Beyond Static Corpora

Toward Adaptive, Specification-Driven Program Repair Evaluation

ABSTRACT

Automated Program Repair (APR) has rapidly advanced with the emergence of Large Language Models (LLMs), and modern repair systems increasingly achieve high success rates on established benchmarks — raising concerns about evaluation saturation and distributional overfitting. This paper argues that the dominant paradigm of static benchmark evaluation is structurally inadequate, and that scaling static datasets cannot resolve this inadequacy. We propose a paradigm shift toward adaptive, specification-driven benchmark generation governed by five organizing principles: generative unboundedness, specification primacy, deterministic certification, adaptive coverage, and oracle independence. We formalize these principles, develop a taxonomy of the dimensions along which repair instances vary, and argue that the generator–verifier separation is the architectural consequence that makes the framework trustworthy. We treat the oracle problem as a conceptual issue in its own right, examine the tradeoffs among available correctness criteria, and identify the open problems the framework surfaces but does not resolve.

KEYWORDS

Automated Program Repair, Adaptive Benchmarks, Evaluation Methodology, Large Language Models, Benchmark Generation

1 Introduction

Automated Program Repair (APR) has undergone a profound transformation with the integration of Large Language Models (LLMs). Early APR techniques relied on generate-and-validate search strategies and manually designed patch templates, exploring predefined edit spaces to synthesize candidate fixes. In contrast, modern systems leverage pretrained language models capable of directly generating semantically rich patches across multiple languages and domains. This shift from search over handcrafted transformation spaces to neural and LLM-based generation has led to dramatic improvements in repair performance, with recent systems achieving unprecedented success rates on established benchmarks (Ma et al. 2025).

However, as repair capabilities have advanced, evaluation practices have remained largely static. APR research continues to depend on fixed benchmark datasets such as Defects4J (Just et al. 2014), BugsInPy (Tufano et al. 2019), and SWE-bench (Jimenez et al., 2024). These collections have driven progress for years, providing common ground for comparing techniques and establishing empirical baselines. Yet they share fundamental structural limitations that threaten the validity and sustainability of APR evaluation (Ebrahim 2025).

The dominant response to these limitations has been to scale benchmark size—creating ever-larger collections in hopes of forestalling saturation. SWE-bench, for instance, contains over 2,000 real-world GitHub issues, dwarfing earlier benchmarks. Yet this approach merely postpones the underlying problem. As LLMs continue to improve, merely expanding dataset size proves insufficient; larger static datasets inherit the same structural rigidity as smaller ones. They capture a single snapshot of the bug distribution landscape, freeze it in time, and eventually succumb to the same saturation dynamics.

This paper argues that evaluation must evolve from static artifact collections to adaptive generation processes. We propose a paradigm shift toward adaptive, specification-driven benchmark pipelines—systems that synthesize validated repair instances on demand based on declarative specifications. Rather than treating benchmarks as fixed resources to be consumed, this approach treats them as programmable instruments capable of generating infinite, controlled variations for rigorous experimentation.

2 The Structural Inadequacy of Static Benchmarks

The instinct to address evaluation saturation by building larger datasets is understandable and has produced valuable resources. However, scaling static benchmarks encounters fundamental constraints that cannot be overcome through size alone. This section makes the theoretical argument for why this is so — not merely that saturation happens, but that it must happen given the nature of static datasets and the combinatorial structure of the program repair problem.

2.1 Combinatorial Incompleteness

No static dataset, regardless of size, can comprehensively represent the combinatorial space of program repair challenges. The dimensions of variation are independently structured: programming languages, each with distinct idioms and error patterns; fault types ranging from semantic errors and security vulnerabilities to concurrency bugs and API misuse; dependency structures spanning single-file bugs and complex multi-file build systems; project scales from small pedagogical examples to large enterprise systems; domain semantics specific to web applications, systems programming, data science, and embedded systems; and context requirements determining whether a bug is reproducible in

isolation or requires specific runtime environments and execution histories.

The number of combinations across these dimensions is effectively infinite. Even a benchmark with 10,000 instances samples this space sparsely. Increasing to 100,000 instances improves coverage marginally but does not change the fundamental sampling problem. This is not an empirical observation about any particular benchmark — it is a structural claim about the relationship between finite sets and combinatorial spaces. No curatorial effort, however sustained, can overcome it (Liu et al. 2021).

2.2 Historical Contingency

Static benchmarks do not sample the instance space randomly — they sample it through the historically contingent process of collecting bugs from popular open-source projects during specific time windows. Their distributional properties reflect the accidents of open-source development history rather than the structure of the repair problem itself. When the APR community optimizes for performance on Defects4J, it implicitly optimizes for that benchmark's specific distribution of fault types and project characteristics — a distribution that reflects curation choices and repository selection, not any inherent property of software faults generally. This is the evaluation equivalent of Goodhart's Law: when a measure becomes a target, it ceases to be a good measure (Campos et al. 2025).

Increasing the size of a historically contingent sample does not make it less contingent; it makes a biased sample larger. The contamination risks that plague modern benchmark evaluation are a direct consequence of this contingency: public benchmarks derived from popular open-source repositories are precisely the kind of data used to train code models, and high repair performance on such benchmarks may reflect memorization rather than genuine repair capability.

2.3 The Illusion of Progress

Perhaps most consequentially, static benchmarks create an illusion of progress that may not reflect genuine advances in repair capability. As models saturate existing benchmarks, researchers report ever-higher scores — yet these improvements may represent distributional overfitting rather than enhanced generalization. The field risks optimizing for leaderboard position while losing sight of the broader goal: building repair systems that work robustly across the vast and varied space of real-world bugs (Smith et al. 2015).

Static benchmarks will remain valuable as historical references and points of community consensus. But as the primary instrument for measuring progress, they are structurally insufficient — and no amount of scaling changes that.

3 Precedents for Generative Evaluation

The program repair community is not the first to confront the limitations of static test corpora. Three adjacent traditions — coverage-guided fuzzing, property-based testing, and random compiler testing — have each made the transition from static artifacts to adaptive generation, and the lessons they offer are directly applicable to benchmark design for APR evaluation.

Fuzz testing originally relied on static test corpora, but evolved toward generation-based approaches in which declarative configurations produce large, diverse input spaces dynamically. The critical innovation was coverage guidance: rather than generating inputs independently of what has already been tested, coverage-guided fuzzers monitor which program paths have been exercised and direct generation pressure toward unexplored regions. This feedback loop — from the artifact under test back to the generator — is what makes the approach adaptive rather than merely generative. The result is exploration of behavioral regions that static corpora never reach, regardless of their size.

Property-based testing frameworks such as QuickCheck made a complementary move: rather than enumerating test cases, they generate them from formal specifications of intended behavior. A specification describes a property that must hold universally — a universally quantified claim about the relationship between inputs and outputs — and the framework samples inputs automatically. The space of generated tests is effectively infinite; the specification is finite. This is the foundational insight that the adaptive benchmark framework inherits: a finite declarative description can generate an infinite instance space, provided the generation mechanism is sufficiently expressive.

Random compiler testing, exemplified by tools like Csmith, applies generative exploration to find bugs in compilers by producing random programs that cover vast behavioral regions no curated test suite could enumerate. The empirical finding that random generation reveals bugs missed by years of manual testing is a direct argument against the assumption that sufficiently large static corpora are adequate.

These three traditions represent instances of the same underlying insight: evaluation quality is a function of coverage and adaptivity, not of corpus size. Program repair evaluation has not yet made the transition that these adjacent fields made years or decades ago. The framework proposed in this paper is the application of that insight to the APR domain — and the analogy between coverage-guided fuzzing and the adaptive benchmark pipeline is more than illustrative. Both architectures solve the same fundamental problem: how to explore a large, structured space efficiently under a budget constraint, using feedback-driven sampling as the mechanism.

4 A Taxonomy of Repair Instance Variation

To argue that adaptive generation is both necessary and achievable, we must first establish that the space of program repair instances is structured — that it has dimensions along which instances vary independently, and that those dimensions can be described declaratively. We define an adaptive benchmark instance as a tuple:

B = (C, F, S, P, V, D)

where each component represents an independently necessary dimension of variation. The claim that these six dimensions are independently necessary is the taxonomy's central theoretical contribution — each dimension captures something about the instance space that cannot be reduced to any other.

4.1 Program Context (C)

The program context captures the environmental and structural conditions in which a bug is situated. A repair system operating on a self-contained Python function faces fundamentally different demands than one addressing a bug embedded in a multi-file Java service with transitive Maven dependencies — even if the logical fault is identical. Context is not merely background information; it determines what a repair system must know before any fault-specific reasoning begins. An evaluation framework that does not vary context systematically cannot measure whether repair systems generalize across environments.

4.2 Fault Model (F)

The fault model characterizes the semantic nature of the defect. Fault type is arguably the most consequential axis of variation in program repair evaluation, because different fault classes require qualitatively different reasoning strategies. A buffer overflow demands knowledge of memory layout and pointer arithmetic; an off-by-one error requires precise boundary reasoning; a race condition demands understanding of concurrent execution semantics. Existing benchmarks are heavily skewed toward semantic and logical errors appearing in popular open-source Java and Python projects, leaving security vulnerabilities, concurrency bugs, and performance issues substantially underrepresented relative to their prevalence in industrial software. A framework that does not make fault type an explicit, controllable dimension cannot correct this skew.

4.3 Seeding Strategy (S)

The seeding strategy governs how faulty instances are constructed — whether by mutating correct programs, extracting real bugs from version control histories, synthesizing faults from scratch, or leveraging LLMs to generate realistic fault patterns. The choice of strategy has significant consequences for the realism, diversity, and evaluability of the resulting instances: extractive strategies preserve ecological validity but inherit the historical biases of their source repositories, while generative strategies offer greater distributional control at potential cost to naturalness. A well-designed framework should be agnostic to seeding strategy, treating it as a dimension that can be varied and controlled rather than a fixed methodological commitment.

4.4 Patch Constraints (P)

Patch constraints delimit the space of repairs considered valid. Without explicit constraints, the repair space is unbounded: a system could trivially satisfy a test suite by deleting all code, or pass a type checker by inserting unconstrained casts. Patch constraints operationalize the intuition that a correct repair must be not only functionally adequate but structurally proportionate — reflecting the norms of software engineering practice in which patches are expected to be minimal, readable, and semantically coherent with the surrounding code. Making patch constraints explicit is also a prerequisite for meaningful difficulty calibration, since the hardness of a repair problem depends partly on how constrained the acceptable solution space is.

4.5 Validation Oracle (V)

The validation oracle specifies how correctness is determined. Crucially, correctness applies at two distinct points in the pipeline with different purposes: at generation time, when the framework decides whether a candidate instance is admissible, and at evaluation time, when the framework decides whether a repair system's proposed fix is correct. Both must be configured independently, as the appropriate standard for judging repair outputs is not necessarily the same as the standard used to admit instances. The available oracle types — test-suite passage, static analysis, differential testing, property checking, and hybrid combinations — represent positions on a tradeoff between tractability and rigor. Section 7 treats this tradeoff as a conceptual problem in its own right.

4.6 Difficulty Profile (D)

The difficulty profile controls the hardness of individual instances along multiple orthogonal axes. Difficulty is not a single scalar property: an instance may be syntactically simple but semantically subtle, involve minimal code changes but require extensive contextual reasoning, or be easy for symbolic techniques while resisting neural approaches. Making difficulty parameters explicit serves two purposes. It allows benchmark designers to deliberately construct benchmarks with calibrated complexity distributions rather than accepting whatever distribution the seeding strategy naturally produces. And it enables fine-grained analysis of repair system capabilities across the hardness spectrum, revealing performance profiles that aggregate success rates obscure.

Together, these six dimensions define a space that is both tractable — it can be described declaratively in a finite specification — and effectively infinite — it cannot be exhaustively sampled. This combination is what makes adaptive generation both necessary and achievable.

5 Principles of Adaptive Benchmark Generation

The structural inadequacy identified in Section 2 and the precedents established in Section 3 jointly imply a set of properties that any adequate evaluation framework must have. This section articulates those properties as five organizing principles. Each principle is derived from the inadequacies of the static paradigm and addresses a specific failure mode that scaling alone cannot correct. Together they define the conceptual foundation of the adaptive benchmark framework.

5.1 Generative Unboundedness

An adequate evaluation framework must be capable of producing an effectively infinite instance space from a finite declarative specification. This principle follows directly from the combinatorial incompleteness argument in Section 2: if the space of repair instances is combinatorially vast and no finite dataset can sample it adequately, then the generation mechanism must be capable of producing instances without exhaustion. This distinguishes the proposed framework from any approach that produces a fixed dataset — including approaches that produce large fixed datasets. The generation mechanism is not a dataset constructor; it is a sampler over a structured, effectively unbounded space.

5.2 Specification Primacy

All framework behavior — what instances are generated, what correctness criteria are applied, what distributional targets are pursued — must be derivable from a single declarative contract: the specification. This principle ensures reproducibility, auditability, and the separation of research intent from implementation detail. Two researchers who share a specification should be able to independently generate equivalent benchmarks. A specification is not a benchmark; it is a generative program whose execution produces a benchmark. This distinction is consequential: where a static benchmark is a fixed artifact that can be exhausted, memorized, or contaminated, a specification is a renewable resource that can produce arbitrarily many distinct instantiations while remaining faithful to the researcher's intent.

5.3 Deterministic Certification

Correctness guarantees for benchmark instances must not depend on probabilistic components. Generation may be probabilistic — and should be, to achieve diversity — but validation must be deterministic, producing the same verdict for any fixed instance and specification unconditionally. This principle is what allows the framework to use LLMs as generators without compromising the integrity of the benchmark set they produce. An LLM is capable and creative but unreliable; a deterministic verifier catches its failures mechanically. The trust that researchers place in the benchmark derives not from the reliability of the generator but from the independence and determinism of the verifier.

5.4 Adaptive Coverage

Generation must respond to the distributional properties of the benchmark set under construction, steering toward underrepresented regions of the specification space rather than proceeding independently of what has already been produced. This principle is the direct analog of coverage guidance in fuzzing: just as a coverage-guided fuzzer monitors which program paths have been exercised and directs mutation pressure toward unexplored paths, an adaptive benchmark pipeline monitors which regions of the specification space have been populated and directs generation pressure toward underrepresented regions. Without this feedback mechanism, a generative pipeline is not adaptive — it is merely generative, and subject to the same natural clustering and distributional biases as any unconstrained sampling process.

5.5 Oracle Independence

The criteria used to certify that a benchmark instance is admissible must be specified independently of the criteria used to judge whether a repair system's proposed fix is correct. These are distinct questions answered at distinct points in the pipeline's lifecycle, and conflating them produces evaluation frameworks whose correctness standards are ambiguous and whose results are difficult to interpret. Oracle independence also enables a crucial form of flexibility: the same benchmark set can be evaluated under different correctness criteria for different research purposes, provided those criteria are declared in the specification rather than embedded in the benchmark artifacts themselves.

6 The Generator–Verifier Separation

The five principles articulated in Section 5 jointly imply a specific architectural relationship between the components responsible for producing instances and the components responsible for certifying them. The generator–verifier separation is not a design decision imposed from outside the framework — it is a logical consequence of the deterministic certification and adaptive coverage principles. If certification must be deterministic and generation must be exploratory and probabilistic, they cannot be the same component.

The asymmetry between the two components is intentional and load-bearing. A system in which the same component both produces and validates instances cannot provide independent quality guarantees: if the generator is biased or mistaken, a validator operating on the same principles will share its blind spots. When generation and verification are implemented independently — and when verification is constrained to deterministic, mechanically executable checks — the verifier provides genuine independence from the generator's assumptions. This independence is what makes the framework trustworthy, regardless of what generation technique is used.

6.1 The Generator

The generator is the creative engine of the framework — the component responsible for exploring the space of possible benchmark instances and producing candidates that are diverse, realistic, and aligned with the active specification. Because coverage and naturalness are the primary goals at this stage, the generator is unconstrained in its methods: it may apply rule-based mutation, sample from version control histories, or leverage large language models to synthesize novel fault patterns. The outputs of the generator carry no intrinsic guarantee of correctness; they are proposals, not verdicts.

The generator must balance three competing pressures. Fidelity requires that generated instances resemble real-world bugs in their surface form, reasoning requirements, and fix complexity. Diversity requires that the generator not gravitate toward the easiest or most common patterns within a fault type. Specification

adherence requires that all generated instances fall within the dimensional constraints declared in the active specification. Managing these pressures simultaneously — particularly when using LLMs as generators, which have strong tendencies toward common patterns — is one of the central design challenges of the framework.

One output of the generator warrants particular attention: the candidate fix. Its purpose is not to provide a unique correct answer — there may be many valid repairs for a given bug — but to confirm that the instance is not pathological. An instance for which no repair can be found, or for which the only repair requires changes far outside the specified patch constraints, should not enter the benchmark set regardless of how realistic the bug appears. The candidate fix therefore functions as a satisfiability witness: evidence that the instance is solvable under the specified constraints, not a ground-truth solution against which evaluated systems are compared.

6.2 The Verifier

Where the generator is permissive by design, the verifier is uncompromising. It applies a fixed, deterministic pipeline to every candidate instance, admitting only those that satisfy all specified correctness criteria. The verifier must never incorporate probabilistic components — its value derives entirely from the reproducibility and auditability of its judgments.

The verifier's determinism has a precise meaning. It does not mean the verifier is simple or limited; a deterministic verifier can execute complex static analyses, measure code coverage to arbitrary precision, and apply sophisticated duplication detection. It means that for any fixed candidate instance and any fixed specification, the verifier always produces the same verdict. This property ensures that the benchmark's contents are fully determined by the specification and the random seed, with no dependence on the internal state of any probabilistic model — and that any researcher holding the specification can independently reproduce the verification results.

The verification pipeline checks syntactic validity, confirms that tests fail on the buggy version, confirms that tests pass on the reference fix, enforces patch constraints, measures test suite coverage, detects near-duplicate instances, screens for contamination against known training corpora, and validates that dimensional metadata is consistent with the specification. Every admitted instance is accompanied by a complete verification log; every rejected candidate produces a rejection record specifying the stage and reason for failure. Rejection records are diagnostic data, not merely discarded output — they reveal systematic generator failures and infeasible specification regions that would otherwise be invisible.

6.3 The Benefits of Separation

The separation between generator and verifier produces five concrete benefits that neither component could provide alone. Trust through verification: even if the generator produces flawed or biased instances, the verifier catches them, decoupling the trustworthiness of the benchmark from the reliability of any particular generation technique. Generative flexibility: because the verifier provides an independent backstop, the generator is free to be ambitious — attempting difficult fault types and rare configurations without the conservative bias that coupled generation-validation would require. Deterministic guarantees: two independent executions of the pipeline with the same specification and seed produce the same benchmark set, making specifications shareable and benchmarks reconstructible. Auditability: the verification log provides a complete evidentiary record of how each instance was evaluated. Generator-agnosticism: because the verifier's criteria are defined by the specification rather than the generator's internal logic, instances produced by any seeding strategy pass through identical verification stages, ensuring internal consistency across mixed-origin benchmark sets.

7 The Oracle Problem

The oracle problem in program repair evaluation is more conceptually difficult than it first appears, and it deserves treatment as an intellectual issue rather than merely an implementation choice. At its core, the problem is this: what does it mean for a repair to be correct? The answer is not obvious, and the choices made in answering it have significant consequences for what evaluation results mean and what they do not mean.

Program repair correctness is irreducibly multi-dimensional. A repair may be behaviorally correct on tested inputs, behaviorally equivalent to the reference fix on all inputs, structurally appropriate given the surrounding code, consistent with the program's formal specifications, and conformant with the patch constraints declared in the benchmark specification — and these criteria are mutually independent. A repair system that satisfies one criterion may fail others. An evaluation that measures only one criterion is not measuring correctness; it is measuring a proxy, and the relationship between the proxy and genuine correctness depends on assumptions that are rarely stated.

7.1 Generation-Time and Evaluation-Time Oracles

The oracle independence principle established in Section 5 requires that correctness be assessed separately at two distinct points in the framework's lifecycle. At generation time, the oracle governs whether a candidate instance is admissible into the benchmark set. At evaluation time, it governs whether a repair system's proposed fix is correct. These are different questions with different consequences, and conflating them — as most existing frameworks implicitly do — produces ambiguity that compromises both the quality of the benchmark and the interpretability of evaluation results.

Generation-time oracles are applied by the verifier and must be deterministic. Their purpose is quality assurance: confirming that every admitted instance is well-formed, fault-sensitive, solvable, and sufficiently exercised by its associated test suite. Evaluation-

time oracles are applied to repair system outputs and define what the benchmark is actually measuring. The same oracle type — test-suite passage, differential testing, property checking — can appear at either stage, but its role and its consequences differ depending on when it is applied.

7.2 The Correctness Criteria and Their Tradeoffs

Test-suite passage is the baseline evaluation-time criterion and the standard adopted by dominant benchmarks in the field. It is fully automatable and domain-agnostic, but it is subject to a well-documented failure mode: a fix may pass all tests by overfitting to test inputs rather than resolving the underlying fault. Test-suite passage measures whether a repair satisfies the tests that were written, not whether it is correct in any deeper sense. The strength of this criterion therefore depends entirely on the strength of the test suite — and weak test suites, which are common in practice, make test-suite passage a poor proxy for genuine correctness.

Differential testing is a more stringent criterion that compares the behavior of the proposed fix against the reference fix across a generated input set, rather than a fixed test suite. This oracle tolerates structural and syntactic divergence between the proposed and reference fixes while requiring behavioral equivalence — which is the appropriate standard for a domain where many valid repairs exist and penalizing them for differing from the reference fix is unjustifiable. Differential testing is more appropriate than test-suite passage when the benchmark targets fault types where alternative repairs are common and test overfitting is a primary concern.

Property checking, the most stringent criterion, evaluates proposed fixes against formal invariants or contracts that the repaired program must satisfy unconditionally. It is independent of any particular reference fix and provides the strongest correctness guarantee — but it requires that invariants be specifiable for the instance's code domain, which is not always possible, and it is the most expensive to evaluate. Property checking is most appropriate for security-critical or safety-critical repair scenarios where the space of acceptable repairs is intentionally narrow.

7.3 The Role of the Reference Fix

The reference fix produced during generation occupies an ambiguous position at evaluation time. It is a satisfiability witness — evidence that the instance is solvable — but it is not a unique ground truth. There are typically many valid repairs for any given bug, and treating the reference fix as the sole correctness standard penalizes valid alternative repairs while rewarding repairs that mimic the reference without being genuinely correct.

The reference fix's legitimate evaluation-time roles are limited to three: serving as the behavioral reference for differential testing, providing an edit-distance upper bound for difficulty calibration, and functioning as a sanity check for suspiciously divergent proposed fixes. It must never be provided to the repair system under evaluation — doing so trivializes the task — and it must never be the sole evaluation-time criterion in a way that forecloses valid alternatives. The oracle independence principle is precisely what creates the conceptual space to use the reference fix appropriately: by separating the admissibility criterion from the evaluation criterion, it allows the reference fix to serve its legitimate functions without distorting the evaluation.

7.4 Reporting Outcomes

A binary correct/incorrect verdict conflates three outcomes that have different implications for understanding repair system behavior: a fix that satisfies the oracle, a fix that fails to repair the bug, and a fix that repairs the bug but violates the structural or semantic constraints specified in the patch constraints dimension. The first failure mode indicates a localization or repair capability failure; the second indicates that the system generates valid but over-permissive repairs that do not respect engineering norms. Collapsing these into a single failure category destroys diagnostic information. Evaluation frameworks should report all three outcomes separately, providing the multi-dimensional correctness profile that characterizes where repair systems succeed and where they fall short.

8 Implications and Open Problems

The framework's principles have implications for how the program repair research community designs, shares, and interprets evaluations — and they surface several open problems that the framework does not resolve.

8.1 Implications for Evaluation Practice

If specification primacy is taken seriously, benchmark sharing changes in character. Rather than sharing benchmark files — which are finite, can be contaminated, and become stale — researchers share specifications, which are renewable, verifiable, and fully transparent about the intent behind the benchmark. A specification makes explicit what a static benchmark leaves implicit: the distributional assumptions, the oracle criteria, the difficulty targets, and the patch constraints that governed instance selection. This transparency enables a kind of scientific accountability that static benchmarks cannot provide.

The adaptive coverage principle implies that benchmark quality should be assessed not by instance count but by coverage of the specification space. A benchmark with 1,000 instances that covers all six dimensions uniformly is a better evaluation instrument than a benchmark with 10,000 instances clustered in the same distributional region. This reorients the community's implicit understanding of what makes a benchmark good.

The oracle independence principle implies that evaluation results should always be accompanied by an explicit statement of the oracle criterion applied, and that comparisons across evaluations using different oracle criteria are not meaningful without adjustment. The current practice of reporting aggregate success rates without specifying oracle details conflates results that are not comparable.

8.2 Open Problems

The framework surfaces three open problems that it does not resolve, and honesty about their difficulty is important for assessing the contribution accurately.

The ecological validity problem asks whether generated instances are as meaningful as real-world bugs. Extractive instances from version control histories have high ecological validity but inherit historical biases; generative instances offer distributional control but may not reflect the patterns that matter in practice. The framework provides the architectural space to vary seeding strategy as a dimension, but it does not provide a criterion for deciding which instances are ecologically valid — that judgment remains with the researchers who write specifications.

The specification authorship problem asks who writes specifications and how their quality is assessed. A framework whose output quality depends on specification quality shifts the burden of benchmark curation from instance selection to specification design. This is a meaningful improvement — specifications are more transparent and reproducible than curatorial decisions — but it is not a complete solution. Poorly designed specifications produce poorly designed benchmarks, and the framework provides no mechanism for detecting specification-level quality failures beyond the feasibility and consistency checks that can be automated.

The oracle completeness problem asks whether any finite set of oracle criteria can fully capture the correctness standard appropriate to a given repair domain. Test-suite passage, differential testing, and property checking each capture something about correctness, but none captures everything. A repair that passes all three criteria may still be wrong in ways that no oracle in the framework measures. The framework makes oracle choice explicit and separable, which is a significant conceptual advance — but it does not solve the underlying epistemological problem of what program correctness ultimately means.

8.3 What the Framework Trades Away

The framework's advantages come with genuine tradeoffs that should be acknowledged. Generative unboundedness trades ecological validity for coverage and control: a specification-driven generator can produce instances in regions of the fault space that no real-world project has yet exhibited, which is both a strength and a source of questions about whether such instances matter. Specification primacy trades the simplicity of a fixed artifact for the expressiveness of a declarative language, which introduces a new form of expertise requirement — specification authorship — that the framework does not make easy. Deterministic certification trades the creative range of LLM-based validation for the mechanically guaranteed consistency of rule-based checking, which may not always be achievable for the most complex fault types.

9 Conclusion

The rapid progress of LLM-based Automated Program Repair has exposed structural limitations in static benchmark evaluation that cannot be resolved by making static benchmarks larger. The combinatorial incompleteness of finite datasets, the historical contingency of their distributional properties, and the contamination risks inherent in public benchmarks derived from open-source repositories all point to the same conclusion: the field requires not larger benchmarks, but a different kind of benchmark.

We have argued that the future of APR evaluation lies in adaptive, specification-driven benchmark generation governed by five principles: generative unboundedness, specification primacy, deterministic certification, adaptive coverage, and oracle independence. These principles are not a wish list — they are the theoretical consequences of taking seriously what evaluation requires. The generator–verifier separation follows from these principles as a structural necessity, and the oracle problem demands treatment as a conceptual issue rather than an implementation detail.

Static benchmarks will remain valuable as historical references and points of community consensus. But the frontier of APR research — understanding generalization, probing failure modes, measuring capability across the vast and varied space of real-world bugs — requires the programmability and adaptability that only specification-driven, adaptive generation can provide. The framework proposed here is not a finished system; it is a set of principles that any adequate system must embody, and a conceptual vocabulary for reasoning about what evaluation requires in an era when repair systems are becoming more capable than the benchmarks designed to measure them.

REFERENCES

[1] Y. Ma, et al., "BloomAPR: A Bloom's Taxonomy-based Framework for Assessing the Capabilities of LLM-Powered APR Solutions*," arXiv preprint arXiv:2509.25465, 2025*.

[2] 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 '14)*. Association for Computing Machinery, New York, NY, USA, 437–440.

[3] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin Monperrus, and Denys Poshyvanyk. 2019. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM,* 567–578.

[3] Carlos Jimenez, John Yang, Tony Zhang, et al. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *Proceedings of the International Conference on Learning Representations (ICLR).*

[4] Yasser Ebrahim, Making the Case for LLM-Generated Automated Program Repair Benchmarks, *2025 IEEE/ACIS 29th SNPD, Busan, Korea, Republic of, 2025,* pp. 842-847

[5] Edward K. Smith, Earl T. Barr, and Claire Le Goues. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM,* 532–543.

[6] Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2024. Investigating Data Contamination in Modern Benchmarks for Large Language Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 8706–8719, Mexico City, Mexico. Association for Computational Linguistics

[7] Viola Campos, Ridwan Shariffdeen, Adrian Ulges, and Yannic Noller. 2025. Empirical Evaluation of Generalizable Automated Program Repair with Large Language Models. In *arXiv preprint*.

[8] Liu et al. (2021). "A critical review on the evaluation of automated program repair systems." *Journal of Systems and Software*, 171, 110817

[9] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (November 2021), 2312–2331.