

Characterizing Multi-Hunk Patches: Divergence, Proximity, and LLM Repair Challenges

Noor Nashid

University of British Columbia
Vancouver, Canada
nashid@ece.ubc.ca

Ahmed E. Hassan

Queen's University
Kingston, Canada
ahmed@cs.queensu.ca

Daniel Ding

University of British Columbia
Vancouver, Canada
dyxd2003@ece.ubc.ca

Keheliya Gallaba

Queen's University
Kingston, Canada
gallabak@sigsoft.org

Ali Mesbah

University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

Abstract—Multi-hunk bugs, where fixes span disjoint regions of code, are common in practice, yet remain underrepresented in automated repair. Existing techniques and benchmarks predominantly target single-hunk scenarios, overlooking the added complexity of coordinating semantically related changes across the codebase. In this work, we characterize HUNK4J, a dataset of multi-hunk patches derived from 372 real-world defects. We propose *hunk divergence*, a metric that quantifies the variation among edits in a patch by capturing lexical, structural, and file-level differences, while incorporating the number of hunks involved. We further define *spatial proximity*, a classification that models how hunks are spatially distributed across the program hierarchy. Our empirical study spanning six LLMs reveals that model success rates decline with increased divergence and spatial dispersion. Notably, when using the LLM alone, no model succeeds in the most dispersed *Fragment* class. These findings highlight a critical gap in LLM capabilities and motivate divergence-aware repair strategies.

Index Terms—Multi-hunk, Benchmark, Program repair, Hunk Divergence, Spatial Proximity, Large Language Model

I. INTRODUCTION

Automated Program Repair (APR) has received significant attention from the research community [1]–[18]. APR techniques have progressed from heuristic-based methods [19]–[22] to deep learning approaches that learn repair patterns from large codebases [10], [15]–[18], achieving promising results on single-hunk bugs [4], [5], [10], [16]–[18], [23]–[27]. Despite recent progress, most APR research remains focused on single-hunk scenarios [10], [16]–[18], [24], overlooking *multi-hunk bugs*, defects that require edits across disjoint code regions. These are both common in practice [28] and harder to fix due to their semantic interdependencies. While the problem has been acknowledged [28], only a few studies have directly addressed it [28]–[33], and they have demonstrated limited effectiveness.

Recent advancements in large language models (LLMs) have shown promise in transforming several software engineering tasks, including program repair [34]. Trained on massive datasets comprising code and natural language, LLMs such as GPT-4 and LLaMA have demonstrated the ability to

understand and generate code [34]–[37]. While these models have already been employed to address single-hunk bugs with notable success, their potential to handle multi-hunk code repair remains underexplored.

Early studies on LLM-based multi-hunk repair include tools such as AlphaRepair [35], RepairAgent [38], and fine-tuning strategies [39], [40]. Although these approaches demonstrate that LLMs can extend beyond single-hunk scenarios, the challenges of reasoning about semantically interdependent edits distributed across the codebase remain unaddressed.

We identify four challenges limiting progress. First, *dataset variation* impedes consistency: prior studies evaluate on synthetic benchmarks such as LMDefects [41], algorithmically curated tasks (EvalRepair-C++/Java) [40], and real-world datasets such as Defects4J [42], often under varying assumptions, input formats, and test harnesses. Second, *bug-type ambiguity* obscures evaluation: multi-hunk bugs span a wide spectrum, from edits in non-contiguous regions within a single function to interdependent changes across multiple files. Yet prior work does not systematically characterize this *heterogeneity*, often conflating distinct forms of multi-location repair such as multi-line, multi-function, and multi-file edits. Third, *reporting inconsistencies* hinder meaningful attribution: existing techniques often report aggregate metrics that combine single- and multi-hunk results [38], [40], making it difficult to isolate and evaluate their effectiveness specifically on multi-hunk repairs. Fourth, existing evaluations lack *patch complexity-aware metrics* that quantify differences in hunks within a patch.

To address these challenges, we (1) propose *hunk divergence*, a metric that quantifies the internal variation within a multi-hunk patch by measuring lexical, structural, and file-level distance among hunks, while accounting for the number of hunks and the dispersion across files; (2) define *spatial proximity* as a classification scheme that assigns each patch to one of five categories—Nucleus, Cluster, Orbit, Sprawl, or Fragment—reflecting increasing levels of dispersion across the program hierarchy, including methods, files, and packages;

(3) characterize multi-hunk patches using these abstractions to analyze hunk divergence and inter-hunk relationships; (4) develop BIRCH, a platform for standardized evaluation of LLM-based multi-hunk repair that supports reproducible comparisons across models and configurations; and (5) conduct a large-scale empirical study, examining the impact of retrieval strategies, context granularity, and feedback on repair success and cost under varying degrees of divergence and dispersion.

In this work, we make the following contributions:

- We propose *hunk divergence*, a metric that quantifies intra-patch variation across lexical, structural, and file-level distance, and *spatial proximity*, a classification of multi-hunk patches based on their spatial layout in the codebase.
- We introduce HUNK4J, a benchmark of 372 real-world multi-hunk bugs. We also present the first systematic characterization of multi-hunk bugs, focusing on their hunk diversity and proximity.
- We present BIRCH, a platform for evaluating the capabilities of LLMs on multi-hunk bugs. Our empirical study spans 6 LLMs, various modes including 2 code representations, and 3 retrieval strategies, while also examining the role of contextual scope and incorporating feedback from compilation errors and test failures.
- Our empirical findings show that model accuracy declines significantly as *hunk divergence* increases and spatial proximity decreases. All evaluated models without augmentation failed to repair any bugs in the most dispersed *Fragment* class, underscoring the challenges posed by highly scattered hunks.

II. CHARACTERIZATION OF MULTI-HUNK PATCHES

Although multi-hunk patches have been studied [28], [33], [38], their structural and spatial properties and how they affect automated repair remain poorly understood.

Definition 1 (Hunk). A *hunk* is a block of consecutive edits applied to a specific region in the source code. We define each hunk h_i as a tuple:

$$h_i = (\text{loc}_i, \text{content}_i, \text{file}_i, \text{method}_i, \text{pkg}_i)$$

where:

- loc_i : line range of the edit,
- content_i : token sequence derived from the raw diff (concatenated deletions and additions),
- file_i : file path of the edit,
- method_i : enclosing method identifier, or None if outside any method.
- pkg_i : package path as a sequence of directory segments.

Definition 2 (Multi-Hunk Patch). A *multi-hunk patch* P is a set of $n \geq 2$ distinct hunks, denoted as $P = \{h_1, h_2, \dots, h_n\}$, where the hunks modify non-contiguous regions of source code. P is an atomic change intended to address a single bug.

Our insight is that multi-hunk patches can vary significantly along several dimensions. In some cases, the hunks are nearly

identical, such as repeated renamings or boilerplate edits, while in others, they involve semantically distinct changes that differ both lexically and structurally. These edits can also vary in proximity, from hunks located within the same method to those scattered across multiple files. This observation led us to ask: to what extent do hunks within a patch differ, and can this variation be meaningfully quantified?

We present the first in-depth characterization of multi-hunk patches by introducing two new metrics: (a) *hunk divergence*, which quantifies lexical, structural, and file-level dissimilarity among hunks, and (b) *spatial proximity*, which classifies the spatial layout of hunks across the code hierarchy.

A. Hunk Divergence

We propose *hunk divergence*, a new metric that captures lexical, structural, and file-level differences among hunks in a patch.

Definition 3 (Pairwise Hunk Divergence). We define the *pairwise hunk divergence* between two hunks h_i and h_j of P through a combination of lexical, structural, and file-level separation distances:

$$\text{Div}(h_i, h_j) = \frac{D_{\text{lex}} \cdot (D_{\text{ast}} + \gamma \cdot D_{\text{file}})}{1 + \gamma}$$

where:

- $D_{\text{lex}}(h_i, h_j)$ quantifies the lexical distance between the code token sequences content_i and content_j , where each token sequence is derived directly from the raw textual diff of the corresponding hunk.
- $D_{\text{ast}}(h_i, h_j)$ measures the structural distance between the abstract syntax tree (AST) nodes of the two hunks.
- $D_{\text{file}}(h_i, h_j)$ quantifies the file-level separation between the files file_i and file_j that contain the hunks h_i and h_j , respectively.
- γ is a context-sensitive weighting factor that amplifies the contribution of inter-file separation to the overall divergence score, increasing the impact of file-based fragmentation when edits span distinct files in the repository.

By construction, if D_{lex} , D_{ast} , and D_{file} are each normalized to the interval $[0, 1]$, then the resulting pairwise divergence score satisfies $\text{Div}(h_i, h_j) \in [0, 1]$.

To compute *lexical distance* (D_{lex}), we tokenize the diff exactly as it appears in the patch, preserving the ordering of added and deleted lines. For hunks involving both additions and deletions, we concatenate the deleted lines (before the change) and the added lines (after the change) prior to tokenization. In cases where a hunk contains only additions or only deletions, the token sequence consists solely of the respective lines as shown in the diff. We use BLEU [43] to quantify lexical similarity between token sequences of code fragments, following prior work [10], [17], [44]–[50].

$$D_{\text{lex}} = 1 - \text{BLEU}(T_1, T_2)$$

where T_1 and T_2 are the token sequences of hunks h_1 and h_2 , respectively.

The goal of *structural distance* (D_{ast}) is to quantify how far apart two buggy hunks in the AST are when they appear in the same file. For this, we compute the shortest path between their corresponding nodes in the AST of the *buggy* (pre-patch) version of the file. Each hunk is associated with the AST node corresponding to the first statement that begins within its line range. For deletions, we extract this node from the pre-patch AST using the first deleted line in the hunk. Let $f = \text{file}(h_1) = \text{file}(h_2)$ be the shared file, and let T_f denote the buggy file's AST. Let $n_1, n_2 \in T_f$ be the AST nodes associated with h_1 and h_2 , respectively. The structural distance is then defined as the normalized distance between these nodes, using the diameter of T_f as a scaling factor.

To capture local structural variation between hunks, we normalize their AST distance using a natural logarithm-scaled formulation.

$$D_{\text{ast}} = \begin{cases} \frac{\ln(1 + \text{ASTDIST}(n_1, n_2))}{\ln(1 + \text{TREEDIAMETER}(T_f))} & \text{if } \text{TREEDIAMETER}(T_f) > 0 \\ 0 & \text{otherwise} \end{cases}$$

This formulation ensures numerical stability by avoiding division by zero when the AST is malformed or corresponds to an empty file. We apply log normalization to reduce the impact of large tree diameters, which can otherwise suppress meaningful structural differences between closely located nodes. If the buggy hunks occur in different files, we set $D_{\text{ast}} = 1$. This reflects that structural comparison is undefined across separate ASTs, and maximum structural divergence is used in this case.

The goal of *file divergence* (D_{file}) is to quantify the degree of file-level separation between two hunks based on the layout of their enclosing file paths within the repository hierarchy. Let $\text{file}(h_i)$ denote the fully qualified file path associated with hunk h_i , represented as an ordered sequence of directory segments. For brevity, we write $f_i = \text{file}(h_i)$ and define $D_{\text{file}} : \mathcal{H} \times \mathcal{H} \rightarrow [0, 1]$ as:

$$D_{\text{file}}(h_i, h_j) = \begin{cases} 0 & \text{if } f_i = f_j \\ 1 - \frac{|\text{LCP}(f_i, f_j)|}{\max(|f_i|, |f_j|)} & \text{otherwise} \end{cases}$$

Here, $\text{LCP}(f_i, f_j)$ denotes the length of the longest common prefix between the directory sequences f_i and f_j , the two distinct buggy files, and $|\cdot|$ denotes the number of directory components in a given path. This formulation produces a normalized estimate of file-level distance: file paths that share a deeper hierarchical structure yield lower divergence values. This provides a continuous, repository hierarchy-aware quantification of inter-file separation that is both bounded and independent of path depth. It captures whether a pair of hunks is confined to the same file or spans across distinct files, serving as a proxy for the inter-file coordination effort required during multi-hunk repair.

Overall, our formulation of pairwise divergence $\text{Div}(h_1, h_2)$ ensures that the score remains bounded in $[0, 1]$, while amplifying file-level separation when edits span

Algorithm 1: PAIRWISE DIVERGENCE SCORE

```

Input: Hunks  $h_1, h_2$  with attributes: file, ast, code;
Output: Normalized divergence score  $D(h_1, h_2) \in [0, 1]$ 
1  $\gamma \leftarrow 1$  // Scaling factor for file-level divergence
2  $D_{\text{lex}} \leftarrow 1 - \text{BLEU}(h_1.\text{code}, h_2.\text{code})$ 
3 if  $h_1.\text{file} = h_2.\text{file}$  then
4    $f \leftarrow h_1.\text{file}$ 
5    $T_f \leftarrow \text{PARSEAST}(f)$ 
6    $\text{treeDiameter} \leftarrow \text{COMPUTEDIAMETER}(T_f)$ 
7   if  $\text{treeDiameter} > 0$  then
8      $n_1 \leftarrow \text{LOCATENODE}(T_f, h_1.\text{loc})$ 
9      $n_2 \leftarrow \text{LOCATENODE}(T_f, h_2.\text{loc})$ 
10     $\text{astDist} \leftarrow \text{ASTDISTANCE}(n_1, n_2)$ 
11     $D_{\text{ast}} \leftarrow \ln(1 + \text{astDist}) / \ln(1 + \text{treeDiameter})$ 
12  else
13     $D_{\text{ast}} \leftarrow 0$ 
14  end
15   $D_{\text{file}} \leftarrow 0$  // No file-level separation: edits are in
   // the same file
16 else
17   $D_{\text{ast}} \leftarrow 1$  // Maximum divergence across separate ASTs
18   $f_1 \leftarrow h_1.\text{file}, f_2 \leftarrow h_2.\text{file}$ 
19   $\text{lcpLen} \leftarrow \text{LCP}(f_1, f_2)$ 
20   $D_{\text{file}} \leftarrow 1 - \text{lcpLen} / \max(|f_1|, |f_2|)$ 
21   $\gamma \leftarrow 2$  // Amplify inter-file separation weight
22 end
23  $\text{pairDiv} \leftarrow D_{\text{lex}} \cdot (D_{\text{ast}} + \gamma \cdot D_{\text{file}})$ 
24  $\text{maxDiv} \leftarrow 1 + \gamma$ 
25 return  $D(h_1, h_2) \leftarrow \text{pairDiv} / \text{maxDiv}$ 

```

multiple files. For same-file edits ($D_{\text{file}} = 0$), the score is proportionally downweighted by $1 + \gamma$, reflecting the reduced coordination effort relative to inter-file changes.

In Algorithm 1, we show the steps for computing the *pairwise divergence* $D(h_1, h_2)$ between two hunks. Line 2 computes lexical divergence using BLEU score. Lines 3–15 implement the conditional logic for determining whether the hunks are co-located in the same file. Line 11 handles the intra-file case: the structural divergence is computed as the normalized distance between AST nodes within the same file, using the AST's diameter for normalization. File-level divergence is omitted by setting $D_{\text{file}} = 0$. In contrast, the cross-file case (beginning at Line 17) assigns $D_{\text{ast}} = 1.0$ to denote maximal syntactic divergence across disjoint ASTs. File-level divergence is then computed using the normalized longest common prefix (LCP) between the two file paths. The scaling factor γ is set to 2.0 to amplify the influence of file-level separation. This dynamic adjustment of γ reflects the increased effort required when developers must coordinate edits across multiple files, ensuring that file-level fragmentation contributes more significantly to the overall divergence score in multi-hunk patches.

Definition 4 (Hunk Divergence). The *hunk divergence* of a multi-hunk patch P quantifies the degree of internal variation among its constituent hunks. Let P consist of n hunks $\{h_1, h_2, \dots, h_n\}$, we define the hunk divergence of P as:

$$\text{Div}(P) = \ln(n) \cdot \left(\frac{2}{n(n-1)} \sum_{1 \leq i < j \leq n} \text{Div}(h_i, h_j) \right)$$

$\text{Div}(P)$ captures how lexically, structurally, and spatially dissimilar the hunks are pairwise and the coordination complexity

introduced by the number of hunks in P .

We apply a natural logarithmic scaling factor $\ln(n)$ to reflect coordination complexity, increasing divergence for more fragmented patches while avoiding over-penalization when hunks are highly similar. As a result, the overall hunk divergence score satisfies $\text{Div}(P) \in [0, \ln(n)]$.

Consider the following examples: for a patch with $n = 5$ hunks that are all *identical*, all pairwise distances are zero, yielding a hunk divergence score of $\text{Div}(P) = 0$. At the other extreme, if all five hunks are *maximally different* (i.e., each pair has a divergence score of 1), then the average pairwise divergence is 1 and the resulting hunk divergence is $\text{Div}(P) = \ln(5) \approx 1.61$. In a more typical case, if the average pairwise divergence among the hunks is moderate (e.g., 0.5), the divergence score becomes $\text{Div}(P) = \ln(5) \cdot 0.5 \approx 0.80$.

B. Spatial Proximity

Spatial proximity classifies each multi-hunk patch into one of five categories—*Nucleus*, *Cluster*, *Orbit*, *Sprawl*, or *Fragment*—representing increasing levels of proximity dispersion across methods, files, and packages. Unlike hunk divergence, which assigns continuous quantitative scores based on hunk dissimilarity, this classification provides rule-based insights into the spatial distribution of hunks.

Definition 5 (Spatial Proximity Class). Let $P = \{h_1, \dots, h_n\}$ be a multi-hunk patch.

$$\begin{aligned} \text{SM}(P) &= \exists m \text{ such that } \forall h_i \in P, m_i = m \\ \text{SF}(P) &= \exists f \text{ such that } \forall h_i \in P, f_i = f \\ \text{SP}(P) &= \exists p \text{ such that } \forall h_i \in P, \text{pkg}_i = p \\ \text{LCP}(P) &= \min_{i \neq j} |\text{LCP}(\text{pkg}_i, \text{pkg}_j)| \end{aligned}$$

These predicates assess different aspects of spatial proximity among the edits in a patch. Specifically:

- $\text{SM}(P)$: all hunks are in the same method.
- $\text{SF}(P)$: all hunks are in the same file.
- $\text{SP}(P)$: all hunks share the same package.
- $\text{LCP}(P)$: minimum shared directory depth across hunk pairs.

Then, the spatial proximity class $C(P)$ is defined as:

$$C(P) = \begin{cases} \text{Nucleus} & \text{if } \text{SF}(P) \wedge \text{SM}(P) \\ \text{Cluster} & \text{if } \text{SF}(P) \wedge \neg \text{SM}(P) \\ \text{Orbit} & \text{if } \neg \text{SF}(P) \wedge \text{SP}(P) \\ \text{Sprawl} & \text{if } \neg \text{SF}(P) \wedge \neg \text{SP}(P) \wedge \text{LCP}(P) > \lambda \\ \text{Fragment} & \text{if } \neg \text{SF}(P) \wedge \neg \text{SP}(P) \wedge \text{LCP}(P) \leq \lambda \end{cases}$$

Here, λ is a threshold on the directory depth used to distinguish structurally dispersed patches (*SPRAWL*) from widely scattered ones (*FRAGMENT*). This distinction is motivated by the intuition that when edits occur in directory paths with minimal shared hierarchy, the coordination burden for developers increases substantially.

TABLE I: Multi-hunk patch file scope and hunk count.

File Scope	Hunk Count Category	Number of Bugs
Single-file	Two hunks	140
	Three hunks	55
	Four or more hunks	49
Multi-file	Two hunks	37
	Three hunks	23
	Four or more hunks	68
Total		372

C. HUNK4J: A Dataset of Multi-Hunk Bugs

To enable principled evaluations, we introduce HUNK4J, a curated dataset of multi-hunk bugs derived from DEFECTS4J [51]. DEFECTS4J includes reproducible, test-suite backed bugs from open-source Java projects across diverse domains. We analyze all the 835 bugs in DEFECTS4J¹ and identify those whose developer-written patches contain multi-hunk fixes (Definition 2). We found that 372 out of 835 bugs (44.6%) are multi-hunk. These 372 multi-hunk bugs form the basis of our HUNK4J for further analysis. Beyond extracting patch-level characteristics, we augment HUNK4J with natural language context for each bug. Specifically, we collect bug reports, titles, and summaries from official project repositories and issue trackers, including Apache JIRA, GitHub Issues, and SourceForge. This enriched metadata facilitates prompt construction and retrieval-augmented strategies for LLM-based program repair.

By combining patches validated by test suites with detailed hunk information (Definitions 1, 2), HUNK4J enables analysis of heterogeneity across lexical, structural, file-level divergence, and spatial distribution.

D. Hunk Characterization

To analyze the complexity of multi-hunk bugs, we examine each patch in HUNK4J. Each patch is systematically analyzed based on the developer-written ground truth patch. We first identify the set of modified files by locating lines that begin with `diff --git`, which mark the inclusion of a file in the patch. Then, we extract distinct hunks within each file by detecting lines starting with `@@`, which indicate hunk boundaries. For single-hunk files, we further quantify the number of modified lines by counting those prefixed with either `-` (deletions) or `+` (additions). If a deletion is directly followed by an addition, we treat it as a single-line substitution rather than two independent changes. These steps facilitate a precise classification of hunks.

We categorize multi-hunk bugs along two axes: the number of change regions (*hunks*) and the number of files touched in Table I. Single-file multi-hunk bugs dominate, with 244 cases where changes remain confined within a single file—this includes 140 two-hunks, 55 with three hunks, and 49 with four or more. These reflect intra-file repair scenarios, requiring local reasoning. In contrast, 128 bugs span multiple files, representing more fragmented, distributed repairs. Notably, 68 of these involve four or more hunks, indicating considerable

¹We use DEFECTS4J version 2.0.1 for our analysis.

TABLE II: HUNK4J dataset: hunk and file statistics

Project	Bugs	Hunks per Bug			Files per Bug			
		Min	Median	Mean	Max	Min	Median	Mean
Chart	10	2	2.50	3.00	6	1	1.00	1.20
Cli	17	2	2.00	3.47	10	1	1.00	1.71
Closure	82	2	3.00	4.06	21	1	1.00	1.63
Codec	7	2	3.00	3.57	7	1	2.00	2.43
Collections	1	3	3.00	3.00	3	1	1.00	1.00
Compress	18	2	3.00	3.89	14	1	1.00	1.61
Csv	4	2	2.00	2.50	4	1	1.00	1.25
Gson	7	2	2.00	4.00	12	1	1.00	1.43
JacksonCore	12	2	3.50	4.33	8	1	2.00	1.75
JacksonDatabind	55	2	3.00	4.58	26	1	1.00	1.87
JacksonXml	3	2	2.00	2.67	4	1	1.00	1.00
Jsoup	37	2	3.00	4.73	47	1	1.00	1.89
JxPath	14	2	4.00	4.50	11	1	2.00	1.93
Lang	25	2	2.00	2.72	7	1	1.00	1.00
Math	50	2	2.00	2.80	12	1	1.00	1.26
Mockito	20	2	3.00	4.00	20	1	1.00	1.40
Time	10	2	3.00	4.00	10	1	1.00	1.50
Total	372	2	3.00	3.86	47	1	1.00	1.59
								16

dispersion across the codebase. For instance, *multi-hunk patch P* for JSOUP_56 [52] touches 10 hunks across 5 files.

To complement our earlier categorization, Table II presents project-specific statistics capturing how multi-hunk bugs manifest across different codebases. In 15 out of 17 projects, the median number of hunks per bug is at most 3, suggesting that the majority of bugs involve relatively localized changes. However, outliers exist, Jsoup_87 [53] has 47 hunks. In contrast, Chart_4 [54] involves a multi-hunk fix with only two hunks confined to a single method, which is simpler to reason about. Across all 17 projects, the median number of files modified per bug does not exceed 2; nonetheless, notable exceptions are present. For instance, JacksonDatabind_103 [55] spans 26 hunks across 16 files, modifying exception handling across the codebase. Conversely, certain projects such as Lang, Collections, and JacksonXml contain no multi-file bugs.

Figure 1 presents the distribution of divergence for HUNK4J. Lexical distance (D_{lex}) ($Md = 0.94$, Mean = 0.82) is sharply skewed toward the upper bound, indicating that most hunk pairs are lexically dissimilar. Structural distance (D_{ast}) ($Md = 0.6$, Mean = 0.62) shows broader dispersion, reflecting moderate variation in the AST of edits. File-level distance for multi-file patches (D_{file}) exhibits a moderate central tendency ($Md = 0.250$, Mean = 0.251), indicating that while many multi-hunk patches involve hunks within closely related files, a non-trivial portion span more distinct locations in the file hierarchy.

Overall hunk divergence in HUNK4J ranges from a minimum of 0.00 to a maximum of 1.60, with a median of 0.42, a mean of 0.47. Notably, 25% of multi-hunk patches exhibit divergence values greater than 0.624. The distribution reflects heterogeneity across lexical content, code structure, and file-level separation. Importantly, the divergence landscape is far from trivial, refuting any assumption of homogeneity.

We also characterize the multi-hunk patches according to our spatial proximity classification (Definition 5). We instantiate the proximity threshold λ based on the empirical distribution of package path depths in HUNK4J. The median package depth across all buggy files is 6; we set $\lambda = 3$, corresponding to half this value, to conservatively separate loosely grouped edits from those that are structurally dispersed across the codebase.

Table III categorizes multi-hunk patches based on their spa-

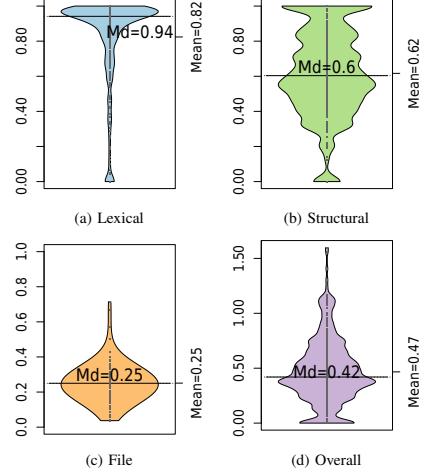


Fig. 1: Distribution of different divergence components and the overall hunk divergence.

TABLE III: Spatial proximity class and hunk divergence.

Proximity Class	Number of Bugs	Mean Hunk Divergence
Nucleus	59	0.2548
Cluster	185	0.4280
Orbit	67	0.5628
Sprawl	50	0.6718
Fragment	11	0.7372

tial proximity, revealing a distinct skew toward more localized fixes. The *Cluster* class, comprising 185 bugs, is the most prevalent and includes patches where multiple hunks occur within the same file but in different methods. This is followed by the *Orbit* class (67 bugs), which captures cases where hunks span multiple files within the same package, and the *Nucleus* class (59 bugs), where all hunks are confined to a single method. Mockito_6 [56] from the *Cluster* class contains 20 hunks, all confined to a single file yet distributed across multiple methods, demonstrating file-level cohesion despite method-level dispersion. In contrast, the *Sprawl* (50 bugs) and *Fragment* (11 bugs) classes represent increasingly dispersed patches. The *Fragment* category consists of patches where hunks are scattered across unrelated packages. An example from *Fragment* class is Closure_144 [57], in which the hunks are scattered across unrelated packages, highlighting the low spatial locality. The average hunk divergence increases monotonically with spatial dispersion. *FRAGMENT* patches, though rare, exhibit the highest divergence (0.7372), underscoring their complexity.

III. EMPIRICAL EVALUATION

Our study addresses the following research questions:

- **RQ1:** How well do LLMs perform on multi-hunk repair tasks?
- **RQ2:** To what extent do augmented prompting techniques (RAG, context, feedback) improve repair accuracy?

- **RQ3:** How do hunk characteristics such as divergence and spatial proximity influence the repair success of LLM-based techniques?
- **RQ4:** Are certain types of divergence and spatial proximity more predictive of challenges in LLM repair?

A. Benchmarking Framework: BIRCH

To enable reproducible evaluations across different LLMs, we present BIRCH, a benchmarking framework for multi-hunk program repair. It offers an end-to-end pipeline, from input preparation to patch validation. The process begins by checking out the buggy version of the program from HUNK4J and setting up a workspace. For each bug, BIRCH extracts the buggy code, relevant test cases, and associated metadata. It then constructs an input prompt that includes the buggy code, the precise bug location, a natural language description of the issue, and the failing test cases with their corresponding error messages. Once the prompt is prepared, it is submitted to a selected LLM. The patch is parsed and applied to the buggy code. The code is then compiled and tested using the test suite. Compilation errors and failing test cases are logged.

Modes. BIRCH supports assessing vanilla LLMs but also provides various optional modes for enhancing LLMs. The *retrieval-based example selection* mode supports both sparse and dense retrieval techniques. The *enclosing context selection* mode extracts context at varying granularities, including the method, class, and file levels. The *feedback loop* mode facilitates iterative patch refinement: if a generated patch fails to compile, the corresponding compilation error message is appended to the subsequent prompt. If the patch compiles but fails to pass the test suite, detailed test failure information is incorporated to guide further refinement attempts. If the code compiles and passes all tests, the patch is marked as successful. All evaluation artifacts, including prompts, generated code, diffs, and test logs, are persisted for inspection and analysis.

Implementation. BIRCH is implemented in Python and supports both proprietary models and open-source models via Ollama [58]. We use `lite11m` [59] to interface with LLM APIs across providers. Prompts are templated using TOML, enabling integration of new prompt variants. We employ `JavaParser` to parse ASTs, which are used to compute node-level distances between hunks, derive the tree diameter, and extract enclosing scopes such as methods and classes. For sparse retrieval, we use BM25; for dense retrieval, we employ `text-embedding-text-small-3` or Mini-LM, storing embeddings in a FAISS [60] index for nearest-neighbor lookup.

B. Models

This study conducts a comparative evaluation of six models, selected to represent both proprietary and open-source variants with diverse architectural characteristics. The proprietary models include GEMINI-2.5-FLASH (1M token context window) from Google, evaluated in non-reasoning mode via the official Gemini API, and NOVA-PRO (300K tokens) from Amazon. From OpenAI, we evaluate both a reasoning-capable model,

TABLE IV: Multi-hunk bug repair with LLMs

Type	Model	Pass	Test Fail	Comp Fail	Plausible@1	Cost
Open Source	Llama3.3 mistral-2407	46 45	73 106	253 221	12.37% 12.10%	\$1.43 \$4.08
Proprietary	nova-pro Gemini 2.5 Flash GPT-4.1 o4-mini	33 50 82 100	82 32 78 57	257 290 212 215	8.87% 13.44% 22.04% 26.88%	\$2.30 \$0.37 \$6.37 \$3.30

o4-MINI (200K tokens), and a non-reasoning variant, GPT-4.1 (1M tokens), accessed directly via the OpenAI API. Parameter counts for proprietary models are not publicly disclosed. The open-source models comprise MISTRAL-LARGE-2407 (128K tokens, 123B parameters) from Mistral AI and LLAMA3.3 (128K tokens, 70B parameters) from Meta. All models, except those from OpenAI and Google, were evaluated using full-parameter inference endpoints through Amazon Bedrock [61]. Our goal is not to perform exhaustive model benchmarking. Rather, we aim to assess the generalizability of our findings across LLMs with varying architectures.

C. Evaluation Metrics

We assess patch correctness based on whether the generated code parses, compiles, and passes all tests provided by the benchmark suite. Following prior work [62], we report *Plausible@1*, defined as the fraction of bugs for which the top-1 generated plausible patch passes the complete test suite:

$$\text{Plausible@1} = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \text{TestPass}(\text{patch}_1^b)$$

where \mathcal{B} is the set of bugs and patch_1^b is the first generated patch for bug b . A patch is counted as plausible if it compiles and passes all test cases, including the originally failing one.

We generate one patch per bug using standard decoding settings (e.g., temperature = 0.0) for all models except o4-MINI, whose API has a fixed temperature of 1.0. While decoding is configured to be deterministic at the API level, generation remains non-deterministic in practice due to low-level system factors such as thread scheduling, memory layout, and backend randomness [63].

D. Effectiveness of LLMs on Multi-Hunk Repair (RQ1)

To assess LLM effectiveness on multi-hunk repair, we ran all 372 bugs through the six models. Table IV summarizes the results, including the number of fully repaired bugs (Pass), test failures, compilation failures, plausible patch rate (Plausible@1), and estimated cost per run. Overall, the best-performing model was o4-MINI, which successfully repaired 100 of the 372 bugs (26.88%), followed by GPT-4.1 with 82 successful repairs (22.04%). Among open-source models, LLAMA3.3 achieved the highest accuracy, repairing 46 bugs (12.37%), followed closely by MISTRAL-2407, which repaired 45 bugs (12.10%).

Across the six LLMs, a total of 127 unique bugs are resolved. Among the two OpenAI models evaluated, we selected O4-MINI for further analysis due to its higher accuracy compared to GPT-4.1. We then assess individual effectiveness

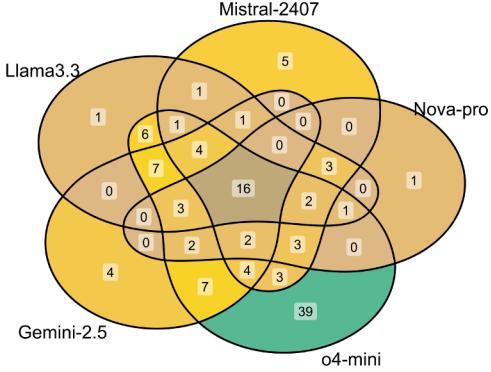


Fig. 2: Overlap of fixes across model families

and collective coverage using the following five models: O4-MINI, GEMINI-2.5-FLASH, NOVA-PRO, MISTRAL-2407, and LLAMA3.3. Specifically, we conduct an intersection analysis on the sets of bugs each model successfully resolves. Together, these five models generate *plausible* patches for 116 unique bugs. This finding underscores that substantial coverage in multi-hunk bug repair can be achieved through a carefully selected ensemble of models. Figure 2 presents a Venn diagram showing the overlap of *Plausible@1* patches among the selected models. Although all five models solve a shared pool of 16 bugs, each contributes additional unique fixes, most notably O4-MINI, which accounts for 39 unique repaired bugs.

When considering cost, model accuracy varies significantly relative to inference expense. GEMINI-2.5-FLASH achieves a *Plausible@1* of 13.44% at a total cost of just \$0.37, the lowest among all evaluated models. In contrast, MISTRAL-LARGE-2407, an open-source model, incurs a substantially higher cost of \$4.08 while yielding a comparable *Plausible@1* of 12.10%. This disparity highlights that increased inference cost does not necessarily translate to higher repair accuracy.

Overall, despite recent advances, LLMs still struggle with multi-hunk bug repair, as even the best model fails to fix over 70% of cases. The higher accuracy of O4-MINI highlights the benefit of using reasoning-capable models with greater test-time compute compared to non-reasoning models. However, the limited repair accuracy even in the best-performing model suggests that increasing test-time compute alone may not be sufficient, particularly for complex repair tasks [64].

E. Impact of Augmented Prompting (RQ2)

To assess the impact of different prompt augmentation strategies on LLM-based program repair, we evaluate three aspects: (a) retrieval-augmented example selection, (b) contextual granularity, and (c) feedback-based refinement. Given the high inference cost of large-scale evaluation across all prompt augmentation strategies, we limit this analysis to two representative models: O4-MINI, the most accurate proprietary model, and LLAMA3.3, the most effective open-source model in our evaluation.

We examine the effectiveness of *retrieval-based* prompting by comparing different ways of representing and retrieving similar repair examples. Each buggy example is encoded either as code tokens or as AST elements, capturing lexical and structural perspectives, respectively. For each representation, we apply one sparse retrieval method and two dense embedding-based methods to identify semantically similar past patches as examples. By pairing the two code representations with these three retrieval techniques, we construct six example-selection strategies, each retrieving the top-5 most similar patches to include in the prompt. The example patches used in retrieval-augmented prompting are selected from multi-hunk patches in the HUNK4J dataset that were fixed prior to the target bug—excluding the target bug itself—based on similarity in token or AST representations, depending on the retrieval configuration.

Table V reports the effectiveness of various prompt augmentation strategies in terms of accuracy, failure outcomes, and inference cost on O4-MINI and LLAMA3.3. The results show that token-level representations paired with dense retrieval outperform both AST-based code views and the sparse retrieval (BM-25) technique. For O4-MINI, the best configuration—token+text-embed—achieves a *Plausible@1* of 31.18%, surpassing the AST+text-embed variant (23.66%) by 7.52 percentage points. Among retrieval strategies, text-embed consistently yields higher accuracy than MiniLM and BM25 across both token and AST inputs. Similar trends are observed for LLAMA3.3, where token+text-embed reaches 20.16% *Plausible@1*, compared to 11.02% with AST+text-embed and 17.20% with token+BM25. This improvement can be attributed to how dense retrieval operates. Unlike sparse retrieval (BM25), which relies on token overlap and term frequency, dense retrieval (text-embed) encodes both the query and candidates into dense vectors using a neural embedding model. The embedding model was trained on token sequences, not structured ASTs. Consequently, similarity search is more accurate when both the query and candidates are represented as token sequences.

In addition to example selection, we study the effect of *contextual granularity* by varying the scope of code included in the prompt. We compare three levels of enclosing context: method, class, and file. The results for enclosing context show that wider context generally improves repair success. For O4-MINI, *Plausible@1* increases from 26.88% (method) to 28.30% (file), with a drop at the class level (23.66%), as class-level prompting often result in incomplete code generation. The reduced effectiveness at the class level can be attributed to LLMs interpreting the class snippet as an incomplete code fragment, often defaulting to patching only individual methods or lower-level constructs. In contrast, file-level prompts, which include import statements and top-level comments, provide clearer semantic cues, prompting the model to generate complete code. LLAMA3.3 shows a similar pattern: file-level prompting yields 15.59%, compared to 12.37% (method) and 11.56% (class). However, these gains come at a significant cost. For O4-MINI, file-level prompts cost \$43.35, 13 times

TABLE V: Multi-hunk bug repair with prompting strategies

Strategy	LLM	Prompt Mode	Pass	Test Fail	Comp Fail	Plausible @1	Cost
RAG (Example Selection)	Llama3.3	AST + BM25	64	76	232	17.20%	\$3.48
		AST + MiniLM	42	96	234	11.29%	\$9.09
		AST + text-embed	41	94	237	11.02%	\$7.94
	o4-mini	Token + BM25	70	75	227	18.82%	\$4.77
		Token + MiniLM	69	75	228	18.55%	\$3.75
		Token + text-embed	75	77	220	20.16%	\$4.00
Context Granularity	Llama3.3	AST + BM25	106	66	200	28.49%	\$6.53
		AST + MiniLM	91	74	207	24.46%	\$14.90
		AST + text-embed	88	79	205	23.66%	\$13.25
	o4-mini	Token + BM25	111	57	204	29.84%	\$8.46
		Token + MiniLM	114	57	201	30.65%	\$8.38
		Token + text-embed	116	59	197	31.18%	\$7.26
Feedback Loop	Llama3.3	Method	46	73	253	12.37%	\$1.43
		Class	43	113	216	11.56%	\$3.80
		File	58	132	182	15.59%	\$4.03
	o4-mini	Method	100	57	215	26.88%	\$3.30
		Class	88	146	138	23.66%	\$35.47
		File	105	186	81	28.30%	\$43.35

more than method-level (\$3.30), which achieves nearly comparable accuracy.

We further examine *feedback-based refinement*, which augments prompting with dynamic execution information derived from compilation failures or test failures, following prior work [65]–[67]. To limit cost, the feedback prompt uses the method-level context and the best retrieval setup (token+text-embed). Despite this conservative design, feedback yields the highest accuracy across all settings. O4-MINI achieves 35.75% Plausible@1 and LLAMA3.3 achieves 24.46%, both surpassing their respective static prompt baselines. Notably, the accuracy of O4-MINI with feedback surpasses that of the static prompt using file-level context (28.30%), while costing less than half as much (\$14.63 vs. \$43.35). These results indicate that feedback, driven by runtime information from failed tests or execution traces, can guide the model more effectively than simply expanding the prompt context. Moreover, feedback mitigates the limitations of narrow context by incorporating dynamic execution traces, which provide concrete, failure-specific guidance that static prompts lack.

Although prompting augmentations such as feedback and retrieval improve repair accuracy over vanilla LLMs, even with the best configurations, 64% of multi-hunk bugs still remain unfixed.

F. Repair Outcomes by Hunk Divergence and Proximity (RQ3)

To examine the conditions under which LLM-based program repair is most effective, we measure descriptive statistics of hunk divergence and spatial class-wise accuracy percentages. These metrics are derived from patches produced by the five models selected in RQ1, each representing a different model family.

Table VI reports the central tendencies of hunk divergence for both successfully and unsuccessfully fixed bugs. Addition-

ally, it presents the proportion of bugs correctly fixed within each spatial proximity class.

Successful repairs are consistently associated with low divergence across all five models (mean \approx 0.26–0.28; median \approx 0.24–0.29). Variance remains narrow (standard deviation $<$ 0.20), indicating high semantic alignment across generated patches. In contrast, unfixed bugs exhibit substantially higher hunk divergence (mean: 0.49–0.53; median: 0.45–0.53), along with broader variance and higher maxima (up to 1.60). These results reflect the increased difficulty LLMs face in coordinating complex, semantically diverse edits across scattered code regions.

Table VII reports our analysis across the best-performing prompting configurations, i.e., (1) Token + text-embed, (2) File Context, and (3) Feedback Loop, applied to O4-MINI and LLAMA3.3. Across all strategies, fixed bugs consistently exhibit lower divergence than unfixed bugs. The mean divergence for fixed bugs ranges from 0.28 to 0.40 (median: 0.28–0.39), while unfixed bugs display substantially higher scores (mean: 0.49–0.56; median: 0.45–0.53), indicating that greater heterogeneity within hunks correlates with repair failure. Among the evaluated strategies, Feedback Loop achieves the lowest divergence values for fixed bugs in both models (mean: 0.29–0.30; median: 0.30–0.31), and the highest percentage of repairs in the NUCLEUS and CLUSTER classes. This pattern suggests improved effectiveness on bugs involving spatially localized edits. These results demonstrate that prompting strategy significantly influences both semantic coherence and spatial locality of successful patches. Notably, the Feedback Loop configuration demonstrates the highest effectiveness in achieving low-divergence repairs within high-proximity contexts, thereby facilitating more precise and coordinated multi-hunk program repair.

The association between spatial proximity and repair success is first evident in Table VI. Repair success is highest in the NUCLEUS and CLUSTER classes, where hunks are in closer proximity, while substantially lower fix rates are observed in the SPRAWL and FRAGMENT categories, with the latter exhibiting no successful repairs. This trend persists across prompt variants, as shown in Table VII. Among the evaluated strategies, Feedback Loop consistently achieves superior accuracy in close proximity. For example, O4-MINI with Feedback Loop repairs 64.41% of bugs in the NUCLEUS class and 33.51% in CLUSTER, compared to only 22.00% and 27.27% in SPRAWL and FRAGMENT, respectively. A comparable pattern is observed for LLAMA3.3.

To further investigate whether the spatial proximity of hunks influences the success of LLM-based repair, we analyze the subset of bugs that were correctly fixed by at least one top-performing model from five distinct LLM families. Out of 372 multi-hunk bugs in HUNK4J, a total of 116 were successfully repaired by at least one model. Figure 3 presents an UpSet-style visualization of these 116 fixed bugs, grouped by spatial proximity class and by the specific combinations of models that produced successful patches. Each bar along the horizontal axis represents a unique combination of models (denoted by

TABLE VI: Hunk divergence for fixed and unfixed bugs, and spatial proximity distribution (% fixed) across model families

Model	Hunk Divergence (Fixed)				Hunk Divergence (Unfixed)				Spatial Proximity (% Fixed)				
	Median	Mean	Max	Std Dev	Median	Mean	Max	Std Dev	Nucleus	Cluster	Orbit	Sprawl	Fragment
mistral-2407	0.27	0.27	0.65	0.19	0.46	0.49	1.60	0.30	20.34	11.35	5.97	16.0	0.0
Llama3.3	0.26	0.27	0.65	0.17	0.47	0.50	1.60	0.30	27.12	9.73	11.94	8.0	0.0
nova-pro	0.24	0.28	0.65	0.19	0.45	0.49	1.60	0.30	11.86	8.11	7.46	12.0	0.0
Gemini 2.5	0.26	0.26	0.65	0.17	0.47	0.50	1.60	0.30	18.64	12.43	14.93	12.0	0.0
o4-mini	0.29	0.28	0.70	0.18	0.50	0.53	1.60	0.30	49.15	23.78	23.88	22.0	0.0

TABLE VII: Hunk divergence for fixed and unfixed bugs, and spatial proximity distribution (% fixed) across best prompt strategies

Model	Technique Variant	Divergence Score (Fixed)				Divergence Score (Unfixed)				Spatial Proximity (% Fixed)				
		Median	Mean	Max	Std Dev	Median	Mean	Max	Std Dev	Nucleus	Cluster	Orbit	Sprawl	Fragment
o4-mini	Token + text-embed	0.34	0.31	0.96	0.21	0.50	0.54	1.60	0.30	54.24	30.81	22.39	20.00	18.18
	File Context	0.39	0.40	1.08	0.23	0.46	0.49	1.60	0.31	37.29	33.51	13.43	22.00	9.09
	Feedback Loop	0.31	0.30	0.96	0.20	0.53	0.56	1.60	0.30	64.41	33.51	28.36	22.00	27.27
Llama3.3	Token + text-embed	0.28	0.28	0.78	0.21	0.47	0.52	1.60	0.29	33.90	18.92	17.91	14.00	9.09
	File Context	0.36	0.36	1.08	0.23	0.45	0.49	1.60	0.30	18.64	17.30	7.46	20.00	0.00
	Feedback Loop	0.30	0.29	0.96	0.20	0.49	0.52	1.60	0.30	42.37	21.62	25.37	14.00	18.18

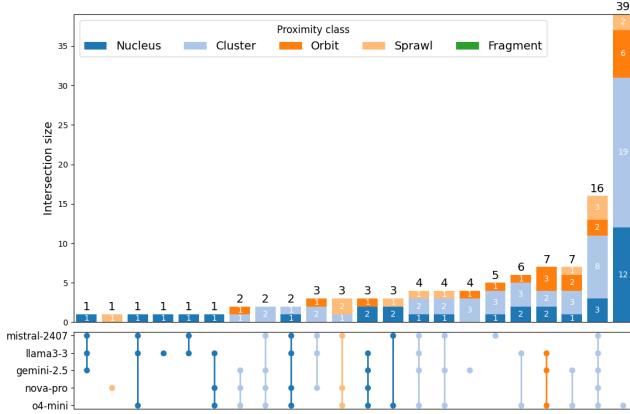


Fig. 3: Upset-style plot showing bugs correctly repaired by the selected models, grouped by spatial proximity class. Bars indicate intersections among models; color segments denote proximity categories.

filled circles in the matrix below), and bars are ordered from left (representing the fewest bugs) to right (representing the most bugs). Within each bar, segments are color-coded by proximity class, NUCLEUS (dark blue), CLUSTER (light blue), ORBIT (orange), SPRawl (peach), and FRAGMENT (green) with each segment labeled by the exact bug count it represents. The total number of bugs in each model intersection is annotated above the corresponding bar. The lower panel presents a matrix of filled circles and vertical connectors: each row denotes one of the 5 models (from top to bottom: MISTRAL-2407, LLAMA3.3, GEMINI-2.5-FLASH, NOVA-PRO, o4-MINI), and each column corresponds to the same model combination that generated the bar above. A filled circle indicates that the model participated in that combination; connectors link the participating models.

Two trends are evident. First, the majority of successful repairs fall into the NUCLEUS and CLUSTER categories, which represent cases where the relevant code elements are tightly co-located. These proximity classes dominate across both

shared and model-specific intersections. Second, cases in the ORBIT and SPRAWL categories, where relevant context is more dispersed, are less frequently solved. The LLM alone was unable to fix any bug in the FRAGMENT category, which reflects extreme dispersion or disconnection of relevant context (Table VI). The largest set of bugs (39 bugs) is uniquely attributable to O4-MINI, while the second-largest (16 bugs) was solved by all five models. Smaller intersections reveal model-specific capabilities or divergences, particularly in handling less proximal context.

Collectively, these findings suggest that successful LLM-based repairs for multi-hunk bugs are characterized by (1) semantically coherent edits with low hunk divergence and (2) spatially localized dependencies with high proximity.

G. Divergence, Proximity, and Repair Difficulty (RQ4)

We now investigate whether hunk divergence and spatial proximity function as predictive indicators of LLM repair difficulty, i.e., the likelihood that a model fails to synthesize a correct patch. This analysis considers two dimensions of multi-hunk patches: the degree of intra-patch variability (hunk divergence) and the extent of spatial dispersion across the program hierarchy (spatial proximity).

To assess this, we first partitioned the data for the Top-5 LLMs into two independent samples, fixed and unfixed bugs, and applied a two-sample Wilcoxon rank-sum test to their divergence scores. Across all five top-performing models, the divergence distributions for fixed and unfixed bugs differ significantly. All Wilcoxon tests yield p -values below 10^{-4} , with three models reporting $p < 10^{-7}$ and the largest p -value at 2.6×10^{-5} . These results confirm that higher divergence is strongly associated with failure to generate a correct patch. We also computed Cliff's Delta to assess the effect sizes. Under LLM-only and feedback-loop prompting, the delta values for o4-MINI were 0.533 and 0.554, respectively. Both values exceed the threshold of 0.474, indicating large effect sizes, that is, substantial differences in hunk divergence between successfully and unsuccessfully repaired bugs. The consistent

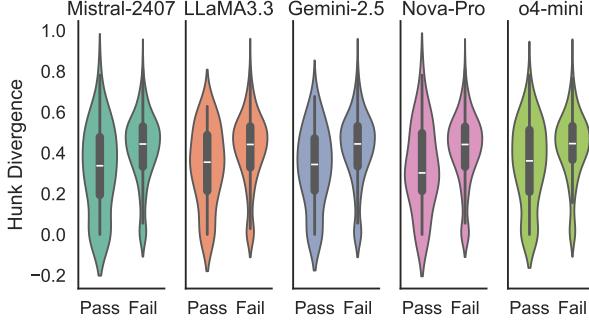


Fig. 4: Faceted violin plots of average hunk divergence, grouped by model and split by outcome.

statistical significance observed across models suggests that hunk divergence may serve as a meaningful indicator of patch complexity and the effectiveness of LLM-based repair.

We observe similar trends when comparing augmented prompting configurations (Table VII). Among the six prompting variants, four yield p -values below 10^{-7} , one yields $p = 3.8 \times 10^{-3}$, and all remain below the conventional threshold of 0.05. This demonstrates that the distinction between fixed and unfixed bugs based on divergence holds across different prompt strategies, further validating its predictive value.

Figure 4 presents faceted violin plots showing the distribution of average hunk divergence scores for each model, separated by outcome. Each violin illustrates the density of divergence values, while the embedded boxplot indicates the median and interquartile range, capturing the central 50% of the data. Fixed bugs are consistently associated with lower divergence scores across all models. This clear and consistent pattern demonstrates that hunk divergence is a discriminative signal for reasoning about multi-hunk bug complexity and the success of LLM-based repair.

In addition to divergence, spatial proximity also emerges as a strong categorical predictor. As shown in Figure 3, repair success declines consistently from NUCLEUS to SPRAWL, with the FRAGMENT class remaining entirely unsolved across all models. This trend suggests that as fault-relevant context becomes more dispersed, LLMs are increasingly unable to synthesize correct patches. The complete absence of fixes in the FRAGMENT category indicates a hard boundary for current model capabilities in handling non-local, scattered dependencies.

Our analysis shows that LLMs exhibit a significant drop in repair accuracy as hunk divergence increases and spatial proximity decreases. Bugs with high intra-patch variation or dispersed hunks across methods, files, or packages remain largely unrepairs. These findings underscore the need for models and prompts that explicitly account for lexical and structural dispersion in multi-hunk patches.

IV. THREATS TO VALIDITY

HUNK4J is derived exclusively from Java projects, which may limit generalizability to other languages. Although these

bugs are real-world and diverse, their distribution reflects the design and testing philosophy of Java projects, which may bias the scope of repair challenges. We evaluate on Java because it is a popular language and use the widely adopted Defects4J benchmark. Our metrics, hunk divergence and spatial proximity, and our evaluation steps do not rely on language-specific features, so the underlying insights should apply to other languages; nevertheless, additional experiments on non-Java datasets are required to empirically validate this hypothesis and assess generalizability beyond Java and Defects4J.

Our goal is to study how the structure of real developer-written patches affects LLM repair. We use the original patches from Defects4J because they reflect how developers fixed the bugs. A simpler or semantically equivalent patch may exist for some bugs; however, we intentionally compute hunk divergence and spatial proximity from the actual developer patch, not hypothetical alternatives. This provides us with a consistent and realistic approach to studying what makes multi-hunk repair challenging.

We use the Plausible@1 metric to reduce inference cost, which may underestimate model accuracy in iterative settings. Decoding parameters and prompt templates are fixed across models to ensure fairness, potentially favoring some architectures. However, our aim is comparative analysis under uniform conditions and not model-specific optimization.

Our evaluation assumes perfect fault localization, following prior LLM-based repair studies to isolate patch generation quality [10], [17], [18], [33]–[35], [38], [41]. While this sets an upper bound on accuracy, real-world scenarios often involve imperfect or incomplete localization. Notably, even with perfect localization, LLMs struggle with multi-hunk repairs, failing on all Fragment-class bugs (Table VI), highlighting challenges in semantic coordination beyond localization.

LLMs are accessed via third-party APIs, and execution can vary due to backend updates or system-level nondeterminism. Although decoding is configured to be deterministic (temperature = 0), minor variations may still occur. We mitigate this by logging all inputs, outputs, and execution metadata for reproducibility.

Our implementation uses JavaParser to extract ASTs and compute divergence metrics such as node-level distance and tree diameter. Alternative parsers (such as javalang) may yield slight variations due to differences in AST tree construction, though all models are evaluated under the consistent criteria.

To support reproducibility, we release BIRCH, all scripts, prompts, and the HUNK4J dataset [68].

V. RELATED WORK

Most APR benchmarks offer limited support for multi-hunk bugs. While Defects4J [51], Bugs.jar [69], BugsInPy [70], and LMDefects [41] include them, others offer limited coverage [71]–[73] or focus on single-hunk edits [74]–[76]. Existing multi-hunk LLM repair studies rely on synthetic [41], function-level [42], or algorithmic [40] benchmarks, limiting

generalizability. We introduce HUNK4J, a real-world, multi-hunk benchmark derived from Defects4J, augmented with divergence metrics for evaluating LLM-based repair.

Prior studies have applied change metrics for fault prediction. Graves et al. [77] used coarse-grained measures such as file and modification counts; D’Ambros et al. [78] highlighted the predictive power of change coupling and scatter; and Ferzund et al. [79] proposed hunk-level metrics. In contrast, we quantify semantic and structural heterogeneity in multi-hunk patches to assess LLM-based repair effectiveness.

Prior work on multi-hunk repair with LLMs [35], [38]–[42], [80] exhibits wide variation in datasets, prompting strategies, and evaluation protocols, often reporting aggregate results without isolating multi-hunk accuracy. While prior studies have examined patch properties such as hunk divisibility [80], they do not account for spatial dispersion or intra-patch heterogeneity. We address this limitation by introducing *Hunk Divergence* and *Spatial Proximity*, two metrics that capture lexical, structural, and hierarchical relationships between hunks to enable complexity-aware analysis.

VI. CONCLUSION

We introduce the first characterization of multi-hunk bugs through the lens of intra-patch divergence. Our metric, *hunk divergence*, captures lexical, structural, and file-separation dissimilarity between hunks, while *spatial proximity* classifies their distribution across the codebase. Our findings reveal that multi-hunk bugs exhibit substantial variability in divergence, with no single model achieving consistently high accuracy across this spectrum. Collectively, these contributions lay the foundation for developing *divergence-aware* techniques capable of addressing the unique challenges posed by multi-hunk bugs. In future work, we aim to develop a divergence-aware model selector that routes bugs to the most capable and cost-efficient LLM based on predicted repair complexity.

REFERENCES

- [1] M. Yasunaga and P. Liang, “Graph-based, self-supervised program repair from diagnostic feedback,” in *37th International Conference on Machine Learning*. JMLR.org, 2020.
- [2] N. Jiang, T. Lutellier, and L. Tan, “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair,” in *43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.
- [3] E. Mashhadí and H. Hemmati, “Applying codebert for automated program repair of java simple bugs,” in *18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.
- [4] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “A Syntax-Guided Edit Decoder for Neural Program Repair,” in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, p. 341–353.
- [5] H. Ye, M. Martinez, and M. Monperrus, “Neural Program Repair with Execution-Based Backpropagation,” in *44th International Conference on Software Engineering*. ACM, 2022, p. 1506–1518.
- [6] H. Hata, E. Shihab, and G. Neubig, “Learning to Generate Corrective Patches using Neural Machine Translation,” *arXiv:1812.07170*, 2019.
- [7] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, “A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, 2022.
- [8] T. H. M. Le, H. Chen, and M. A. Babar, “Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges,” *ACM Comput. Surv.*, vol. 53, no. 3, 2020.
- [9] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation,” *ACM Transactions on Software Engineering and Methodology*, 2019.
- [10] M. Namavar, N. Nashid, and A. Mesbah, “A Controlled Experiment of Different Code Representations for Learning-Based Bug Repair,” *Empirical Software Engineering Journal*, 2022.
- [11] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair,” in *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2020, p. 101–114.
- [12] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, “SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
- [13] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, “A Survey of Learning-based Automated Program Repair,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, 2023.
- [14] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, “Evolving Paradigms in Automated Program Repair: Taxonomy, Challenges, and Opportunities,” *ACM Computing Surveys*, 2024.
- [15] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, “DeepDelta: Learning to Repair Compilation Errors,” in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, p. 925–936.
- [16] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs,” in *International Conference on Learning Representations*, 2020.
- [17] N. Nashid, M. Sintaha, and A. Mesbah, “Embedding context as code dependencies for neural program repair,” in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 95–106.
- [18] M. Sintaha, N. Noor, and A. Mesbah, “Dual Slicing-Based Context for Learning Bug Fixes,” *Transactions on Software Engineering and Methodology (TOSEM)*, p. 27 pages, 2023.
- [19] C. Le Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.
- [20] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
- [21] W. Weimer, Z. P. Fry, and S. Forrest, “Leveraging program equivalence for adaptive program repair: Models and first results,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.
- [22] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 802–811.
- [23] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair,” in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 602–614.
- [24] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in java programs,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.
- [25] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.
- [26] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, “Tare: Type-aware neural program repair,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1443–1455.
- [27] D. Yang, X. Mao, L. Chen, X. Xu, Y. Lei, D. Lo, and J. He, “Transplantfix: Graph differencing-based code transplantation for automated program repair,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [28] S. Saha, R. K. Saha, and M. R. Prasad, “Harnessing evolution for multi-hunk program repair,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE. IEEE Press, 2019, pp. 13–24.

- [29] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.
- [30] Y. Yuan and W. Banzhaf, “A hybrid evolutionary system for automatic software repair,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1417–1425.
- [31] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues, “VarFix: balancing edit expressiveness and search effectiveness in automated program repair,” in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 354–366.
- [32] Y. Li, S. Wang, and T. N. Nguyen, “Dear: A novel deep learning-based approach for automated program repair,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 511–523.
- [33] H. Ye and M. Monperrus, “ITER: Iterative Neural Repair for Multi-Location Patches,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. ACM, 2024.
- [34] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.
- [35] C. S. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. Association for Computing Machinery, 2022, p. 959–971.
- [36] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, “A Deep Dive into Large Language Models for Automated Bug Localization and Repair,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, 2024.
- [37] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, “A critical review of large language model on software engineering: An example from chatgpt and automated program repair,” *arXiv preprint arXiv:2310.08879*, 2023.
- [38] I. Bouzenia, P. Devanbu, and M. Pradel, “RepairAgent: An Autonomous, LLM-Based Agent for Program Repair,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 694–694.
- [39] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, “An empirical study on fine-tuning large language models of code for automated program repair,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1162–1174.
- [40] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. F. Bissyandé, C. L. Goues, and S. Jin, “Multi-Objective Fine-Tuning for Enhanced Program Repair with LLMs,” *arXiv preprint arXiv:2404.12636*, 2024.
- [41] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, “Automated Repair of Programs from Large Language Models,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23. IEEE Press, 2023, p. 1469–1481.
- [42] J. Xiang, X. Xu, F. Kong, M. Wu, H. Zhang, and Y. Zhang, “How Far Can We Go with Practical Function-Level Program Repair?” *arXiv preprint arXiv:2404.12833*, 2024.
- [43] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2002, p. 311–318.
- [44] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, “Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, 2024.
- [45] A. Shirafuji, M. M. Rahman, M. F. Ibne Amin, and Y. Watanobe, “Program Repair with Minimal Edits Using CodeT5,” in *2023 12th International Conference on Awareness Science and Technology (iCAST)*, 2023, pp. 178–184.
- [46] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep API learning,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. Association for Computing Machinery, 2016, p. 631–642.
- [47] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 135–146.
- [48] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, “On learning meaningful assert statements for unit test cases,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, 2020, p. 1398–1409.
- [49] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, “A survey of learning-based automated program repair,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, 2023.
- [50] C. Koutcheme, S. Sarsa, J. Leinonen, L. Haaranen, and A. Hellas, “Evaluating Distance Measures for Program Repair,” in *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*. Association for Computing Machinery, 2023, p. 495–507.
- [51] R. Just, D. Jalali, and M. D. Ernst, “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*, 2014, pp. 437–440.
- [52] Jsoup 56, “Multi-Hunk Patch for Jsoup Bug 56,” 2025, accessed: May 26, 2025. [Online]. Available: https://github.com/nashid/birch/tree/main/hunk4j/patches/Jsoup_56_patch.patch
- [53] Jsoup 87, “Multi-Hunk Patch for Jsoup Bug 87,” 2025, accessed: May 26, 2025. [Online]. Available: https://github.com/nashid/birch/tree/main/hunk4j/patches/Jsoup_87_patch.patch
- [54] Chart 4, “Multi-Hunk Patch for Chart Bug 4,” 2025, accessed: May 26, 2025. [Online]. Available: https://github.com/nashid/birch/tree/main/hunk4j/patches/Chart_4_patch.patch
- [55] JacksonDatabind 103, “Multi-Hunk Patch for Jackson-Databind Bug 103,” 2025, accessed: May 26, 2025. [Online]. Available: https://github.com/nashid/birch/tree/main/hunk4j/patches/JacksonDatabind_103_patch.patch
- [56] Mockito 6, “Multi-Hunk Patch for Mockito Bug 6,” 2025, accessed: May 26, 2025. [Online]. Available: https://github.com/nashid/birch/tree/main/hunk4j/patches/Mockito_6_patch.patch
- [57] Closure 144, “Multi-Hunk Patch for Closure Bug 144,” 2025, accessed: May 27, 2025. [Online]. Available: https://github.com/nashid/birch/tree/main/hunk4j/patches/Closure_144_patch.patch
- [58] Ollama, “Ollama: Run open-source LLMs locally,” <https://ollama.com/>, 2024, accessed: 2024-05-26.
- [59] LiteLLM, “LiteLLM: A Unified Toolkit for LLM Inference,” <https://github.com/BerriAI/litellm>, 2024, accessed: 2025-05-27.
- [60] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with gpus,” in *IEEE Transactions on Big Data*. IEEE, 2019.
- [61] Amazon Web Services, “Amazon Bedrock,” <https://aws.amazon.com/bedrock/>, 2024, accessed: 2024-05-26.
- [62] A. Silva and M. Monperrus, “Repairbench: Leaderboard of frontier models for program repair,” *arXiv*, Tech. Rep. 2409.18952, 2024. [Online]. Available: <https://arxiv.org/abs/2409.18952>
- [63] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, “An Empirical Study of the Non-Determinism of ChatGPT in Code Generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, Jan. 2025.
- [64] C. V. Snell, J. Lee, K. Xu, and A. Kumar, “Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning,” in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: <https://openreview.net/forum?id=4FWAwZtd2n>
- [65] T. Shabani, N. Nashid, P. Aliani, and A. Mesbah, “Dockerfile Flakiness: Characterization and Repair,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2025, pp. 774–774.
- [66] H. Ruan, Y. Zhang, and A. Roychoudhury, “SpecRover: Code Intent Extraction via LLMs,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 617–617.
- [67] N. Nashid, I. Bouzenia, M. Pradel, and A. Mesbah, “Issue2test: Generating reproducing test cases from issue reports,” 2025.
- [68] BIRCH, “Characterizing Multi-Hunk Patches: Divergence, Proximity, and LLM Repair Challenges,” 2025. [Online]. Available: <https://github.com/nashid/birch/>
- [69] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, “Bugs. jar: A large-scale, diverse dataset of real-world java bugs,” in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 10–13.

- [70] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, “BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. Association for Computing Machinery, 2020, p. 1556–1560.
- [71] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, “Bears: An extensible java bug benchmark for automatic program repair studies,” in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 468–478.
- [72] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The ManyBugs and IntroClass benchmarks for automated repair of C programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.
- [73] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, “BUGSJJS: a benchmark and taxonomy of JavaScript bugs,” *Journal of Software Testing, Verification and Reliability (STVR)*, vol. 31, 2020.
- [74] Z. Chen and M. Monperrus, “The codrep machine learning on source code competition,” *arXiv preprint arXiv:1807.03200*, 2018.
- [75] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: a multilingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. Association for Computing Machinery, 2017, p. 55–56.
- [76] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the manysstubs4j dataset,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.
- [77] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, “Predicting fault incidence using software change history,” in *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. IEEE, 2000, pp. 273–282.
- [78] M. D’Ambros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: a benchmark and an extensive comparison,” *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.
- [79] J. Ferzund, S. N. Ahsan, and F. Wotawa, “Software change classification using hunk metrics,” in *2009 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 349–358.
- [80] Q. Xin, H. Wu, J. Tang, X. Liu, S. P. Reiss, and J. Xuan, “Detecting, Creating, Repairing, and Understanding Indivisible Multi-Hunk Bugs,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2747–2770, 2024.