Beyond Reproduction: Uncovering Latent Performance Regressions with LLM-Guided Fuzzing (Work In Progress Paper)

Renming Zheng
Stevens Institute of Technology
Hoboken, NJ, USA
rzheng3@stevens.edu

Yutong Zhao
California State University, Long Beach
Long Beach, CA, USA
Yutong.Zhao@csulb.edu

Lu Xiao
Stevens Institute of Technology
Hoboken, NJ, USA
lxiao6@stevens.edu

Weiyi Shang
University of Waterloo
Waterloo, ON, Canada
wshang@uwaterloo.ca

Lizhi Liao
University of Guelph
Guelph, ON, Canada
lizhi.liao@uoguelph.ca

Abstract

Performance regressions are notoriously difficult to detect due to their silent nature and dependency on complex, structure-specific input conditions. While state-of-the-art performance fuzzers effectively maximize execution path lengths, they often lack semantic direction, wasting significant resources probing stable regions. In this paper, we propose *Issue-Driven Performance Fuzzing*, which posits that a reported regression is rarely an isolated incident but a visible symptom of a broader cluster of latent vulnerabilities. To operationalize this, we introduce an automated framework utilizing LLMs to bridge the semantic gap between unstructured issue reports and executable fuzzing seeds. By extracting the essence of a historical bug to synthesize targeted mutation strategies, our approach explores the bug's structural neighborhood for hidden defects. We evaluated this framework on *Apache PDFBox*, where it successfully isolated latent degradation factors and revealed resilience regressions, i.e., the instances where optimized versions exhibit higher algorithmic fragility. In total, our framework generated 22 valid mutation strategies and identified 2 previously unknown performance weaknesses.

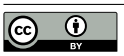
CCS Concepts

• **Software and its engineering → Software performance**; **Software defect analysis**; *Software evolution*; *Maintaining software*; • **Computing methodologies → Information extraction**.

Keywords

Performance Testing, Fuzzing, Large Language Models, Regression Reproduction, Issue Reports, Software Maintenance

ACM Reference Format:
Renming Zheng, Yutong Zhao, Lu Xiao, Weiyi Shang, and Lizhi Liao. 2026. Beyond Reproduction: Uncovering Latent Performance Regressions with LLM-Guided Fuzzing (Work In Progress Paper). In *Companion of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE Companion '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3777911.3801111



ICPE Companion '26, Florence, Italy

ACM ISBN 979-8-4007-2326-1/2026/05
https://doi.org/10.1145/3777911.3801111

1 Introduction

Performance regressions are notoriously difficult to detect due to their silent nature and dependence on specific structural configurations [1, 2]. While automated techniques like *PerfFuzz* [3] and *SlowFuzz* [4] effectively generate regression inputs [5], they suffer from a lack of *semantic direction*. Recent advancements integrating Large Language Models (LLMs) with fuzzing face similar limitations. For example, recent works utilize LLMs either to infer programmatic constraints for performance testing, such as *WEDGE* [6], or to drive universal input generation across diverse languages, such as *Fuzz4All* [7]. Although these LLM enhanced techniques excel at exploring general program boundaries or syntax spaces, they operate without targeted guidance. By treating software primarily as a black box, both traditional and LLM enhanced tools waste resources probing stable regions, missing complex regressions that require highly structured preconditions [8, 9].

We argue that project history offers the missing map for guiding performance testing [10, 11]. To leverage the rich semantic data in issue trackers, we propose *Issue Driven Performance Fuzzing*. Our rationale is grounded in the defect clustering principle: the observation that a single visible defect rarely exists in isolation, but rather indicates a concentration of structural weaknesses. A reported regression often signals a fragile functional module; by systematically mutating the structural characteristics of a known issue, we can uncover the underlying cluster of latent degradation vectors.

To operationalize this, we introduce a framework using LLMs to bridge the gap between textual reports and executable fuzzing seeds. Moving beyond random bit-flipping, our approach leverages LLMs to extract the "DNA" of a bug—its triggering logic and constraints—to synthesize *Targeted Mutation Strategies*. This effectively converts a single historical report into a comprehensive, structure-aware stress test suite.

We evaluated this framework on *Apache PDFBox*. Starting from a single report on font extraction (*PDFBOX-959* [12]), we not only reproduced the original regression but also discovered latent performance costs in Annotation objects. Furthermore, our analysis

revealed a "Resilience Regression", where the newer, supposedly optimized version exhibited higher algorithmic fragility under synthesized stress.

In summary, this paper makes three contributions:

- **Concept:** We adapt the defect clustering principle for performance testing, using historical reports as anchors to systematically explore and locate clusters of latent defects.
- **Framework:** We present an automated, LLM-driven pipeline that transforms natural language reports into validated performance mutants.
- **Findings:** Our evaluation on Apache PDFBox generated 22 valid strategies and discovered 2 new weaknesses, including confirmed resilience regressions.

2 Approach

We propose an automated framework that leverages historical issue reports to generate performance fuzzing scenarios. As illustrated in Figure 1, our pipeline transcends simple reproduction by exploiting the semantic knowledge embedded in these reports to infer latent performance degradation factors. By utilizing Large Language Models (LLMs) to guide mutations within the structural neighborhoods of known bugs, we uncover degradation vectors that traditional, unguided fuzzing would miss.

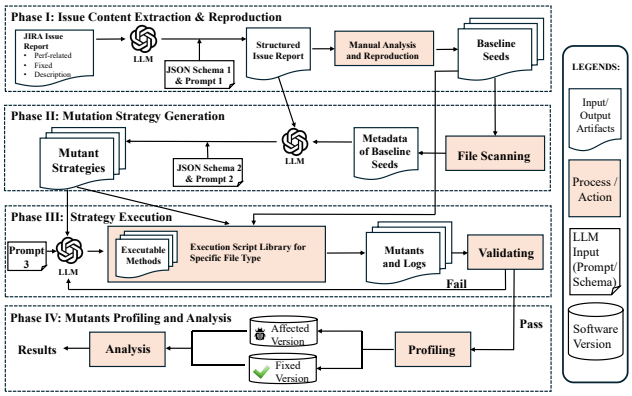


Figure 1: Overview of the Proposed Workflow.

Formally, the framework takes a historical issue report, denoted as $\mathcal{I}$, as input. From $\mathcal{I}$, we derive a set of validated baseline seeds $\mathcal{S}$. We define the testing context as a pair of software versions, $\mathcal{V} = \{V_{\text{affected}}, V_{\text{fixed}}\}$. Our core objective is to synthesize a set of mutants $\mathcal{M}$ derived from $\mathcal{S}$ that maximizes the probability of triggering performance degradation in V_{affected} relative to V_{fixed}. The pipeline is structured into four distinct phases, transitioning from natural language reports to validated performance regressions:

2.1 Phase I: Issue Content Extraction and Reproduction

This phase bridges the semantic gap between unstructured natural language issue reports and executable test configurations. As illustrated in Figure 1, the output is a set of validated *Baseline Seeds* ($\mathcal{S}$) that deterministically trigger the reported performance degradation, ensuring subsequent fuzzing phases are rooted in high-fidelity regression conditions.

This process begins with ingesting *JIRA Issue Reports* and filtering them based on three essential criteria: (1) *Performance Relevance*: issues describing efficiency degradation (e.g., latency, memory leaks) rather than functional failures; (2) *Reproducibility Context*: reports containing sufficient context on triggering inputs or environments; and (3) *Fix Availability*: resolved issues with accessible fixed versions, enabling differential analysis between affected (V_{affected}) and fixed (V_{fixed}) states.

LLM-Driven Content Extraction. Raw issue reports ($\mathcal{I}$) often contain conversational noise. To normalize this data, we utilize an LLM to map $\mathcal{I}$ into a rigorous *JSON schema*, defined formally as:

$$\mathcal{R} = \text{LLM}(\mathcal{I} \mid C_{schema}) \tag{1}$$

Table 1: The Schema of Issue Report Content Extraction

Property	Type	Definition
`title`	String	The title of the issue report.
`component`	String	The specific function under test.
`affected_versions`	List[String]	The specific version(s) reported as buggy.
`fixed_versions`	List[String]	The specific version(s) in which the issue is resolved.
`symptoms`	List[String]	A collection of observable behaviors.
`observed_cost`	List[Object]	Structured pairs of {metric, effect} describing the resource impact.
`description`	String	A normalized, technical restatement of the triggering conditions.

To ensure high-fidelity extraction, we employ a role-based prompt instructing the model to act as an expert performance engineer. This prompt explicitly directs the model to identify performance bottlenecks while strictly enforcing technical constraints. As detailed in Table 1, the schema C_{schema} mandates critical fields such as version numbers and observable symptoms, converting vague descriptions into actionable, structured specifications ($\mathcal{R}$).

Manual Analysis and Reproduction. While the specification $\mathcal{R}$ provides a logical description, a robust baseline requires empirical verification. We employ *Manual Analysis and Reproduction* to construct the *Baseline Seeds* ($\mathcal{S}$). Crucially, to strictly define a performance regression, $\mathcal{S}$ must contain at least two distinct artifacts:

(1) A *Triggering Seed* (s_{trigger}) that strictly matches the issue description and reproduces the degradation.
(2) A *Non-triggering Seed* (s_{non}) acting as a control. This artifact is structurally identical to s_{trigger} except for the specific regression-inducing factor.

Formally, this set $\mathcal{S}$ is considered valid only if the performance consumption metric $\mu(s, v)$ (e.g., execution time, peak memory footprint) satisfies the following cross-validation criteria:

- **C1 (Stability):** $\mu(s_{non}, V_{affected}) \approx \mu(s_{non}, V_{fixed})$. This rules out environmental performance fluctuations.

- **C2 (Regression):** $\mu(s_{trigger}, V_{affected}) \gg \mu(s_{trigger}, V_{fixed})$. This confirms the performance degradation is specific to the affected version.
- **C3 (Impact):** $\mu(s_{trigger}, V_{affected}) \gg \mu(s_{non}, V_{affected})$. This establishes that the degradation is induced by the specific triggering factor in $s_{trigger}$.
- **C4 (Fix Verification):** $\mu(s_{trigger}, V_{fixed}) \approx \mu(s_{non}, V_{fixed})$. This verifies that the fix successfully mitigates the structural vulnerability.

Only seed sets satisfying these four criteria are admitted into the subsequent phases.

Nature of the Baseline Seeds. The *Baseline Seeds* ($\mathcal{S}$) constitute a verified regression context rather than simple test inputs. Unlike functional testing, performance testing necessitates a comparative reference. Thus, $\mathcal{S}$ is defined by its dual composition: it encompasses both the problematic input (s_{trigger}) and the control input (s_{non}). This structure ensures that any reproduced issue is grounded in relative comparison, effectively filtering out environmental noise. In subsequent phases, $\mathcal{S}$ serves as the *Mutation Template* for exploring the bug's structural neighborhood.

2.2 Phase II: Mutation Strategy Generation

The objective of this phase is to synthesize executable mutation strategies by integrating the semantic specifications from the *Structured Issue Report* ($\mathcal{R}$) with the structural metadata of the *Baseline Seeds* ($\mathcal{S}$). Moving beyond random bit-flipping, this process utilizes these inputs to formulate precise, structure-aware directives that specifically target the reported performance regressions.

File Scanning. To acquire structural context, we employ a *File Scanner* that traverses the internal architecture of the Baseline Seeds ($\mathcal{S}$). Distinct from flat byte-stream processing, the scanner parses the file's inherent *object graph* to identify key elements such as object types, hierarchy depths, and cross-reference dependencies. We implement this using standard, format-specific libraries (e.g., *pikepdf* for PDFs, *python-docx* for documents) to ensure both parsing accuracy and framework extensibility. This design allows the framework to readily adapt to diverse input formats by leveraging the existing ecosystem.

Metadata Abstraction. The scanner's output is distilled into a lightweight textual representation, termed *Metadata*. This abstraction addresses two critical constraints: (1) raw binary artifacts often exceed LLM context windows, and (2) LLMs process semantic text more effectively than raw data streams. By generating a structural summary, we provide the subsequent synthesis phase with a concise yet comprehensive map of the seed's available mutation vectors.

LLM-Driven Strategy Synthesis. At the core of this phase is the *LLM-Driven Strategy Synthesizer*, which serves as the reasoning engine to formulate targeted mutation plans. This synthesis is driven by two primary data inputs:

- **Structured Issue Report ($\mathcal{R}$):** Derived from Phase I, this input provides the *performance regression specification*. It dictates what logic needs to be injected based on the historical issue analysis.
- **Metadata of Baseline Seeds:** This input provides the *structural context*. It dictates where the injection can occur, mapping the abstract performance regression logic to concrete objects and coordinates within the specific baseline seeds.

To ensure structural compliance and logical rigour, the generation process is governed by a *Role-Based Prompt* and a strict *JSON Schema*. The prompt conditions the model to act as a performance testing expert, enforcing a deductive workflow: (1) performing root cause analysis on the issue report; (2) selecting target seeds based on feature matching; and (3) synthesizing actionable mutation strategies defined by quantitative parameters. Complementing this, the JSON Schema (as defined in Table 2) mandates a rigorous output structure. By enforcing specific fields such as execution rationale and atomic modification sequences, the schema effectively eliminates natural language ambiguity (hallucinations) and ensures direct executability by the mutation engine.

The resulting output is a set of *Mutant Strategies*, which serve as formalized JSON specifications prescribing precise structural transformations. Each strategy explicitly maps a specific mutation operation (e.g., duplication, recursive injection, or stream expansion) to a target node within the seed's metadata, complete with parameters designed to explore latent performance degradation factors.

Table 2: Definition of the Mutation Strategy Schema

Property	Type	Definition
`priority`	Enum	The execution priority (high, medium, low) based on the likelihood of triggering new issue.
`short_description`	String	A concise summary of the intended structural change.
`rationale`	String	The logical derivation connecting the mutation strategy to specific symptoms in the issue report.
`applicability`	List[ID]	The specific subset of baseline seeds compatible with this mutation.
`modifications`	List[Obj]	A sequence of atomic actions, where each action defines: • **target**: The component of seed (e.g., *Page Tree*, *XRef Table*). • **operation**: The mutation primitive (e.g., *duplicate*, *bit_flip*). • **params**: Key-value arguments (e.g., {count : 50}).
`expected_effects`	List[Str]	Hypothesized system behaviors (e.g., "Stack Overflow", "Timeout").
`risk_notes`	List[Str]	Potential validity risks.

2.3 Phase III: Strategy Execution

Phase III materializes the theoretical mutation strategies generated in Phase II into concrete, testable artifacts. As illustrated in Figure 1, this phase functions as a code-synthesis and execution engine, transforming abstract strategy specifications into executable software mutants through an iterative, LLM-driven programming process.

Automated Code Synthesis. The process initiates by feeding the *Mutant Strategies* into LLM. Acting as a developer, the LLM translates the described mutation logic into specific *Executable Methods* implemented in Python. To facilitate this translation, we employ a prompt used for code generation that conditions LLM to function as a specialized automation engineer. The prompt enforces a strict implementation protocol: it requires the model to map the declarative JSON operations to specific API calls within the target manipulation library (e.g., `pikepdf` or `python-docx`), while explicitly mandating robustness constraints such as exception handling. This step effectively bridges the gap between the semantic intent of the strategy and the imperative syntax required to modify the binary or textual structure of the seeds.

Mutation Execution via Script Library. Initially, the LLM generates raw Python executable methods corresponding to each mutation strategy. To ensure reliability and maintainability, these raw snippets are not directly deployed but undergo a process of manual inspection and refinement. In this phase, we aggregate the generated codes to remove redundancies, merge overlapping logic into modular functions, and inject robust error handling mechanisms. The resulting consolidated codebase constitutes the *Execution Script Library*. This library functions as a specialized utility suite, providing standardized APIs to parse, modify, and reconstruct the target file format. By invoking these refined methods upon the baseline seeds, the system produces a set of high-quality mutants.

Crucially, to ensure rigorous traceability, the library is instrumented with a dedicated modification tracker. As the mutation logic executes, this tracker captures granular telemetry for every atomic operation, serializing it into a structured JSON log. Specifically, each entry records the *operation type*, the unique *target object ID* within the file structure, *modification details*, and a *quantification metric*. This logging mechanism serves as the ground truth to verify that the intended structural changes were successfully materialized in the binary artifact.

Validation and Iterative Refinement. To ensure testing viability, the *Validator* rigorously assesses generated mutants against two critical criteria. First, the *Syntactic Validity* mandates that the mutant conforms to the target software's basic format specifications, preventing immediate parsing rejection. Second, the *Strategy Compliance* verifies the internal structure by re-employing the *File Scanner* in Phase II. This step cross-references scanned metadata with execution logs to confirm that prescribed mutation operations were deterministically materialized at the intended target nodes.

Self-Correction Loop. If the validator detects a failure (e.g., a syntax error in the generated script or a corrupted file), the system triggers a feedback loop. The failure logs are fed back into the LLM, which then analyzes the error context and regenerates corrected executable methods. This iterative refinement continues until the method produces a valid mutant, which is then promoted to the final set of inputs for Phase IV.

2.4 Phase IV: Mutants Profiling and Analysis

The final phase of the framework focuses on the empirical validation and quantitative assessment of the generated mutants. The objective is to determine whether the validated mutants successfully induce new performance degradation. This is achieved through a rigorous execution process that evaluates the mutants against specific software version pairs $\mathcal{V} = \{V_{affected}, V_{fixed}\}$.

Metric Acquisition. To quantify the performance behavior, we execute both the original baseline seed (s) and the generated mutant (m) within an isolated and deterministic environment to minimize background noise. During execution, the profiling module captures a comprehensive vector of resource metrics. We define $\mu(x, v)$ as the measured value of a specific metric when input x is processed by version v. These metrics span multiple resource dimensions, including but not limited to wall time, CPU cycles and peak resident set size (RSS).

Quantitative Analysis. The analysis module interprets raw metrics through two complementary dimensions. This dual-perspective approach not only isolates the performance cost induced by the mutation but also scrutinizes the *scalability gradient* across versions. By synthesizing these dimensions, the system discriminates between absolute regressions, resilience regressions, and ineffective mutations.

Dimension 1: Mutation Impact Factor (ρ_{impact}). First, to measure the sensitivity of a specific version v to the structural modification, we calculate the mutation impact factor (ρ_{impact}). This metric quantifies the normalized resource consumption relative to the baseline seed within the same version environment:

$$\rho_{impact}(v) = \frac{\mu(m, v)}{\mu(s, v)} \tag{2}$$

A high $\rho_{impact}(v)$ indicates that version v exhibits significant structural sensitivity to the mutation m. We compute this factor independently for both the affected ($\rho_{impact}(V_{affected})$) and fixed ($\rho_{impact}(V_{fixed})$) versions.

Dimension 2: Cross-Version Disparity ($\rho_{disparity}$). Second, to evaluate the absolute performance differential, we calculate the ratio between the affected and fixed environments:

$$\rho_{disparity} = \frac{\mu(m, V_{affected})}{\mu(m, V_{fixed})} \tag{3}$$

While $\rho_{disparity} \gg 1$ typically signals a standard regression, a value approaching 1 does not necessarily imply stability, as it may mask non-linear degradation trends.

Joint Diagnostic. The relationship between the impact factors and the disparity allows us to pinpoint complex performance behaviors:

- **Absolute Regression:** The standard failure mode where the affected version consumes significantly more resources than the fixed version. It is crucial to distinguish regression from natural scaling: for mutants designed to amplify workload (e.g., via content duplication), an absolute increase in resource consumption is the expected baseline behavior for any version. Therefore, high raw consumption values alone do not indicate a defect.
- **Resilience Regression ($\rho_{impact}(V_{fixed}) > \rho_{impact}(V_{affected})$):** This captures the performance inversion. Even if the fixed version remains superior in absolute terms (low $\rho_{disparity}$), a higher impact factor indicates that it suffers from a steeper degradation gradient under stress. This signals introduced algorithmic fragility or poor scalability in the "optimized" release.

- **Ineffective Mutation ($\rho_{impact}(V_{affected}) \approx 1$):** If the mutation fails to induce a notable cost increase compared to the seed, it is considered ineffective.

3 Preliminary Case Study

To evaluate the effectiveness and generalizability of our framework, we conduct a case study on *Apache PDFBox*, a widely used open-source Java PDF library.

3.1 Reproduction of Issue PDFBOX-959

To establish a concrete ground truth for evaluation, we focus on PDFBOX-959 [12], a reported performance regression in which specific font encodings cause severe latency during text extraction.

Scenario Reconstruction. The structured issue report ($\mathcal{R}$) indicates that PDFs utilizing Type1C fonts trigger significant performance degradation. Based on this specification, we manually selected a baseline seed containing the corresponding font dictionaries and performed a differential evaluation between the affected version ($V_{affected}$) and the fixed version (V_{fixed}).

Controlled Isolation via Synthesis. To isolate the regression factor while eliminating confounding file attributes (e.g., stream length or image resolution), we developed a standalone synthesis script to construct minimal, controlled seeds. Starting from a common skeletal base, the script deterministically generated:

(1) **Seed A (s_{T1C}):** a PDF embedding *Type1C* font objects.
(2) **Seed B (s_{TT}):** a structurally identical PDF using standard *TrueType* font objects.

This construction follows the atomic modification principle, ensuring that all other object parameters remain constant.

Verification Results. As illustrated in Table 3, when processing s_{T1C} on $V_{affected}$, the text extraction time was significantly higher compared to V_{fixed}. In contrast, s_{TT} showed negligible performance disparity between the two versions.

Insight. This successful isolation confirms the validity of the structural isolation methodology. Manually verifying that the performance gap is driven solely by the specific font structure, we establish that the historical regression is indeed reproducible through structural synthesis. This serves as a critical validation step, confirming that the targeted regression logic is structurally determinable and providing a verified ground truth for the subsequent phases.

3.2 Mutation Strategy Generation & Execution

Starting from the structurally isolated baseline seeds (i.e., Seed A (s_{T1C}) targeting the *Type1C* regression and Seed B (s_{TT}) serving as the control), we proceed to the mutation phase to evaluate the framework's ability to diversify the test space.

Quantitative Overview of Strategies. Prompting the LLM to maximize the diversity of mutation operators yielded a total of 22 distinct mutation strategies. These strategies were automatically mapped to their compatible targets seeds based on the structural features of the seeds:

- Targeting s_{T1C} (17 Strategies): The majority of strategies focused specifically on exploiting the Type1C font structure, suggesting the LLM correctly identified the complexity of the regression target.
- Targeting s_{TT} (1 Strategy): A single strategy was generated exclusively for the TrueType control structure.
- Dual-Target (4 Strategies): Four generic strategies were deemed applicable to both font formats (e.g., page tree manipulation independent of font type).

Consequently, the execution of these strategies initially targeted the synthesis of 26 unique mutants (17 derived from s_{T1C}, 1 from s_{TT}, and 4×2 from the dual-target strategies). However, our framework's strict validation protocol in Phase III filtered out 2 invalid candidates that failed to materialize the intended object hierarchy. As a result, a final set of 24 verified mutants advanced to the profiling phase.

Qualitative Strategy Analysis. The generated strategies demonstrate semantic awareness of the PDF object hierarchy, categorizing broadly into two dimensions enabled by our schema design:

- Cardinality Amplification: Strategies that iteratively expand existing structures, such as "Duplicating the font dictionary 100 times" or "Injecting recursive references into the trailer array." These test the scalability of the parser.
- Object-Centric Manipulation: Strategies targeting specific PDF entities, such as "Modifying the font stream length" or "Alter the encoding array in the Type1C dictionary."

Consolidated Execution Library. A key finding in the execution phase was the high degree of functional convergence. Although 22 distinct strategies were specified, the *Execution Script Library* required only 9 core executable methods to materialize them. This reduction is attributed to the modular design of the synthesized Python code:

- **Parametric Polymorphism:** Multiple strategies often shared the same underlying operation (e.g., "Add 50 pages" vs. "Add 500 pages") but differed only in quantitative parameters. The library consolidated these into single, parameterized methods.
- **Object-Oriented Abstraction:** Operations targeting different but structurally similar objects (e.g., distinct dictionary types) were abstracted into generic dictionary manipulation methods.

The ratio of strategies to executable methods highlights the high reusability of the generated code. The library effectively functions as a domain-specific API for *PDFBox* mutation. This suggests that as the framework iterates through more issues, the marginal cost of generating new mutants will decrease, as the library evolves from a single-issue utility into a project-wide mutation infrastructure capable of supporting diverse regression testing scenarios beyond the scope of *PDFBOX-959*.

3.3 Profiling Results

Beyond verifying the specific regression reported in *PDFBOX-959*, we leveraged the framework to uncover latent performance characteristics and evolutionary trends. This section details two key findings: the identification of independent structural cost factors and the detection of resilience regressions in optimized versions.

3.3.1 Identification of Latent Performance Factors. During the exploratory mutation campaign, the analysis module flagged a recurring pattern: mutants containing specific Annotation objects consistently exhibited high mutation impact factors (ρ_{impact}), regardless of the textual content size.

Controlled Experiment Design. To verify whether specific Annotation subtypes constitute definitive independent variables for

Table 3: Comprehensive Performance Diagnosis of PDFBOX-959. Dimension 1 (Structural Impact): Shows the overhead of Type1C (s_{T1C}) relative to TrueType (s_{TT}) within each version. Note the significant gap in $V_{affected}$ (Left) vs. negligible gap in V_{fixed} (Middle). Dimension 2 (Regression): Compares the performance stability across versions. The control seed (s_{TT}) remains stable (low gap), while the target (s_{T1C}) exhibits significant regression.

Scale	Dimension 1: Mutation Impact (Pre-fix) *Environment: $V_{affected}$ (v1.4.0)*			Dimension 1: Mutation Impact (Post-fix) *Environment: V_{fixed} (v1.6.0)*			Dimension 2: Regression Magnitude *Cross-Version Gap (V_{aff} vs V_{fix})*	
	s_{TT} (Control)	s_{T1C} (Target)	**Impact Gap (%)**	s_{TT} (Control)	s_{T1C} (Target)	**Impact Gap (%)**	s_{TT} **(Control)**	s_{T1C} **(Target)**
1×	557.5 ms	700.6 ms	**+25.7%**	551.0 ms	595.1 ms	+8.0%	+1.2%	**+17.7%**
50×	3665.7 ms	3768.0 ms	+2.8%	3627.4 ms	3533.6 ms	*-2.6%*	+1.1%	**+6.6%**
100×	6443.5 ms	7132.7 ms	**+10.7%**	6479.8 ms	6358.5 ms	*-1.9%*	-0.6%	**+12.1%**

Table 4: Structured Issue Report ($\mathcal{R}$) for PDFBOX-959.

Field	Content
Title	Text extraction slow and `/tmp` fills up with AWT font files
Versions	**Affected:** 1.4.0 **Fixed:** 1.6.0
Component	Text extraction
Symptoms	• Slow text extraction performance • `/tmp` directory fills up with temporary AWT font files after many text extractions • Text extraction is effectively single-threaded due to synchronized AWT font creation • Unnecessary AWT font creation during text extraction when it is not needed
Observed Cost	• **CPU time**: Increased due to unnecessary creation of AWT fonts for Type1C fonts during text extraction initialization • **Disk space**: Temporary AWT font files accumulate and fill `/tmp` after a few thousand text extraction operations • **Throughput**: Reduced because AWT font creation occurs in a synchronized region, serializing text extraction across threads
Description	In PDFBox 1.4.0, when performing text extraction on PDFs that use Type1C fonts, the Type1C font implementation eagerly creates a corresponding AWT font object during font initialization. This AWT font creation is done unconditionally, even though text extraction itself does not require AWT fonts. The AWT font creation occurs inside a synchronized block, which serializes access and forces text extraction operations that touch Type1C fonts to run effectively single-threaded across threads. Additionally, creating the AWT font causes the underlying font data to be copied into temporary files under `/tmp`.

performance degradation, we designed a controlled isolation experiment. We selected a baseline seed (s_{clean}) containing zero annotations and synthesized two targeted variants designed to mark text spans:

(1) **Variant A ($m_{Straight}$):** s_{clean} injected with straight line annotations (specifically underline style) targeting specific text segments.
(2) **Variant B (m_{Wavy}):** s_{clean} injected with wavy line annotations targeting the exact same text segments.

The synthesis process adhered to the atomic modification principle. We rigorously controlled the spatial and quantitative parameters, ensuring that the quantity of marked characters and their geometric coordinates remained mathematically identical across both variants. This isolation ensures that any observed performance divergence is attributable solely to the rendering logic of the specific annotation style (straight vs. wavy).

Findings. We profiled the execution time and memory consumption for these samples (see Figure 2). The results demonstrate a strictly structural overhead: both $m_{Straight}$ and m_{Wavy} triggered a substantial increase in wall time compared to the baseline (s_{clean}), resulting in $\rho_{impact} \gg 1$. This confirms that the text extraction logic in PDFBox incurs additional cost when traversing annotation dictionaries, even if those objects do not contribute to the extracted text. This finding validates the framework's capability to proactively identify latent structural factors that correlate with performance degradation.

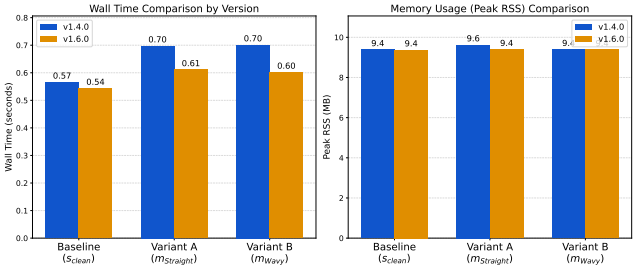


Figure 2: Performance impact of injecting different Annotation types

3.3.2 Cross-Version Stability Analysis. Finally, we analyzed the evolutionary characteristics of performance stability between the affected version ($V_{affected}$, v1.4.0) and the fixed version (V_{fixed}, v1.6.0). Our analysis reveals a complex, non-linear performance relationship.

The Optimization-Resilience Paradox. In nominal scenarios, V_{fixed} consistently outperforms $V_{affected}$ ($\rho_{disparity} < 1$), confirming valid general optimizations. However, high-load text mutants reveal a conflicting pattern: while absolutely faster, V_{fixed} suffers from a steeper degradation gradient. Our empirical data indicates:

$$\rho_{impact}(V_{fixed}) > \rho_{impact}(V_{affected}) \tag{4}$$

This inequality presents a paradox: the newer version, despite its absolute speed advantage, is relatively more sensitive to workload scaling.

Implications: Resilience Regression. This phenomenon indicates an algorithmic fragility. The optimizations in version 1.6.0, while beneficial for average cases, appear to scale more poorly than the predecessor when extracting massive text payloads. This finding highlights the framework's unique capability to detect resilience

regressions, which manifest as inversions in performance characteristics where a nominally optimized version exhibits severe degradation gradients under specific structural pressures—a critical insight frequently overlooked by standard benchmarks focused solely on average-case speed.

4 Future Work

While the current framework demonstrates the efficacy of LLM-driven structural mutation in finding new performance regressions, our ongoing research aims to further enhance the autonomy of the system.

Adaptive Prompt and Schema Engineering. To eliminate reliance on static, manually aligned templates, we plan to implement a context-aware prompt and schema generation module. This module will extract semantic context from issue reports, dynamically synthesizing optimal prompts and tailored JSON schemas. This automation will significantly enhances the framework's generalizability across diverse defect categories, obviating the need for manual template engineering.

Automated Script Synthesis and Fidelity Enhancement. To refine the execution phase, we plan to implement an automated script synthesis pipeline that autonomously constructs reusable libraries while maximizing code fidelity. Currently, library construction relies on manual aggregation, future iterations will leverage LLMs to classify and abstract atomic operations into object-oriented modules automatically. Simultaneously, to mitigate the computational latency of the current self-correction loop, we aim to integrate lightweight static code analysis and fine-tune models on domain-specific corpora. This unified approach optimizes the generation of syntactically sound scripts, significantly reducing both the setup cost for new targets and the overall time to test.

Generalization to Diverse Software Domains. We will validate the framework's universality by expanding the experimental scope beyond PDF processing. Future work will apply this workflow to other software categories, such as image processing engines and multimedia decoders. Evaluating heterogeneous target software will allow us to assess whether structural synthesis can consistently identify performance regressions across diverse file formats and architectural patterns, establishing the approach as a general solution for performance regression testing.

5 Conclusion

We proposed *Issue-Driven Performance Fuzzing*, a framework that transforms historical issue reports into proactive performance tests. Guided by the defect clustering principle, which posits that visible bugs are symptoms of broader structural weaknesses in their immediate vicinity, we utilized LLMs to extract failure patterns and synthesize targeted mutants. Our evaluation on *Apache PDFBox* confirmed that this approach not only reproduces known regressions but also exposes hidden vulnerabilities, including "Resilience Regressions" where optimized versions degrade faster under stress. These findings highlight the value of leveraging historical semantic data for automated performance testing.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation (NSF) under grant numbers CCF-2044888 and 2451553.

References

[1] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 199–208. IEEE, 2012.

[2] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[3] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 254–265, 2018.

[4] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2155–2168, 2017.

[5] Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[6] Jun Yang, Cheng-Chi Wang, Bogdan Alexandru Stoica, and Kexin Pei. Wedge: Synthesizing performance constraints for evaluating and improving code efficiency. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[7] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[8] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings]*, pages 1–14. Internet Society, 2017.

[9] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[10] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices*, 49(10):561–578, 2014.

[11] Yutong Zhao, Lu Xiao, Andre B Bondi, Bihuan Chen, and Yang Liu. A large-scale empirical study of real-life performance issues in open source projects. *IEEE transactions on software engineering*, 49(2):924–946, 2022.

[12] Apache PDFBox JIRA. [PDFBOX-959] Text extraction slow and /tmp fills up with AWT font files. https://issues.apache.org/jira/browse/PDFBOX-959, 2011. Accessed: 2026-01-26.