass the tests?" toward semantic correctness and long-term robustness. Together, these contributions provide principled architectural scaffolding that must precede the construction of robust empirical tools.

Building on these theoretical foundations, we propose the orchestrated Repair framework as a blueprint for the next generation of multi-fault repair tools. We demonstrate its computational feasibility for core dependency inference, opening a clear agenda for future work: large-scale empirical instantiation, integration of advances in multi-fault localization, and leveraging large language models. These heuristic techniques are promising, but their success depends on the structured, dependency-aware framework and formal oracles established here.

We present this work not as a final implementation, but as a starting point for a broader research direction: to transition multi-fault program repair from an ad hoc practice to a principled, system-level discipline.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan Van Gemund. 2007. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 223–232.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2011. Simultaneous debugging of software faults. *J. Syst. Softw.* 84, 4 (2011), 573–586.

[3] Omar I. Al-Bataineh. 2024. Invariant-based Program Repair. In *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14573)*, Dirk Beyer and Ana Cavalcanti (Eds.). Springer, 255–265.

[4] Omar I. Al Bataineh. 2025. Debugging the Undebuggable: Why Multi-Fault Programs Break Debugging and Repair Tools. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE).* Seoul, South Korea. New Ideas and Emerging Results (NIER) Track.

[5] Omar I. Al-Bataineh. 2025. Towards Interaction-Aware Validation Oracles for Multi-Fault Program Repair. In *Proceedings of the 41st International Conference on Software Maintenance and Evolution (ICSME 2025).* Accepted for publication in the NIER track.

[6] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification.* 184–190.

[7] Dylan Callaghan and Bernd Fischer. 2023. Improving Spectrum-Based Localization of Multiple Faults by Iterative Test Suite Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, René Just and Gordon Fraser (Eds.). ACM, 1445–1457.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qijia Yuan, Urszula Andrychowicz, Adam Mikolajczyk, Piotr Tay, Barret Zoph, Xuanyu Yuan, Alvin Shi, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). https://arxiv.org/abs/2107.03374

[9] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM*, Vol. 7504. 233–247.

[10] Brian C. Dean, William B. Pressly, Brian A. Malloy, and Adam A. Whitley. 2009. A Linear Programming Approach for Automated Localization of Multiple Faults. In *2009 IEEE/ACM International Conference on Automated Software Engineering.* 640–644.

[11] Vidroha Debroy and W. Eric Wong. 2009. Insights on Fault Interference for Programs with Multiple Bugs. In *2009 20th International Symposium on Software Reliability Engineering.* 165–174.

[12] Nicholas DiGiuseppe and James A. Jones. 2011. Fault interaction and its repercussions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM).* 3–12.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020.* Association for Computational Linguistics, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[14] Hadar Frenkel, Orna Grumberg, Bat-Chen Rothenberg, and Sarai Sheinvald. 2022. Automated Program Repair Using Formal Verification Techniques. In *Principles of Systems Design.* Lecture Notes in Computer Science, Vol. 13660. Springer, 511–534.

[15] Ruizhi Gao and W. Eric Wong. 2019. MSeer—An Advanced Technique for Locating Multiple Bugs in Parallel. *IEEE Transactions on Software Engineering* 45, 3 (2019), 301–318.

[16] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[17] Debolina Ghosh and Jagannath Singh. 2021. Spectrum-based multi-fault localization using Chaotic Genetic Algorithm. *Inf. Softw. Technol.* 133 (2021), 106512.

[18] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE)* 41, 12 (2015), 1236–1256. doi:10.1109/TSE.2015.2427842

[19] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI).* 1343–1349. https://www.aaai.org/ojs/index.php/AAAI/article/view/10742

[20] Ye He and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th International Conference on Software Engineering (ICSE).*

[21] Li Huang, Bertrand Meyer, Ilgiz Mustafin, and Manuel Oriol. 2024. Execution-Free Program Repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 517–521.

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA).* ACM, 437–445. doi:10.1145/2610384.2610404

[23] Si-Mohamed Lamraoui and Shin Nakajima. 2016. A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs. *J. Inf. Process.* 24, 1 (2016), 88–98.

[24] Xuan-Bach D Le, David Lo, Claire Le Goues, and Willem Visser. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).* ACM, 163–173.

[25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[26] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* (2012), 54–72.

[27] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65.

[28] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 43–54. doi:10.1145/3293882.3330577

[29] Fan Long, Tushar Sharma, and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 197–211. doi:10.1145/2837614.2837617

[30] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 691–701.

[31] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701.

[32] Fairuz Nawer Meem, Justin Smith, and Brittany Johnson. 2024. Exploring Experiences with Automated Program Repair in Practice. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*. 141–153.

[33] Amirfarhad Nilizadeh, Gary T. Leavens, Xuan-Bach D. Le, Corina S. Păsăreanu, and David R. Cok. 2021. 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, 229–239.

[34] Amirfarhad Nilizadeh, Gary T. Leavens, Xuan-Bach D. Le, Corina S. Păsăreanu, and David R. Cok. 2021. Exploring True Test Overfitting in Dynamic Automated Program Repair using Formal Methods. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 229–240.

[35] OpenAI. 2023. GPT-4 Technical Report. https://openai.com/research/gpt-4. Accessed: 2025-04-23.

[36] Pedro Orvalho, Mikolás Janota, and Vasco M. Manquinho. 2024. cfaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. In *Formal Methods - 26th International Symposium, FM (Lecture Notes in Computer Science, Vol. 14933)*. Springer, 463–481.

[37] Justyna Petke, Matias Martinez, Maria Kechagia, Aldeida Aleti, and Federica Sarro. 2024. 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)*. 452–456.

[38] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Jade Copet, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023). https://arxiv.org/abs/2308.12950

[39] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 23rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 729–739.

[40] Christian von Essen and Barbara Jobstmann. 2015. Program repair without regret. *Formal Methods Syst. Des.* 47, 1 (2015), 26–50.

[41] Sihan Xu, Ya Gao, Xiangrui Cai, Zhiyu Wang, and Hua Ji. 2021. Effective Multi-Fault Localization Based on Fault-Relevant Statistics. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 998–1003.

[42] Sihan Xu, Ya Gao, Xiangrui Cai, Zhiyu Wang, and Hua Ji. 2021. Effective Multi-Fault Localization Based on Fault-Relevant Statistics. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 998–1003.

[43] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML)*, Vol. 148. 1105–1112.

[44] Yan Zheng, Zan Wang, Xiangyu Fan, Xiang Chen, and Zijiang Yang. 2018. Localizing multiple software faults based on evolution algorithm. *J. Syst. Softw.* 139 (2018), 107–123.