Intent-Aware Defect Pattern Extraction from Singular Examples

JIACHEN HAN, Tianjin University, China
FENGJIE LI, Tianjin University, China
JIAJUN JIANG*, Tianjin University, China
RUIHANG FAN, Tianjin University, China
YINGFEI XIONG, Peking University, China
LINJIE PAN, Huawei Cloud, China
BILIAN WANG, Huawei Cloud, China
JUNJIE CHEN, Tianjin University, China

As software systems grow in complexity, particularly in safety-critical domains, detecting defects early is crucial to prevent severe failures. Among existing automated defect detection methods, rule-based systems excel at catching well-understood mistakes but are hard to generalize beyond their predefined rules. To address this challenge, we introduce DslGen, a novel approach that automatically extracts high-quality defect patterns from singular defect repair examples by leveraging large language models (LLMs). Specifically, it first infers the semantic intent of a defect repair for capturing the defect's essential characteristics via LLM-powered analysis, then generates precise constraint rules to define the defect pattern. Unlike prior methods, DslGen does not require numerous examples or rigid abstraction schemes, enabling robust generalization even from a single fix. To evaluate the effectiveness of DslGen, we evaluated it in two distinct scenarios: (1) controlled experiments and (2) a simulated real-world defect detection pipeline. In the controlled setting, results show that DslGen outperforms all baselines, achieving a precision up to 81.1% and a recall of over 73.0%. In the simulated scenario, DslGen also performed best with an average improvement of 425.1% over state-of-the-art competitors (including GPT-4o). A user study with 15 experienced developers further confirmed DslGen's practicality, with participants praising the readability and generalizability of its generated DSLs and expressing strong adoption intent.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Domain specific languages**.

Additional Key Words and Phrases: Pattern mining, Defect detection, Large language model

*Corresponding author.

Authors' Contact Information: Jiachen Han, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China, jiachen_han@tju.edu.cn; Fengjie Li, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China, fengjie@tju.edu.cn; Jiajun Jiang, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China, jiangjiajun@tju.edu.cn; Ruihang Fan, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China, frexvite@tju.edu.cn; Yingfei Xiong, Key Lab of HCST, MOE, School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn; Linjie Pan, Huawei Cloud, Beijing, China, panlinjie@huawei.com; Bilian Wang, Huawei Cloud, Beijing, China, wangbilian@huawei.com; Junjie Chen, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China, junjiechen@tju.edu.cn.

ACM 1557-7392/2026/6-ART
https://doi.org/10.1145/3820049

1 Introduction

As software systems become increasingly complex and are deployed in safety-critical domains, ensuring their quality has become essential [38]. However, software defects are inevitably introduced during development and maintenance, and can lead to serious usability failures or even catastrophic incidents [56]. For example, the Log4Shell vulnerability enabled remote code execution, exposing many organizations to data breaches and causing significant financial losses [16]. Consequently, there is an urgent need to detect software-quality issues before they escalate.

However, as codebases grow in size and interconnectivity, detecting potential defects becomes increasingly difficult [20, 28]. Effective defect identification usually demands deep functional knowledge and high expertise, making purely manual debugging both time-consuming and, in many cases, impractical at scale [46]. To address these challenges, the software engineering community has embraced automated defect-detection methods that can efficiently uncover defects across large codebases [47, 50]. In particular, static defect detection stands out due to its effectiveness and efficiency. It can spot suspicious patterns and coding violations using either rule-based heuristics [17, 22], which usually apply handcrafted defect patterns derived from coding standards and expert knowledge, or machine-learning models [24, 47] , which learn statistical indicators of defects from large, labeled code corpora. Although rule-based systems can automatically apply defect patterns once defined, these patterns must be manually crafted based on expert knowledge of defect characteristics, analysis tools, and code contexts. As a result, while excelling at capturing well-understood mistakes, such systems face significant challenges in scaling pattern construction to cover a large and diverse space of defects, as each additional pattern incurs substantial manual effort [26]. In contrast, machine-learning methods can capture new defect patterns but require large, high-quality training datasets, which are scarce—especially for emerging defects. Moreover, their predictions are often opaque and their performance struggles to generalize across projects and coding styles [9], undermining developer trust and hindering adoption. These limitations highlight the need for approaches that can automate the acquisition of defect patterns while preserving their explicit and interpretable rule-based form. In this paper, we address this challenge by further advancing automated approaches for constructing rule-based defect patterns.

As mentioned earlier, rule-based defect detection methods often rely on defect patterns that can be applied automatically once defined. Specifically, such a defect pattern is typically defined as a set of constraints that capture the essential structural and contextual properties of the defective code [45]. The core challenge in automated pattern extraction is therefore determining which context constraints are truly critical for capturing a certain defect pattern [10, 25]. Consider null-pointer dereferences: while numerous contextual features of the surrounding code could be considered, the decisive constraints are *whether a variable may be null and whether a null-check is performed before its dereference*. Identifying and prioritizing such critical constraints over incidental context illustrates the central challenge of automated pattern extraction.

To address this challenge, existing approaches typically mine from multiple, repetitive defect-fix examples, where the frequently appeared contextual features among these examples are typically deemed as the potential indicative constraints of the defect [42]. In practice, however, real-world defect repositories – especially those involving newly discovered vulnerabilities – rarely provide enough diverse defect-fix examples to support reliable generalization [48]. When examples are insufficient or lack diversity, such automated pattern mining struggles to distinguish truly critical constraints from incidental features, since recurring elements provide reliable signals of importance only when validated across diverse contexts. In the extreme case of a single defect-fix, it becomes especially difficult to determine which code properties should be treated as essential. The difficulty is compounded by the fact that the critical constraints of many defect patterns lie in subtle semantic dependencies or design-level properties, which extend beyond surface syntax and are hard to capture automatically [23, 53]. To mitigate the limitations of single-example settings, recent methods either apply predefined abstraction schemes [30, 41] to

derive constraints, or exploit large codebases to filter out overly frequent contexts on the assumption that such ubiquitous features are unlikely to be critical for the defect [25]. While such strategies lessen the need for multiple examples, they tend to produce low-quality patterns – both by overlooking the semantic and syntactic cues that truly characterize the defect and by mistakenly elevating trivial, ubiquitous features to the status of defect-related constraints. Consequently, automatically extracting high-quality patterns from limited (e.g., only one) examples remains an open challenge.

Large language models (LLMs) have recently shown strong performance across a range of software-engineering tasks – ranging from code generation to semantic understanding [8, 13, 21, 32, 39, 55]. This semantic understanding provides a foundation for tackling the challenge of identifying critical constraints from limited examples. Building on this ability to comprehend code semantics, in this paper, we introduce DslGen, a novel defect-pattern extraction approach. The core insight of DslGen is that **understanding the semantic intent of a defect repair provides valuable guidance for identifying the essential constraints that precisely capture the contextual features of a certain type of defects**. For instance, in null-pointer dereference bugs, the fix's intent lies in recognizing that the dereferenced variable may be null and that a null-check is required before its use. From this intent, we may identify the critical constraints: *the variable must be potentially null, and the dereference must lack a guarding null-check.* Incidental properties, such as variable names or statement order, are irrelevant to the pattern. As a consequence, DslGen puts this insight into practice through a two-stage process: (1) Intent Extraction: We design tailored prompts that guide the LLM to infer the fixing intent of a code change, aiming to uncover the defect's root cause and thereby capture its essential characteristics. This intent serves as explicit guidance for subsequent constraint derivation. (2) Constraint Construction: Guided by the extracted intent and defect's root cause, we employ a program-controlled multi-turn interaction with the LLM, through which relevant code elements and contextual features are iteratively abstracted and organized into precise constraint rules. In particular, to facilitate human review and refinement, we further provide an adapter that automatically translates the extracted constraints into a human-readable Domain-Specific Language (DSL) program, enabling developers to understand the semantics of the pattern and refine it when necessary.

To comprehensively evaluate DslGen, we conducted extensive experiments on two datasets and compared it with state-of-the-art methods. Specifically, we assessed DslGen in two distinct defect detection scenarios: (1) a controlled experiment with pairwise instances of the same defect and (2) a simulated real-world development pipeline, where defect detection is triggered once a new code change is committed. The results demonstrate that DslGen significantly outperforms all baseline methods. In the controlled scenario, DslGen achieved a precision up to 81% and a recall of over 73%, surpassing the strongest baseline by an average of 15.6% precision and 14.0% recall across datasets. In the simulated scenario, DslGen also achieved the highest effectiveness, outperforming the best baseline, GPT-4o, with up to 49.8% (0.665 *vs.* 0.444) higher F1 score across datasets. An ablation study further validated the contribution of each component in DslGen to its overall performance. Additionally, to assess real-world applicability, we conducted a user study with 15 experienced developers. Their feedback confirmed DslGen's practical value – they consistently rated the generated DSLs as highly readable and generalizable, and expressed willingness to adopt DslGen in their development workflows.

In summary, this paper makes the following contributions:

- We introduce DslGen, a novel automated defect pattern extraction framework that leverages LLMs to infer the semantic intent of defect repairs and derive high-quality constraint rules even from a single example.
- We curate and publicly release a rigorously validated dataset of 1287 repetitive defect-fix pairs mined from real-world open-source projects, establishing a new benchmark for pattern mining research.
- We conduct extensive experiments – on both our new dataset and a widely used benchmark – to demonstrate that DslGen substantially outperforms existing techniques.

- We have published the implementation of our approach to support reproducibility and further exploration. https://github.com/jiachenhan/DSLGEN

2 Related Work

2.1 Manual Modeling of Bug Patterns

Rule-based approaches represent one of the earliest and most enduring foundations for static bug and vulnerability detection. Initial efforts in this direction focused on encoding expert knowledge into general-purpose constraint rules to capture common programming errors. A seminal work by Engler et al. [17] proposed six types of constraint rules to detect bugs. Tools like FindBugs [22] and PMD [5] emerged as early rule-based analyzers for Java, detecting common bugs. Similarly, Yamaguchi et al. [54] model vulnerabilities as subgraphs over code property graphs, while CodeQL [1] and Semgrep [7] support logic-based rule authoring for customizable vulnerability detection. Pan et al. [45] manually analyzed historical bug fixes to construct 27 fix patterns, illustrating how patches can inform the design of reusable bug detection patterns.

2.2 Automated Bug Pattern Discovery

Recent work has shifted from manual pattern creation to automated techniques for defect discovery. Some approaches [12, 34, 43] mine frequent code patterns from large corpora to detect deviations as potential defects. However, these methods require substantial data and often miss rare or project-specific bugs. To address this issue, several techniques [10, 36, 42, 49] infer bug patterns and associated transformations for repair from multiple fix examples by generalizing over observed edits. Among them, Getafix [10] and Genesis [36] generate multiple reusable patterns to account for different abstraction levels, while Lase [42] and REFAZER [49] aim to synthesize a single generalized bug and transformation pattern that satisfies all provided examples. Other methods [25, 30, 41] seek to extract reusable patterns even from individual patches. Specifically, FixMiner [30] iteratively clusters AST-level edits to mine context-rich bug and repair patterns. SYDIT [41] generates transformation rules associated to a certain bug pattern from a single annotated change by identifying consistent edit operations. GenPat [25] further combines single-example abstraction with statistical signals from large code corpora to guide pattern generalization. While these approaches automate bug or fix pattern extraction, their reliance on structural similarities without incorporating semantic information limits their ability to capture semantically meaningful defect characteristics.

2.3 Transformation by Example in Model-Driven Engineering

Several studies also learn transformation knowledge from concrete examples, and in the model-driven engineering (MDE) community, this line of work is known as model transformation by example [29]. Model transformations automatically generate a target model from a given source model, which can then drive downstream development activities such as design, maintenance, and refactoring. These studies aim to reduce the effort of writing model transformations, which are required to support such development activities, by deriving transformation rules or programs from example source–target model pairs. Early work on model transformation by example derived transformation rules by iteratively and interactively generalizing over prototypical pairs of source and target models [51]. On this basis, Varr'o and Balogh proposed a learning-based approach using inductive logic programming to infer transformation rules from example source–target model pairs [52]. More recently, statistical machine translation has been explored to learn model transformations from example source–target model pairs [11]. Overall, these studies primarily focus on synthesizing model transformation rules or programs from examples. Different from these works, we do not aim to synthesize executable transformations from source–target examples. Instead, we generate declarative defect patterns from defect-fix examples, intending to characterize defect-related code elements rather than generate target artifacts. Thus, while both directions learn from examples, prior MDE

```
1 public static String migrate(String xml) {
2    Document xmlDoc;
3    try {
4 -    xmlDoc = new SAXReader().read(new StringReader(xml));
5 +    SAXReader reader = new SAXReader();
6 +    /* Prevent XXE attack as
          the xml might be provided by malicious users */
7 +    reader.setFeature("http://apache.org/xml/...",true);
8 +    xmlDoc = reader.read(new StringReader(xml));
9 -  } catch (DocumentException e) {
10+  } catch (DocumentException | SAXException e) {
11     throw new RuntimeException(e);
12   }
13
14  ......
15 }
```

(a) An example code repair for the real-world XXE vulnerability (CVE-2021-21250). Partial defect-related elements are highlighted with color blocks, whose numbering on the right side of each block corresponds to the nodes in Fig. 4.

```
1  functionDeclaration fd_1 where
2    and(
3      fd_1.body contain objectCreationExpression oc_1 where
4        oc_1.name == "SAXReader",
5      not(
6        fd_1.body contain functionCall fc_2 where
7          and(
8            fc_2.name == "setFeature",
9            fc_2.arguments contain alias1 where
10             and(
11               alias1 is stringLiteral,
12               alias1.value match "http://apache.org/xml/.*"
13             )
14         )
15     )
16   );
```

(b) A DSL program including a set of constraints capturing the essential vulnerability features, describing code that creates a SAXReader object without enabling the corresponding security features via setFeature.

Fig. 1. From bug-fix pair to DSL program

work focuses on transformation synthesis, whereas our approach focuses on defect-pattern abstraction and detection.

3 Motivation

In this section, we present a real-world example to illustrate the challenges of extracting a pattern from one single example and to highlight the limitations of existing methods. This example motivates the intuition and necessity of our approach. Fig. 1a shows the bug fix for an *XML External Entity Injection* vulnerability (CVE-2021-21250) [15], with lines prefixed by "-" indicating deletions and those prefixed by "+" indicating insertions. In this example, the faulty code (line 4) parses untrusted XML with `SAXReader().read()` while failing to disable external entity resolution. This flaw enables attackers to submit crafted XML documents containing external entity declarations, potentially exposing sensitive files. For example, an attacker may declare an entity referencing a local file (e.g., `/etc/passwd`) and embed it through a `DOCTYPE` declaration. When such input is processed by a parser like `SAXReader` without disabling entity expansion, the entity is resolved and the file's content is injected into the parsed output. This can result in sensitive data leakage.

When applied to this example, existing approaches treat identifiers such as `xmlDoc` using either frequency heuristics [25] or rigid rules [41], while ignoring code semantics. This leads to imprecise patterns that either retain irrelevant elements or miss critical ones. Notably, even developers struggle to identify such defects in isolation, as recognizing vulnerabilities often requires domain knowledge (e.g., understanding XML parser behavior) and contextual reasoning that goes beyond the immediate code structure. In such cases, examining how incorrect behavior is corrected can provide valuable semantic cues that are absent from the defective code alone. The contrast between the defective and fixed code constraints the space of plausible defect explanations. Reasoning over this constrained space allows developers to infer defect's root cause that accounts for the observed correction. Inspired by these modifications to the defect and their possible root causes, developers can also suspect similar defects when encountering analogous code patterns in the future, as the fix highlights the core characteristics that define the vulnerability. In practice, this means identifying how defect's root cause is reflected in specific syntactic and contextual patterns in code, such as the creation of a `SAXReader` object without the corresponding configuration that disables external entity processing. We refer to such code-level reflections of defect's root cause

as **defect-relevant elements**, which indicate these code elements that should be examined when recognizing analogous defects in previously unseen programs.

Given this observation, we investigate *whether LLMs can reason over bug-fix pairs to identify the defect-relevant elements and can be examined to recognize analogous defects in previously unseen code.* Prior studies have shown that LLMs can capture the semantic rationale of code modifications [39]. In our setting, we provide the defective and fixed code together with a structured prompt (§4.3.1) to an LLM (i.e., DeepSeek-v3), and examine whether it can relate inferred defect explanations to defect-relevant elements in code. For example, the model associated an inferred defect explanation with specific element sites, such as the call to `SAXReader().read()` without disabling DTD parsing, and the corresponding configuration change in the fix[1] (i.e., the option that disables DTD processing).

These responses indicate that the model can infer defect's root cause and point out the related elements to recognize analogous defects in previously unseen code. This positions LLMs as a semantic reasoning component for locating related elements of defects, which provides the foundation for subsequent pattern and DSL construction.

These defect-relevant elements are illustrated in our example: the unrestricted use of `SAXReader` in the defective version and the introduction of a corresponding `setFeature` configuration in the patch reflect how the same defect explanation is realized and addressed at the code level. Building on this insight, we developed DslGen, which leverages LLMs to reason over a defect-fix example, infer intent-guided constraints, and ground them into specific code elements, from which generalizable defect patterns can be constructed.

4 Approach

In this section, we present the details of our approach, DslGen, which addresses the challenge of deriving defect patterns from singular examples. The input to DslGen is a single code change pair, consisting of a pair $\langle b, f \rangle$ of the defective and fixed versions of a method and an optional natural-language description of the fix. In our evaluation, only the method pair is provided. The core idea is to employ large language models (LLMs) as *virtual domain experts* in the intent-aware abstraction stage, where they infer the fix intent and then synthesize defect's root cause from code changes and ground them into concrete code elements, thereby indicating which elements should be examined when recognizing analogous defects. By using the fix intent as a semantic cue to infer defect's root cause, DslGen constructs constraints based on the identified defect-relevant elements that align to the root cause, thereby capturing the underlying defect semantics and deriving more precise patterns. Finally, the extracted constraints are organized into a domain-specific language (DSL) program, which provides a compact and expressive representation of defect patterns.

Before applying DslGen, we first perform a lightweight preprocessing step using an LLM. Given a defect-fix example, the LLM determines whether it contains multiple defect-fix instances; if so, it isolates a single instance so that the subsequent pattern extraction focuses on one defect at a time.

Specifically, DslGen operates in three core phases: ❶ **Transfer Graph Construction** models each defect instance (i.e. bug-fix pair) as a unified graph by integrating the abstract syntax tree (AST) structure with fine-grained edit operations. ❷ **Intent-Aware Constraint Construction** leverages LLMs as *virtual domain experts* to reason over code changes, infer defect's root cause from the fix intent, and identify the code elements to be examined for pattern construction. ❸ **DSL-Based Pattern Generation** synthesizes DSL representations of defect patterns by consolidating the structural information captured in the transfer graph with the constraints inferred in the intent-aware constraint construction stage. The overview of our approach is presented in Fig. 2. Before introducing the details of each phase, we first introduce the underlying DSL, which provides the formal notation for pattern representation.

[1] https://github.com/jiachenhan/DSLGEN/09appendix/Infer Intent with LLM.jpeg

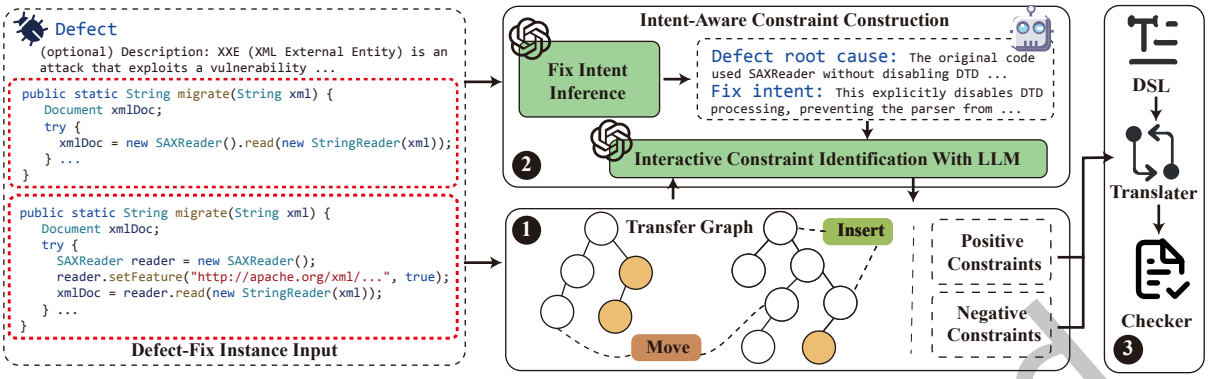


Fig. 2. Overview of DslGen.

4.1 Domain-Specific Language (DSL)

To ensure both readability for developers and executability within automated analysis, defect patterns in DslGen are represented using a domain-specific language (DSL). This design ensures that patterns are human-readable and amenable to refinement when needed, while DslGen automatically produces executable patterns without requiring human intervention. In DslGen, we adopt the DSL used in *CodeNavi* [2] as the default choice for three reasons. First, it is already deployed as a VS Code plugin, making it readily accessible to developers. Second, the plugin provides integrated detection capability by directly executing DSL patterns to identify potential defect instances. Third, its AST-like syntax is closely aligned with the structure of mainstream programming languages (e.g., Java and C++), which provides a natural bridge from program syntax to DSL-based pattern specification. In this section, we present the core grammar of the DSL, as illustrated in Fig. 3, to support the understanding of our approach. Specifically, a typical defect pattern – defined as a *Query* in the DSL – comprises two principal components: *Entity Declarations* and *Conditional Expressions*. Entity declarations specify the target AST node types and optionally assign aliases for scoped reference, maintaining direct correspondence with the underlying language's AST. Conditional expressions define essential constraints for capturing contextual features of defective code. They are typically written as compound conditions that specify required or prohibited code properties.

$$
\begin{aligned}
\text{Query} &::= \text{EntityDecl } \textit{where} \text{ Condition ;} \\
\text{EntityDecl} &::= \text{NodeType (Alias)}^{?} \\
\text{Condition} &::= \text{AtomCond} \mid \textit{not}\,(\text{Condition}) \\
&\quad \mid \textit{and}\,(\text{Condition}(,\ \text{Condition})^{+}) \\
&\quad \mid \textit{or}\,(\text{Condition}(,\ \text{Condition})^{+}) \\
\text{AtomCond} &::= \text{ValMatch} \mid \text{RelMatch} \\
\text{ValMatch} &::= \text{Attribute ValComp Value} \\
\text{RelMatch} &::= \text{Attribute NodeComp Query}
\end{aligned}
$$

$$
\begin{aligned}
\text{Attribute} &::= \text{Alias (. Property)}^{*} \\
\text{Value} &::= \text{NodeType} \mid \langle \text{var} \mid \text{liter name} \rangle \\
\text{NodeType} &::= \textit{functionCall} \mid \textit{ifBlock} \mid \cdots \\
\text{Property} &::= \textit{body} \mid \textit{parameter} \mid \cdots \\
\text{ValComp} &::= \textit{match} \mid \textit{is} \mid == \mid \neq \\
\text{NodeComp} &::= \textit{contain} \mid \textit{in} \\
\text{Alias} &::= \langle \text{any valid identifier} \rangle
\end{aligned}
$$

Fig. 3. Core grammar of the DSL used in DslGen.

[2]https://marketplace.visualstudio.com/items?itemName=HuaweiCloud.codenavi

Specifically, conditional expressions are of two kinds: *ValMatch*, which enforces attribute-level constraints (e.g., node identifiers or type requirements), and *RelMatch*, which specifies structural relationships such as parent–child containment. Both *ValMatch* and *RelMatch* rely on the notion of an *Attribute*, which models a reference to an object or its properties (e.g., `o.name` refers to the `name` property of object `o`). Together, *ValMatch*, *RelMatch*, and *Attribute* enable precise code pattern matching by combining local property checks with hierarchical structural context.

Consider again the vulnerability in Fig. 1a. Fig. 1b illustrates how this defect can be represented as a defect pattern in our DSL. The constraints specify that candidate programs must contain a `SAXReader` object instantiation (`oc_1`) within a method body (`fd_1`), while lacking any `setFeature` call with the specified argument (i.e., a string literal matching the given URL). The example demonstrates how the DSL expresses both the presence and absence of critical constructs in a concise and intuitive form. Notably, the DSL is interchangeable: an adapter can convert our intermediate representation into alternative DSLs (see Section 4.4).

4.2 Transfer Graph Construction

As discussed in the overall workflow, the goal of this phase is to construct a structural representation that consolidates all syntactic and edit information required for subsequent constraint construction. Since deriving pattern constraints directly from complex defect contexts without first organizing structural and edit information into a unified representation makes it difficult to consistently relate defect semantics to specific code elements and edits. To address this, DslGen introduces transfer graph that integrates code structure and edit operations into a single representation, providing a common structural basis for subsequent constraint construction. Specifically, DslGen converts the defective and fixed versions of source code into abstract syntax trees (ASTs), and then derives a unified graph representation from these ASTs. This graph serves as an explicit intermediate structure that localizes syntactic and edit-level information at the node level, enabling subsequent intent-aware constraint construction to focus on defect-relevant regions of the code. This strategy provides two main advantages. First, by explicitly organizing syntactic and edit-level information into a focused intermediate representation, it enables subsequent abstraction stages to reason over well-localized code regions rather than the entire program. Second, the AST-based graph structure naturally aligns with DSL constructs, enabling seamless translation into pattern representations. In the following, we formalize the definition of transfer graphs and describe their construction methodology.

We begin by formalizing abstract syntax trees (ASTs), which serve as the foundation for constructing transfer graphs.

Definition 4.1 (AST Node). An AST node is defined as a tuple of $\langle id, \mathcal{A}, l \rangle$, where id is a unique number for indexing, $\mathcal{A}$ is a set of attributes associated with the node (e.g., node types), and l is the location of the node in the source code.

Definition 4.2 (AST). An AST is defined as a tuple of $\langle \mathcal{N}, \mathcal{R}, r \rangle$, where $\mathcal{N}$ is a set of AST nodes, $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$ records the parent-child relation between two AST nodes in $\mathcal{N}$, while $r \in \mathcal{N}$ denotes the root node of the AST. Specifically, each $n_c \rightarrow n_p \in \mathcal{R}$ indicates node $n_c \in \mathcal{N}$ is the child node of $n_p \in \mathcal{N}$. Each node n_c has only one parent node n_p, while n_p may have multiple child nodes.

Building on the definitions of AST and AST node, we define code edit operations, which are essential since they implicitly capture the developer's intent behind the fix. Specifically, the defective and fixed versions of source code are transformed into AST representations – denoted as $T_b = \langle \mathcal{N}_b, \mathcal{R}_b, r_b \rangle$ and $T_f = \left\langle \mathcal{N}_f, \mathcal{R}_f, r_f \right\rangle$, respectively – by using a syntax parser. Then, DslGen applies an AST differencing tool to extract an edit script — a minimal sequence of node-level operations that transforms T_b into T_f. Specifically, each operation in the script is one of the following four types:

- **Delete**(n): Remove the subtree rooted at node $n \in \mathcal{N}_b$ from the defective AST T_b.
- **Update**(n, n', v_1, v_2): Change the attribute value of node $n \in \mathcal{N}_b$ from v_1 to v_2, yielding a new node $n' \in \mathcal{N}_f$.
- **Insert**(n_c, n_p, l): Insert a subtree rooted at node $n_c \in \mathcal{N}_f$ into the AST as a child node of $n_p \in \mathcal{N}_f$ at position l.
- **Move**(n_c, n_p, l): Move the subtree rooted at node $n_c \in \mathcal{N}_b$ to become a child node of $n_p \in \mathcal{N}_f$ at position l.

These atomic edit operations capture fine-grained changes between defective code and the fixed version, thereby reflecting essential aspects of developer repair strategies. As the foundational elements of transfer graph construction, each operation is represented as an operation node in the graph.

Definition 4.3 (Operation Node). An operation node is formally defined as a tuple of $\langle idx, ty, s, t, \mathcal{A}_o \rangle$, where idx indicates the execution order of the associated operation in the edit script, ty represents the operation type that is encoded as an `enum` with four possible values as defined above, s and t are the source and target AST nodes, and finally $\mathcal{A}_o$ is a set of operation-specific attributes, such as positions or values. For deletion operations, t is empty and denoted by $\perp$, while for insertion operations, s is empty and denoted by $\perp$.

Let T_b be the defective AST and T_f its fixed version, where T_f is derived by applying a sequence of operations $\mathcal{O}$. We represent this sequence with an operation node set $\mathcal{M}$ corresponding to $\mathcal{O}$. Based on these definitions, the transfer graph is formally defined as follows:

Definition 4.4 (Transfer Graph). A transfer graph from $T_b = \langle \mathcal{N}_b, \mathcal{R}_b, r_b \rangle$ to $T_f = \langle \mathcal{N}_f, \mathcal{R}_f, r_f \rangle$ is a directed acyclic graph (DAG) $G_{T_b \to T_f} = \langle \mathcal{N}, \mathcal{R}, \mathcal{M} \rangle$, where $\mathcal{N} = \mathcal{N}_b \cup \mathcal{N}_f$, $\mathcal{R} = \mathcal{R}_b \cup \mathcal{R}_f \cup \mathcal{R}_m$, and $\mathcal{R}_m$ is a set of edges capturing transformation relationships. For each operation node $o = \langle i, ty, s, t, \_ \rangle \in \mathcal{M}$, $s \to o \in \mathcal{R}_m \wedge o \to t \in \mathcal{R}_m$, and for each unchanged node $n_s \in \mathcal{N}_b$, $n_s \to n_t \in \mathcal{R}_m$, where $n_t \in \mathcal{N}_f$ is the corresponding node of n_s in the fix code.

This graph structure preserves the syntactic hierarchy of both code versions while retaining the fine-grained modification information. It provides a unified representation that serves as the foundation for contextual feature abstraction and pattern extraction. Fig. 4 presents an example transfer graph constructed from the motivating buggy and fixed code snippets shown in Fig. 1.

In this example, the transfer graph encodes the modifications between the defective and fixed versions through four representative edit operations: **Move**(n_5, n_{10^*}, 1), **Delete**(n_2), **Insert**(n_{1^*}, n_{10^*}, 1), and **Insert**(n_{1^*}, n_{13^*}, 2). As shown in Fig. 1a, defect-relevant nodes in the graph are annotated with superscript indices in the code snippet, and these indices correspond to the same nodes in the graph. To facilitate understanding of the graph, we describe several representative nodes as follows. Node n_1 represents `try` block, while node n_2 corresponds to a statement that parses an XML document using a newly created parser instance. Nodes n_5 and n_{5^*} denote the creation of a parser object before and after the fix, respectively. Node n_{13^*} represents a newly introduced method invocation `setFeature` that configures a parser feature in the fixed version.

4.3 Intent-Aware Constraint Construction

After the transfer graph is built, DslGen enters the stage of intent-aware constraint construction, which focuses on constructing the constraints required for subsequent DSL-based pattern formulation. In this stage, the LLM reasons over the fix context together with the transfer graph to infer the underlying defect's root cause, and highlights defect-relevant elements in the graph, including relevant nodes in the defective tree and edit operations associated with the repair, as the basis for constructing both positive and negative constraints.

As highlighted in our motivation, identifying which aspects of a fix reflect the underlying defect's root cause is often challenging and typically requires substantial domain knowledge of defect characteristics, analysis tools, and code context. To overcome this difficulty, DslGen introduces an intent-aware constraint construction

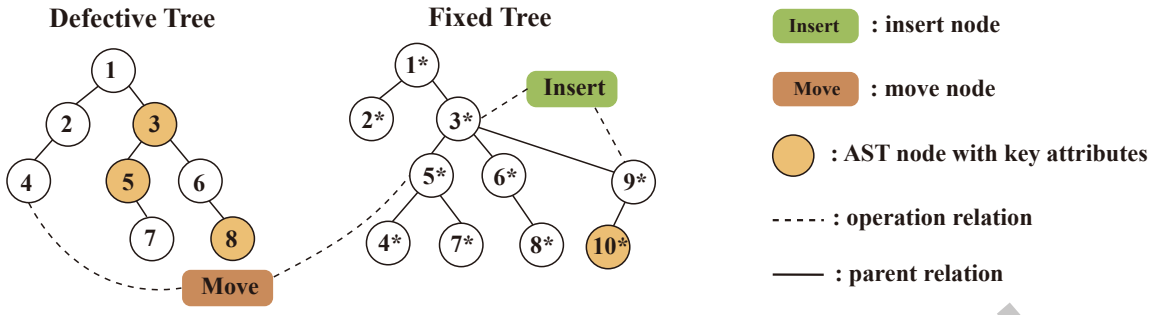


Fig. 4. A simplified example transfer graph constructed from the motivating defect-fix example in Fig. 1a. Nodes in the fixed tree are marked with a star (e.g., 1*), and identical numbering (e.g., 1 and 1*) indicates their correspondence.

component that draws on the reasoning capability of LLMs. In this paper, we define fix ***intent*** as a semantic explanation of *what errors* a given code change aims to avoid and how it alters the original program behavior, reflecting how developers reason about the purpose of the modification. Correspondingly, we define ***the root cause of a program defect*** refers to the underlying condition in the code, logic, or design that directly leads to the defect and whose correction prevents its recurrence. Based on this semantic understanding, DSLGEN further determines whether the change aims to fix an underlying defect and, if so, infers the corresponding defect's root cause reflected in the original code. Building on the inferred defect's root cause, DSLGEN identifies code elements that indicate the presence of the defect, while using edit operations to identify elements whose introduction serves to resolve the defect, together forming the final set of constraints. In summary, intent-aware constraint construction proceeds by first establishing a semantic understanding of the code change and inferring potential defect's root cause, and then forming both positive and negative constraints grounded in code elements and edit operations. We next describe these steps in detail.

4.3.1 Intent Extraction. As mentioned earlier, DSLGEN leverages LLMs to obtain a high-level semantic description(i.e, fix intent) of a code change, summarizing what the change attempts to achieve and how it alters the program behavior. Furthermore, we prompt the LLM to infer the defect's root cause from the fix intent. Building on this inferred defect's root cause, we further use LLMs to identify how the cause is reflected in the defective code, by describing the code constructs and usage patterns that give rise to the defect. For clarity, we present simplified versions of these prompts here, the complete prompt specifications are provided in our repository[3].

Specifically, Prompt 1 provides the LLM with the defective and fixed code snippets. It then instructs the model to (1) summarize the repairing intent — what the fix does and why it resolves the problem — and (2) identify the defect or violation that the fix addresses.

Prompt 1: Capturing Fix Intent and Defect Cause

You are given two code versions: one buggy, one fixed. Your task is to:

(1) Summarize what the fix is doing and why.
(2) Identify the violation or issue addressed by the fix.

Buggy code: [buggy_code]
Fixed code: [fixed_code]
(Optional) Error info: [error_description]

[3]https://github.com/jiachenhan/DSLGEN/script/app/abs/classified_topdown/prompts.py

By understanding why the fix resolves the problem, the model can infer what underlying condition was causing the defect, thus obtaining the root cause, as illustrated below (output may vary across runs):

Output of Prompt 1

The modification introduces an important security improvement to prevent XML External Entity (XXE) attacks. Here's why this change was made:
1. Security Vulnerability: The original code (before.java) used SAXReader to parse XML without any security configurations. This makes it vulnerable to XXE attacks, where an attacker could include malicious external entities in the XML.
2. The fix: ...

Building on the fix intent and defect's root cause inferred from the first step, we design a targeted prompt (i.e., Prompt 2) that asks the LLM to explain how the inferred defect's root cause is reflected in the defective code. The LLM is encouraged to move beyond free-form explanations by expressing defect-related constructs in rule-like form (e.g., "invoking a method on a newly created object") and anchoring them to concrete nodes and attributes. This step provides an intermediate alignment that links the semantic understanding of the defect's root cause to concrete program elements, effectively narrowing the scope for subsequent analysis.

Prompt 2: Align relevant contexts

Imagine you are generalizing the defect into a reusable pattern.
(1) Identify which expressions or syntax structures caused the violation.
(2) Describe them as rules (e.g., "a method invocation whose target is an object creation expression, such as `new String().getClass()`")
(3) Highlight *defect relevant elements* in the buggy version that match these rules.

The result of this step can be regarded as a semi-structured sketch of the defect. The sketch consists of defect-relevant concise rule-like statements paired with references to corresponding code locations. Situated between high-level natural-language intent descriptions and the final DSL constraints, they capture essential characteristics of the defect while preserving direct ties to the program's syntax and structure. For the example code repair shown in Fig. 1a, Prompt 2 produces a semi-structured description of the defect in the form of rule-like conditions (the excerpt of the complete output for clarity):

Output of Prompt 2

A potential violation is characterized when all of the following conditions hold:
- an object creation that instantiates an XML parser (e.g., `SAXReader`);
- a method invocation on that parser object that performs XML parsing;
- the absence of a preceding security configuration call on the same parser instance (e.g., `setFeature`).

These statements are not yet executable constraints; instead, they highlight the relevant code patterns that will later be formalized during constraint identification. The sketch is intended as intermediate cue within the LLM's output rather than directly executable constraints, as they may be incomplete or syntactically inconsistent; we further validate this limitation in our ablation study (Section 6.3).

4.3.2 Constraint Identification. Building upon the semantic cues identified during intent extraction, DslGen advances to the stage of constraint identification, where the statements in the sketch are translated into explicit constraints defined over the transfer graph of the given defect–fix pair. Concretely, each constraint is anchored to a specific node in the transfer graph: nodes from the defective tree define positive constraints that must be

present for a defect to be detected, while nodes from the fixed tree define negative constraints that should be absent in the defective version. Together, these positive and negative constraints form the key constraints, all of which must be satisfied for incoming code to be identified as containing the same defect. During this stage, candidate nodes are selected according to the transfer graph structure, and for each selected node, the system provides its source code snippet and relevant attributes (e.g., name) to the LLM, which then evaluates whether the element is relevant to the previously inferred root cause.

As formalized in Definition 4.4, a transfer graph $G_{T_b \to T_f} = \langle \mathcal{N}, \mathcal{R}, \mathcal{M} \rangle$ contains nodes from three distinct sources: defective code ($\mathcal{N}_b$), fixed code ($\mathcal{N}_f$), and edit operations ($\mathcal{M}$). From this structure, DslGen derives constraints along two dimensions. Constraints from defective code capture problematic contexts that give rise to the defect, while constraints from fixed code reflect the corrected contexts where the issue is resolved. Edit-operation nodes act as semantic bridges that link these two constraint categories and clarify their correspondence. For instance, the edit node **Insert**(n_{1^*}, n_{13^*}, 2) in Fig. 4 records that a `setFeature` invocation is inserted into the function body in the fixed version. This insertion defines the structural change that resolves the defect. The corresponding negative constraint, later expressed as `not(fd_1.body contain functionCall fc_2 ...)` in the DSL in Fig. 1b, is derived by identifying the absence of this construct in the defective version. In this way, the edit node connects the introduced structure in the fix to the missing structure in the defective code, thereby establishing the semantic linkage between positive and negative constraints. In what follows, we elaborate on the procedure of constraint identification in detail.

Constraints from defective code: Defective-side constraints are constructed by identifying code elements in the defective code that correspond to the defect's root cause characterized in the previous stage. To identify elements related to the inferred defect's root cause, DslGen examines candidate nodes encountered during traversal in a top-down manner over the AST and queries the LLM to determine whether each node is semantically relevant to the defect. In particular, the LLM takes the source code associated with the AST node as input. Taking the code shown in Fig. 1a and its associated AST shown in Fig. 4 as examples, when considering the AST node n_5, its associated source code "`new SAXReader()`" will be fed to the LLM, which includes its child node n_7 whose associated code is "SAXReader". In other words, when judging whether the current node is defect-relevant, all its child nodes (i.e., their corresponding code elements) can be seen by the LLM. As a result, if the node is determined as irrelevant, it means all the given code elements are irrelevant, including its child node. Consequently, we will not check its subsequent child nodes anymore for narrowing the scope of consideration and avoiding unnecessary LLM queries. On the contrary, if the node is considered relevant, the LLM will further evaluate which of its attributes (e.g, *value type* and *name*) are essential for characterizing the defect. Relevant nodes and attributes are tagged and retained as constraint candidates. In particular, if no attribute of the node is relevant, the node will be considered irrelevant. Nevertheless, the top-down evaluation process then will continue to check the relevance of its child nodes along the AST. To implement this traversal, we carefully design prompts that steer the LLM's output, constraining the interaction context to the current subtree to ensure consistent interpretation and reliable tagging of relevant nodes and attributes. Throughout the process, the previously extracted intent provides contextual knowledge that guides the LLM's decisions[4].

The traversal yields a set of tagged nodes and attributes in the defective AST T_b, forming a relevant subgraph of the complete transfer graph. These tagged elements constitute positive constraints in the resulting pattern, specifying the syntactic and semantic conditions that candidate defective code must satisfy. For example, in the transfer graph shown in Fig. 4, two nodes in the defective tree are tagged during traversal, namely n_5 and n_7. Here, node n_5 corresponds to an object creation expression, and node n_7 denotes the concrete instantiation of a `SAXReader`. These positive constraints capture the presence of faulty contexts in the defective code.

[4]https://github.com/jiachenhan/DSLGEN/script/app/abs/classified_topdown/prompts.py

Constraints from fixed code: Constraint identification on the fixed side follows a similar approach to that on the defective side, with the key difference lying in how the starting nodes are located. The relevant contexts in the fixed code are intrinsically linked to the operations that realize the defect repair. Because these fine-grained edit operations directly mark the syntactic constructs introduced or repositioned by the fix, they provide natural anchors for constraint extraction. DslGen therefore focuses on **Insert** and **Move** operations, as these introduce new syntactic elements into the fixed code. In contrast, **Delete** operations introduce no new contexts, and the relevant aspects of **Update** operations have already been examined on the defective side.

Because defect repairs often include extraneous edits unrelated to the core fix (e.g., refactoring-related insertions)[44], DslGen first interacts with the LLM to filter operations and retain only those genuinely relevant to resolving the defect. This filtering step leverages the previously extracted intent to distinguish core fixing actions from incidental ones. For each confirmed **Insert**(n_c, n_p, l) or **Move**(n_c, n_p, l) operation, DslGen designates the parent node $n_p \in \mathcal{N}_f$ as the starting point for a depth-first traversal of the fixed AST (T_f). During this traversal, DslGen applies the same LLM-based methodology and prompt templates as in the defective-side analysis, ensuring consistency in how positive and negative constraints are extracted.

The final output consists of tagged nodes and attributes within T_f, which collectively form a relevant subgraph of the complete transfer graph. Unlike the elements extracted from defective code, these tagged elements from the fixed code function as negative constraints in the pattern definition. These negative constraints originate from the code elements that the fix introduces or relocates in order to establish the correct context. In essence, such constraints specify the corrective conditions absent in defective code, thereby highlighting what must change for correctness to be restored. This perspective further helps characterize the broader semantic context of the defect, including the conditions that distinguish defective from correct behavior. Similarly, taking the transfer graph shown in Fig. 4 as an example, the colored nodes (n_{13^*} and n_{15^*}) in the fixed tree are tagged as it involves such negative constraints.

4.4 DSL-Based Pattern Generation

To provide a compact and executable representation of defect patterns, we encode the extracted constraints into a domain-specific language (DSL). Building on the constraints constructed in previous phases in transfer graph, we describe the process of generating DSL programs for defect patterns. This generation is a fully rule-based translation from the constraint-annotated transfer graph into executable DSL code, mechanically converting graph elements into DSL constructs while preserving their semantic relationships.

DSL generation translates the constraint-annotated transfer graph into an executable DSL program by converting tagged nodes into declarative rules. The translation distinguishes between two categories of constraints: positive constraints derived from defective-side nodes $\mathcal{N}_b \subseteq T_b$, which describe contexts that must be present in defective code, and negative constraints derived from fixed-side nodes $\mathcal{N}_f \subseteq T_f$, which describe corrective contexts that must be absent. In the following, we explain how each category is mapped into DSL constructs.

For positive constraints (from T_b): For positive constraints, DslGen first locates the nearest common ancestor n_s of all tagged nodes in $\mathcal{N}_b$ and initializes the DSL program with a *Query* whose *EntityDecl* declares the type of n_s. From this starting point, DslGen recursively expands the subtree to generate *Condition* rules. Tagged attributes are translated into *AtomCond*, with equality or value checks expressed through *ValMatch*; when multiple attributes apply to the same node, they are combined using the *and* operator. Structural requirements are captured by *RelMatch*: When the pattern requires that a node contain a designated element in its subtree, DslGen emits a *RelMatch* with the *contain* relation to assert that requirement, and the recursive expansion continues from the contained element. Through this recursive process, the resulting DSL conditions naturally mirror the hierarchical structure of the AST, essentially forming a structural pattern that matches subgraphs of the transfer graph corresponding to defective contexts.

For negative constraints (from T_f): For negative constraints, DslGen follows the same translation scheme as for positive ones, but starts from the parent node $n_p \in \mathcal{N}_f$ of each relevant **Insert**(n_c, n_p, l) or **Move**(n_c, n_p, l) operation. The resulting conditions are wrapped under a *not* operator, forming a negated condition. The parent node n_p indicates where the inserted or moved code appears in the fixed tree. This determines where the negated condition should be placed in the DSL. Since nodes in the fixed and defective trees are matched in the transfer graph, the same parent node can be located when constructing the DSL. The negated condition is then placed under that parent node in the DSL. If this node is not explicitly represented, the condition is attached to the nearest ancestor that is present. In the motivating example, the function call `setFeature` is inserted under a function declaration in the fixed tree. The parent node of this insertion determines where the negated condition should appear in the DSL. When translated, the condition is placed under *functionDeclaration fd_1*, resulting in a condition of the form *not(fd_1.body contain functionCall ...)*.

As illustrated in Fig. 1b, lines 3-4 demonstrate positive constraints derived from the defective code, requiring candidate code to satisfy these conditions, while lines 5-15 present negative constraints extracted from the fixed code, specifying conditions that must be violated to match the defect pattern. Expressed in DSL form, these rules not only provide a compact and executable representation of the defect pattern but also remain human-readable, allowing developers to understand the underlying constraints and refine them when necessary.

5 Experimental Setup

5.1 Research Questions

- **RQ1**: How effective is DslGen in identifying defect patterns across code change pairs?
- **RQ2**: How effective is DslGen in predicting real-world defects in practical usage scenarios?
- **RQ3**: How do each component and LLM configurations affect the effectiveness of DslGen?
- **RQ4**: Can DslGen assist developers by producing interpretable and generalizable defect queries?

5.2 Datasets

Effective evaluation requires a dataset that organizes similar real-world defects into clusters. Although widely-used bug benchmarks such as Defects4J [27] provide high-quality bug-fix pairs and have been adopted in pattern mining and automatic program repair [25], they are organized around individual bugs from specific projects without grouping recurrent defects of the same category, and thus do not support the within-cluster generalization evaluation required by our task. Other resources like C3 [31], though widely referenced [25], were designed for systematic editing rather than defect detection. We therefore include C3 as a comparative reference, and construct a new dataset, Defs, by mining bug-fix commits from popular GitHub repositories [4] and clustering similar defects associated to the same defect patterns. Specifically, each defect repair commit forms a pair consisting of the defective and fixed codes, and pairs exhibiting similar defect patterns are grouped into clusters. This structure serves two purposes: (1) evaluating whether patterns learned from one pair can generalize to other pairs within the same cluster, and (2) examining whether patterns can precisely detect the potential defect in the defective version while avoiding false positives on the fixed version.

5.2.1 Defs Dataset. We started by mining defect repair commits from GitHub repositories, focusing on 102 Java projects with more than 5,000 stars as a data quality criterion to ensure both popularity and active maintenance. Using the keywords "fix" and "repair," we collected 95,335 candidate commits dated between January 1, 2023 and January 1, 2025. Since commit messages alone are often unreliable, we refined this set using two static analysis tools, CodeQL [1] (138 defect-related queries) and PMD [5] (147 rules), solely as data collection instruments to retain commits that corresponded to genuine defect fixes. From these verified commits, we extracted method-level code changes associated with the same checker, reflecting the intra-procedural focus of our approach, and applied

a hierarchical clustering algorithm [31] to group similar fixes into clusters. In our dataset, a category refers to a checker rule of the underlying tool, while a cluster groups defect instances within a category that share the same repair shape; a single category may therefore yield multiple clusters when real-world fixes take structurally distinct forms.

To ensure dataset quality, the first two authors independently annotated all clusters after aligning on annotation criteria, achieving substantial agreement (Cohen's kappa ≈ 0.8)[14]. Discrepancies were resolved through discussion to finalize the dataset. To align with the intra-procedural scope of our approach, we excluded commits involving multiple methods. The resulting dataset is divided into two subsets, Defs-QL and Defs-PMD, according to the checker used during data collection to identify candidate defect fixes.

Overall, Defs consists of 1,287 defect–fix pairs organized into 319 clusters and spanning 63 distinct defect categories, ranging from *null-pointer dereferences* to *resource leaks*. The defect categories are provided in our repository[5]. Each cluster represents a certain defect pattern and contains multiple instances, providing semantically comparable defect–fix pairs required for evaluating pattern generalization. The fixes themselves cover diverse modification scales, from concise one-line changes to larger edits exceeding 20 lines, ensuring both coverage of common small fixes and representation of more complex repairs. With its combination of scale, diversity, and validated quality, the dataset serves as a representative resource for studying defect pattern extraction in realistic settings. It is publicly available in our project repository, with additional statistics reported in the appendix A.2.

5.2.2 C3 Dataset. C3 [31] contains systematic code edits that are not limited to defect fixes, allowing us to evaluate our approach on a dataset broader than defect-specific collections. The dataset includes over 200,000 clusters of similar edits, organized as groups of related code changes. For our evaluation, we randomly sampled 1,000 clusters and, from each cluster, selected two pairs of code changes: one used to extract a pattern and the other reserved for testing, striking a balance between experimental feasibility and statistical reliability.

5.3 Baselines

To address RQ1, since most existing defect pattern extraction or defect-detection methods depend on large corpora to identify frequent transformations and are therefore inapplicable to the single-example setting of our task, we selected baselines from automated program repair (APR) pattern-mining techniques. Specifically, GenPat [25] and FixMiner [30] are the only two techniques that can be reasonably adapted, as both operate on structural edit patterns over ASTs and can be repurposed to identify similar defects from individual defect-fix examples, i.e., they can extract a defect pattern from one single example, aligning our approach.

To address RQ2, we evaluate our approach in a practical pre-merge setting, where defect-inducing commits must be examined before merging in the Continuous Integration (CI) pipeline (see Section 6.2). In this setting, we compare DslGen with both baselines in RQ1 and additional straightforward LLM-based baseline designed in our study, since LLM-based methods have recently demonstrated strong performance in defect detection [33, 37, 57]. Specifically, we include two state-of-the-art LLMs, including GPT-4o [6] and DeepSeek-v3 [2], two widely-used LLMs as the representatives.

5.4 Evaluation Metrics

We employed four standard metrics that are widely used in defect detection and classification tasks, including **Accuracy**, **Precision**, **Recall**, and **F1-score**.

Additionally, we also employed **validity rate** (the percentage of syntactically valid DSL programs) to evaluate the contribution of each component in our approach. This metric is unique to our method as baselines don't generate DSL programs.

[5]https://github.com/jiachenhan/DSLGEN/09appendix/Final Checkers.pdf.

Table 1. Defect detection across code change pairs (RQ1)

Dataset	Tech.	Accuracy	Precision	Recall	F1-score
Defs-QL	DslGen	**0.780**	**0.811**	**0.729**	**0.768**
	GenPat	0.576	0.567	0.644	0.603
	FixMiner	0.508	0.667	0.034	0.065
Defs-PMD	DslGen	**0.800**	**0.810**	**0.785**	**0.797**
	GenPat	0.613	0.659	0.469	0.548
	FixMiner	0.506	0.556	0.058	0.105
C3	DslGen	**0.579**	**0.603**	**0.460**	**0.522**
	GenPat	0.525	0.530	0.441	0.481
	FixMiner	0.487	0.438	0.091	0.151

5.5 Implementation and Configuration

We implemented DslGen in Java with over 25k lines of code. We utilized Eclipse JDT (Java Development Tools) [3] v3.30.0 to parse Java source code and employed GumTreeDiff [18, 19, 40] v4.0.0-beta2 to compute code differences. By default, DslGen employed DeepSeek-v3 [2] via API calls to perform fix intent extraction and essential attribute constraint identification as mentioned in Section 4.3, with temperature and top-p both set to their default value of 1.0.

Environment. Our experiments were conducted on a local machine equipped with dual Intel Xeon 6388 CPUs, 512GB RAM, and four A800 GPUs, running Ubuntu 20.04.6LTS.

6 Result Analysis

6.1 RQ1: Defect Detection Across Code Change Pairs

To evaluate DslGen's ability to extract generalizable defect patterns and recognize similar defects, we formalize each code change example within a cluster as $c_i = \langle b_i, f_i \rangle$, where b_i and f_i denote the defective and fixed methods, respectively. For each cluster, we randomly sample two examples, c_1 and c_2, to form a pair (c_1, c_2). Specifically, for RQ1, DslGen extracts a defect pattern from c_1 and tests whether it can detect the same defect in c_2. A true positive (TP) occurs when the pattern matches b_2, while a false positive (FP) occurs when it matches f_2 instead. Symmetrically, a false negative (FN) occurs when the pattern does not match b_2, and a true negative (TN) when it does not match f_2. The pattern is applied to both b_2 and f_2, so each pair contributes one positive and one negative location, and detections on the fixed version are explicitly measured as errors.

As shown in Table 1, DslGen consistently outperforms GenPat and FixMiner across all three datasets. On both Defs-QL and Defs-PMD, DslGen achieved the highest F1-scores – 0.768 and 0.797, respectively—significantly higher than GenPat (0.603 and 0.548) and FixMiner (0.065 and 0.105). These results highlight DslGen's balanced precision and recall, demonstrating both accuracy and broad coverage of diverse defect patterns.

In addition, DslGen maintains a precision above 80% on both high-quality datasets (Defs-QL and Defs-PMD), indicating that it effectively captures the key constraints underlying defects. GenPat achieves moderate precision (56.7% on Defs-QL and 65.9% on Defs-PMD), but suffers from frequent false positives because its constraints are derived from corpus-level statistics, which retain only frequent tokens without considering the context of certain fixes and thus fail to capture the essential defect characteristics. FixMiner achieves the lowest false positive rates (1.7% and 4.6%), but at the cost of extremely low recall, missing more than 90% of actual defects. This trade-off reflects FixMiner's rigid matching strategy, which severely limits its ability to generalize across variants of similar

defects. In contrast, DslGen attains superior recall, reaching 72.9% on Defs-QL and 78.5% on Defs-PMD, far surpassing both GenPat and FixMiner. This improvement arises from DslGen's ability to leverage defect repair intent to uncover the defect's root cause and distill their essential characteristics, enabling the construction of patterns that are both precise and generalizable. We also note that our instance separation strategy contributes to part of the improvement on Defs-PMD: 24 examples contain multiple defect-fix instances(all from Defs-PMD), and the LLM successfully separates them in 21 cases. This partly explains why the improvement over GenPat is more pronounced on Defs-PMD (78.5% vs. 46.9%) than on Defs-QL (72.9% vs. 64.4%).

Regarding execution efficiency, we measured the runtime of the generated DSL programs at the method level. Among the 319 DSL programs produced in our evaluation, 318 completed within 1 second when applied to a method pair. Only one valid program failed to finish within one minute due to excessive nesting levels, which caused performance degradation in the matching engine. This indicates that the generated patterns can be executed efficiently in practice. Overall, these results demonstrate that DslGen provides a reliable approach for extracting and applying defect patterns across diverse datasets.

We also evaluated DslGen on the C3 dataset to assess its generality on benchmarks not specifically designed for defect detection. As shown in Table 1, DslGen still outperforms GenPat and FixMiner across all datasets, demonstrating the robustness of our approach. Nevertheless, its performance on C3 noticeably dropped, because many edits in this dataset are defect-irrelevant and lack a clear fixing intent (e.g., refactorings such as renaming), to guide the precise pattern constraint identification, In contrast, the baselines depend on predefined rules rather than such intent and are thus less sensitive to such issues.

Moreover, the C3 dataset itself is of limited quality, as it was fully automatically collected without manual inspection. Many included changes do not correspond to real defects or lack a consistent fixing purpose within clusters. This not only reduces the performance of all methods on C3 but also makes it an unreliable benchmark for evaluating defect pattern extraction. The observation also highlights the necessity of our curated datasets tailored for defect pattern extraction and defect detection, where high-quality and manually confirmed similar defect repair examples enable more reliable evaluation. To further substantiate this observation, we manually analyzed 100 cases from C3 – 50 false positives and 50 false negatives reported by DslGen. Among the false negatives, all arose from semantic inconsistency within clusters: although grouped together, the code modifications reflected disparate editing purposes. For the false positives, 43 out of 50 were benign refactorings with little semantic impact (e.g., prefixing variable names with "m_" to indicate member variables), while the remaining 7 involved changes unsupported by our DSL (e.g., updating method signatures like adding the `final` modifier to function parameters), preventing precise constraint generation. Therefore, in the subsequent experiments, we only employing the high-quality datasets Defs-QL and Defs-PMD while discard the C3 dataset to ensure the reliability of the results.

6.2 RQ2: Defect Detection in Practice

As aforementioned, we evaluate DslGen in a practical pre-merge setting: before a change is integrated into the project's codebase, a checker examines whether the change may introduce defects. Formally, assuming $c1 = \langle b_1, f_1 \rangle$ and $c2 = \langle b_2, f_2 \rangle$ represent two similar code change examples, forming a pair (c_1, c_2). Specifically, b_1/b_2 represent defective methods, while f_1/f_2 are the corresponding fixed methods. Then, we first extracted the bug-inducing commit of b_2, and then apply the extracted pattern from c_1 to detect candidate defects from the changed files involved in that bug-inducing commit. This is a controlled experiment as we know this commit contains the exact defect of b_2. In particular, each bug-inducing commit involves code changes on an average of 96.73 methods in our dataset, which is the scope of the code for defect detection. Among these methods, usually several of them are truly defective ones and should be detected, while the remaining large majority of methods are deemed defect-free and thus should not be detected. Specifically, those defective methods were decided by leveraging the corresponding static checker. A true positive (TP) is a defective method matched by the pattern,

Table 2. Effectiveness of DslGen in practice (RQ2)

Dataset	Tech.	Accuracy	Precision	Recall	F1-score
Defs-QL	DslGen	**0.969**	0.481	**0.865**	**0.618**
	GenPat	0.936	0.270	0.689	0.387
	FixMiner	0.854	0.556	0.027	0.051
	Def-Pred$_{4o}$	0.521	0.654	0.337	0.445
	Def-Pred$_{v3}$	0.494	**0.608**	0.171	0.267
Defs-PMD	DslGen	**0.921**	0.595	**0.745**	**0.665**
	GenPat	0.872	0.255	0.342	0.292
	FixMiner	0.620	0.632	0.019	0.037
	Def-Pred$_{4o}$	0.412	0.814	0.305	0.444
	Def-Pred$_{v3}$	0.313	**0.865**	0.135	0.234

a false negative (FN) is a defective method not matched, a false positive (FP) is a defect-free modified method matched, and a true negative (TN) is a defect-free modified method not matched. The commit scope corresponds to a realistic pre-merge inspection scenario, where the pattern is applied to the methods touched in a submission to flag defects before the change is merged.

Table 2 summarizes the experimental results of defect detection across all approaches. On both Defs-QL and Defs-PMD, DslGen achieved the highest F1-scores (0.618 and 0.665), demonstrating superior overall detection performance. Compared to GenPat and FixMiner, DslGen maintains reasonable precision while achieving substantially better recall. As discussed in RQ1, FixMiner's rigid constraints lead to low recall, while GenPat's loose abstraction reduces precision. In addition, we also designed two LLM-based defect detection approaches (Def-Pred$_{4o}$ and Def-Pred$_{v3}$ by employing GPT-4o and DeepSeek-v3, respectively) for comparison in this scenario, where the LLMs take a candidate method as input and output whether it contains a defect. As a consequence, these models achieved a high precision (up to 86.5% on Defs-PMD when adopting DeepSeek-v3) but very limited recall (as low as 13.5%). This indicates that while LLMs can identify some defective methods with high confidence, they lack the coverage necessary for robust pre-merge defect detection.

In summary, the experimental results demonstrate that DslGen can effectively detect potential defects in the real pre-merge defect detection scenario, highlighting its significance and value for practical use. Moreover, as it will be explained in Section 6.4, the defect patterns generated by our approach have good readability and support manual refinement, which can further improve its effectiveness, while the baselines cannot.

6.3 RQ3: Component Ablation and LLM Configurations

We conducted an ablation study to assess the contribution of different components and design choices in DslGen. Table 3 summarizes the results across four configurations. (1) ***LLM-Gen*** relies solely on the LLM to generate defect patterns directly from code change examples under the grammar specification, without leveraging our transfer graph; this configuration highlights the risks of direct LLM generation, which are substantiated by the results presented below. (2) ***LLM-Sketch*** derives DSL constraints directly from the LLM-generated sketch. Specifically, we prompt the LLM to decompose the sketch into defect-relevant AST nodes and attributes, which are then mapped to the transfer graph via literal-value matching, bypassing the node-by-node traversal. (3) ***TG-P*** incorporates only the positive constraints derived from the defective code. (4) ***TG-PN*** incorporates both positive and negative constraints from the defect-fix example, and is the default setting of DslGen.

Table 3. Ablation study results of DslGen (TG-PN is the default configuration)

Dataset	Tech.	Validity	Accuracy	Precision	Recall	F1-score
Defs-QL	LLM-Gen	27/59	0.559	**1.000**	0.119	0.213
	LLM-Sketch	59/59	0.627	0.800	0.339	0.476
	TG-P	59/59	0.720	0.710	**0.746**	0.727
	TG-PN	59/59	**0.780**	0.811	0.729	**0.768**
Defs-PMD	LLM-Gen	63/260	0.563	**0.946**	0.135	0.255
	LLM-Sketch	260/260	0.706	0.854	0.496	0.628
	TG-P	260/260	0.762	0.721	**0.854**	0.782
	TG-PN	260/260	**0.800**	0.810	0.785	**0.797**

Table 4. Effectiveness with different LLMs (RQ3)

Dataset	Tech.	Accuracy	Precision	Recall	F1-score
Defs-QL	DslGen w/ v3	**0.780**	0.811	**0.729**	**0.768**
	DslGen w/ 4o	0.754	**0.895**	0.576	0.701
	DslGen w/ Qwen	0.746	0.754	**0.729**	0.741
Defs-PMD	DslGen w/ v3	**0.800**	0.810	**0.785**	**0.797**
	DslGen w/ 4o	**0.800**	**0.868**	0.708	0.780
	DslGen w/ Qwen	0.787	0.809	0.750	0.778

As shown in Table 3, in the LLM-Gen setting the LLM produced only a small fraction of syntactically valid defect patterns (27/59 cases in Defs-QL and 63/260 cases in Defs-PMD). In contrast, incorporating the transfer graph ensures syntactic validity across cases. While iterative prompting or post-processing might alleviate syntax errors, the deeper challenge lies in ensuring that the extracted constraints are semantically consistent and contextually complete. This underscores the need for a structured pipeline—rather than relying solely on direct generation—to construct defect patterns that are both valid and generalizable. For the LLM-Sketch configuration, although syntactic validity is preserved, recall drops substantially across both datasets. This is because literal-value matching suffers from AST misalignment, and direct decomposition tends to retain all mentioned concrete values (e.g., specific names) as constraints, making the resulting patterns overly restrictive. In contrast, our node-by-node traversal enables fine-grained decisions on which attributes are truly essential. When using TG-P, precision and accuracy decrease while recall increases compared with TG-PN, since negative constraints are absent to restrict the scope of the patterns. This illustrates the inherent trade-off between precision and recall. With both positive and negative constraints TG-PN, our approach attains the best overall performance, achieving the highest F1 scores across the two datasets.

Finally, we evaluated the robustness of our approach across different LLM backends by comparing its performance when adopting DeepSeek-v3 (the default configuration), GPT-4o, and Qwen-Max, respectively. The results are presented in Table 4. Across the datasets, the findings are consistent: GPT-4o delivers higher precision through more conservative interpretations, but this comes at the expense of lower recall, while Qwen-Max achieves a more balanced trade-off between precision and recall. However, because the abstraction process depends on the model's semantic understanding, weaker LLMs may cause degradation not only in capturing the defect's intent but also in accurately identifying its essential characteristics. In addition, model differences involve more than

the number of constraints produced; they also concern how constraints are selected, suggesting that combining multiple LLMs could further improve constraint quality. Importantly, our approach consistently outperforms all baselines regardless of the LLM backend, indicating that DslGen does not depend on a particular model and thus remains flexible and adaptable in practice.

6.4 RQ4: User Study – Quality of DslGen

While DslGen is designed as an automatic and standalone defect pattern extraction technique from a single defect repair example, the generated patterns are not guaranteed to be fully precise. These patterns are represented as DSL programs in our system , and developers may need to interpret and refine them further in real applications (This requirement is confirmed by our industrial company). To evaluate the quality of the patterns produced by our approach from a practitioner perspective, we further conducted a user study. [6]

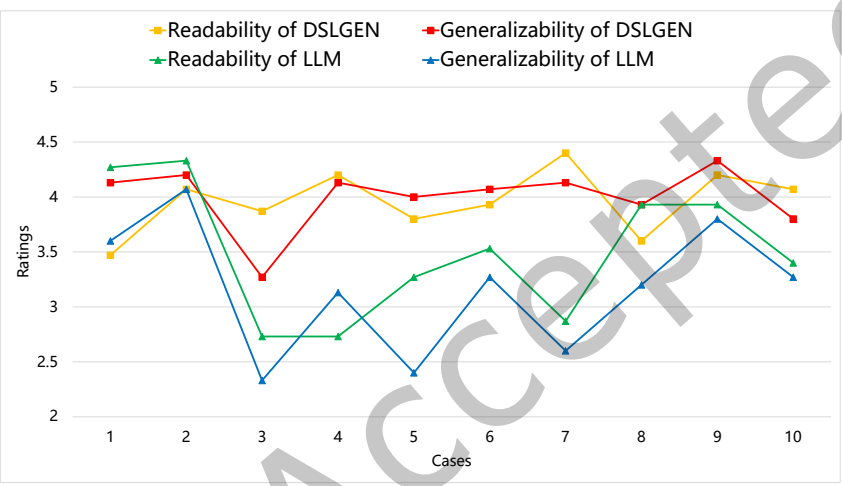


Fig. 5. Ratings of defect patterns generated by DslGen and LLMs

Participants. We recruited 15 professional developers from an international technology company. All were familiar with the DSL adopted in our approach. The group comprised 5 DSL engine developers, 6 DSL rule authors, and 4 end users. In terms of DSL usage frequency, 6 reported daily use, 3 weekly, and 6 monthly. Their industry experience also varied: 3 had 0–2 years, 7 had 3–5 years, 3 had 6–10 years, and 2 had more than 10 years. This diversity provided a balanced basis for evaluating both early-career developers' adaptability and experienced professionals' expectations for generated DSL usability.

Study Design. We randomly selected 10 example cases, each accompanied by defect patterns generated by both DslGen and DeepSeek-v3, which is the same LLM used by DslGen in this study. To avoid potential bias, we anonymized the origins of the defect patterns, providing them without annotations or disclosure of the underlying generation techniques. Participants were instructed to evaluate each DSL program on **Readability** and **Generalizability** using a 5-point Likert scale [35]. We further collected feedback on the overall quality of the defect patterns and participants' preferred ways of using DslGen in their workflows. The full dataset is publicly available on our project website.

Fig. 5 presents the participants' ratings on the 10 examples. DslGen-generated defect patterns received higher scores in 7 out of 10 cases, with an average rating of 3.96 compared to 3.50 for LLM-generated defect patterns.

[6] https://github.com/jiachenhan/DSLGEN/09appendix/Survey on the Quality of Automatically Generated DSLs.pdf

In terms of generalizability, DslGen's defect patterns were rated higher as well, with an average score of 4.00 versus 3.17.

Beyond quantitative results, we further investigated how developers perceive and envision using automatically generated defect patterns in practice. All 15 participants identified at least one adoption scenario: 14 would use them for rapid prototyping, and 12 expressed willingness to deploy them in production environments. Regarding workflows, most participants (12 of 15) explicitly favored a semi-automated process—starting from a valid auto-generated DSL and refining it as needed—while only 3 preferred a fully automated approach, and none opted for manual authoring. These findings underscore the practical value of DslGen, which guarantees syntactic correctness and produces interpretable defect patterns that developers can readily refine, making it naturally aligned with real-world, developer-in-the-loop workflows.

In addition to specific example-based evaluations, we also gathered participants' general feedback on the quality of generated defect patterns. Most participants found DslGen's outputs to be readable and requiring only minor adjustments, while LLM-generated defect patterns were perceived as more error-prone and harder to edit. Specifically, 14 out of 15 participants rated DslGen's defect patterns as usable with minimal changes, whereas only 9 participants gave the same rating to LLM-generated counterparts. Moreover, several developers raised concerns about the verbosity and inconsistency of LLM outputs, with one participant explicitly describing them as unreliable. These responses further support the practical advantages of DslGen in real-world development workflows.

7 Discussion

7.1 Limitation

This work shares two common limitations with existing pattern mining works [10, 25, 30, 36, 41, 42, 49]. First, DslGen is only capable of identifying defect patterns that have already appeared, and may struggle with previously unseen defect types. Second, our approach currently focuses on intra-procedural code structures, without explicitly modeling data-flow dependencies. As a result, some defect patterns that rely on data-flow reasoning may not be fully captured in all cases. Although the LLM may still implicitly infer such relations during its reasoning process, they will not be explicitly included in our DSL representation, which may affect the generalizability of the extracted patterns in certain scenarios.

7.2 Threats to Validity

Internal. Since the Defs dataset used for validating DslGen is collected through running static analysis tools, which may introduce false positives. Such noise in the dataset could potentially affect the performance of our evaluation results. To mitigate this issue, the first two authors independently conducted a careful manual annotation of all samples. We also released the dataset to support reproducibility.

External. This threat primarily comes from the evaluation datasets we used, and the performance of DslGen may not be generalized to other datasets. To mitigate this threat, we collected data from nearly 100 diverse open-source repositories to ensure the representativeness and practicality of the dataset. On the other hand, our dataset is derived from defects reported by static analysis tools, meaning the defects are already known and statically detectable. The effectiveness of our method on defects beyond those detected by static analysis tools used in our work(i.e., CodeQL and PMD) remains unevaluated. Furthermore, each defect corresponds to a single detection rule from CodeQL or PMD. The performance of our approach when working with more complicated defects that potentially require the combination of multiple detection rules is still unexplored. To mitigate this threat, our dataset covers 63 distinct defect categories involving non-trivial control-flow and data-flow reasoning, ranging from null-pointer dereferences to resource leaks. However, to systematically evaluate the effectiveness of our approach in a broader range of applications is still underexplored. Since this process needs to construct

a more comprehensive dataset – including defects reported by diverse static analysis tools or even dynamic detection methods, we leave it as our future work.

7.3 Cost and Practicality of LLM-Driven Interaction

One practical concern of our approach lies in its reliance on LLMs during the Constraint Construction phase, which may incur non-negligible computational and financial cost. Therefore, we measured the actual LLM usage during our experiments and found that **the cost of generating a single pattern is relatively low — approximately $0.006 on average**, which we consider minimal and well within an acceptable range for practical use.

7.4 Future Work

We outline three directions for future work that extend the current framework.

Inter-procedure defect pattern extraction: Our current implementation detects defects within a single method, focusing solely on intra-procedural code changes. However, many defects—such as memory leaks—involve complex, inter-procedural dependencies. To broaden the applicability of our approach, extending the transfer graph to capture inter-procedural contexts (e.g., caller–callee relationships and cross-method data flow) would enable the framework to model defect patterns that span multiple methods.

Online defect pattern refinement: As described in the paper, our method currently extracts defect patterns from individual examples. This process may yield imperfect patterns, leading to false positives that increase manual verification effort. To enhance pattern quality, detection feedback can be leveraged to iteratively refine extracted patterns. False positives observed during detection often indicate overly general or missing constraints. Incorporating such feedback into the pattern construction process would support an iterative refinement loop, thereby improving precision.

Effective code change isolation: In practice, defect repairs and refactorings are frequently tangled, complicating the extraction of high-quality defect patterns. Similarly, multiple independent defect-fix instances may be interwoven. Our current implementation employs LLM-assisted analysis to disentangle such instances prior to pattern abstraction. While this strategy is effective in most cases, more robust instance-disentanglement techniques would help ensure that unrelated changes do not interfere with one another during pattern construction.

8 Conclusion

In this paper, we presented DslGen, a novel approach that automatically extracts defect patterns by modeling the semantic intent behind the developer's fix. By leveraging large language models to infer fix intent, our approach extracts a series of defect-related constraints and translates them into a DSL representation, enabling the construction of high-quality and generalizable patterns. We comprehensively evaluated DslGen across two datasets. On curated datasets, DslGen achieves a precision up to 81% and a recall over 73%, consistently outperforming the strongest baseline across all metrics. In a simulated real development pipeline, DslGen achieves the best overall effectiveness among all baselines, with substantially higher recall and competitive precision, demonstrating its practical utility for detecting defects before code integration. Finally, a user study involving 15 experienced developers from industry further confirmed the value of our approach for practical use.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant Nos. 62572344, 92582202, 62322208, 62232001), the Beijing-Tianjin-Hebei Natural Science Foundation Cooperation Project (Grant No. 25JJJJC0030), and the CCF-Huawei Populus Grove Fund.

References

[1] 2024. CodeQL. https://codeql.github.com/. Accessed: 2024-11-20.
[2] 2024. DeepSeek-V3. https://www.deepseek.com/. Accessed: 2024-03-20.
[3] 2024. Eclipse JDT. https://projects.eclipse.org/projects/eclipse.jdt. Accessed: 2024-11-20.
[4] 2024. GitHub. https://github.com/. Accessed: 2024-11-20.
[5] 2024. PMD. https://pmd.github.io/. Accessed: 2024-11-20.
[6] 2025. GPT-4o. https://openai.com/index/gpt-4o-system-card/. Accessed: 2025-03-20.
[7] 2025. semgrep. https://semgrep.dev/. Accessed: 2025-5-29.
[8] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. *Unified pre-training for program understanding and generation.*
[9] Elena N Akimova, Alexander Yu Bersenev, Artem A Deikov, Konstantin S Kobylkin, Anton V Konygin, Ilya P Mezentsev, and Vladimir E Misilov. 2021. A survey on software defect prediction using deep learning. *Mathematics* 9, 11 (2021), 1180.
[10] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.
[11] Karima Berramla, El Abbassia Deba, Jiechen Wu, Houari A. Sahraoui, and Abou El Hassan Benyamina. 2020. Model Transformation by Example with Statistical Machine Translation. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic (Eds.). SCITEPRESS, 76–83. doi:10.5220/0009168200760083
[12] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. 2007. Finding what's not there: a new approach to revealing neglected conditions in software. (2007), 163–173.
[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. *Evaluating large language models trained on code.*
[14] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
[15] CVE. 2021. CVE-2021-21250: Post-Auth External Entity Expansion (XXE). https://www.cve.org/CVERecord?id=CVE-2021-21250. Accessed: 2025-05-29.
[16] CVE. 2021. CVE-2021-44228: Apache Log4j2 JNDI. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228. Accessed: 2025-05-29.
[17] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.
[18] Jean-Rémy Falleri and Matias Martinez. 2024. Fine-grained, accurate and scalable source differencing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 231:1–231:12. doi:10.1145/3597503.3639148
[19] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014.* 313–324. doi:10.1145/2642937.2642982
[20] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.
[21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.
[22] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *Acm sigplan notices* 39, 12 (2004), 92–106.
[23] Fuqun Huang and Lorenzo Strigini. 2023. HEDF: A Method for Early Forecasting Software Defects Based on Human Error Mechanisms. *IEEE Access* 11 (2023), 3626–3652. doi:10.1109/access.2023.3234490
[24] Jiajun Jiang, Fengjie Li, Zijie Zhao, Zhirui Ye, Mengjiao Liu, Bo Wang, Hongyu Zhang, and Junjie Chen. 2025. Boosting Redundancy-based Automated Program Repair by Fine-grained Pattern Mining. arXiv:2312.15955 [cs.SE] https://arxiv.org/abs/2312.15955
[25] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.
[26] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.
[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055
[28] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[29] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 2012. Model Transformation By-Example: A Survey of the First Wave. In *Conceptual Modelling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 7260)*, Antje Düsterhöft, Meike Klettke, and Klaus-Dieter Schewe (Eds.). Springer, 197–215. doi:10.1007/978-3-642-28279-9_15

[30] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25 (2020), 1980–2024.

[31] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. 2016. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 61–72.

[32] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 202 (Aug. 2025), 28 pages. doi:10.1145/3715004

[33] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[34] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[35] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[36] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.

[37] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031.

[38] Robyn R Lutz. 2000. Software engineering for safety: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 213–226.

[39] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 640–652. doi:10.1109/ICSE48619.2023.00063

[40] Matias Martinez, Jean-Rémy Falleri, and Martin Monperrus. 2023. Hyperparameter Optimization for AST Differencing. *IEEE Trans. Software Eng.* 49, 10 (2023), 4814–4828. doi:10.1109/TSE.2023.3315935

[41] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM Sigplan Notices* 46, 6 (2011), 329–342.

[42] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511.

[43] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. 383–392.

[44] Hayatou Oumarou, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Kolyang Dina Taiwe. 2015. Identifying the exact fixing actions of static rule violation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 371–379.

[45] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14 (2009), 286–315.

[46] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[47] C Lakshmi Prabha and N Shivakumar. 2020. Software defect prediction using machine learning techniques. In *2020 4th International conference on trends in electronics and informatics (ICOEI)(48184)*. IEEE, 728–733.

[48] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. 428–439.

[49] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[50] Mahesh Kumar Thota, Francis H Shajin, P Rajesh, et al. 2020. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering* 17, 4 (2020), 331–344.

[51] Dániel Varró. 2006. Model Transformation by Example. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4199)*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.). Springer, 410–424. doi:10.1007/11880240_29

[52] Dániel Varró and Zoltán Balogh. 2007. Automating model transformation by example using inductive logic programming. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo (Eds.). ACM, 978–984. doi:10.1145/1244002.1244217

[53] Yanlin Wang, Kefeng Duan, Dewu Zheng, Ensheng Shi, Fengji Zhang, Yanli Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Hongyu Zhang, Qianxiang Wang, and Zibin Zheng. 2025. Towards an Understanding of Context Utilization in Code Intelligence. arXiv:2504.08734 [cs.SE] https://arxiv.org/abs/2504.08734

[54] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*. IEEE, 590–604.

[55] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. arXiv:2406.18181 [cs.SE] https://arxiv.org/abs/2406.18181

[56] Michael Zhivich and Robert K Cunningham. 2009. The real cost of software errors. *IEEE Security & Privacy* 7, 2 (2009), 87–90.

[57] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* (2024).

A Defs Construction Details

A.1 Construction Process

We constructed Defs through three main stages.

Commit Collection. We first mined defect-fixing commits from GitHub. To ensure both diversity and practical relevance, we filtered projects based on two criteria: (1) more than 5,000 stars, and (2) containing Java code. This configuration yielded a total of 488 projects, each exhibiting active development and a rich history of real defect fixes. We restricted commits to the period between January 1, 2023 and January 1, 2025, and removed non-code changes such as documentation updates. Using the keywords "fix" and "repair" in commit messages, we collected 95,335 candidate commits from 102 popular repositories.

Checker-Based Identification. The initial keyword search using commit messages ("fix," "repair") served only as a coarse-grained entry point and was not intended to directly yield usable defect-fix data. To obtain reliable instances, we applied two complementary static analyzers: CodeQL (138 defect-related queries) and PMD (147 rules). For each commit in the candidate set, we executed the corresponding analyzers on both the parent and the fixed revision. If a checker-reported warning disappeared in the fixed revision, the commit was considered a potential defect fix for that defect. From such commits, we extracted method-level code changes by differencing the defective and fixed versions, associating each pair with the checker that identified it. This process produced method-level defective–fix pairs, which were grouped according to the checker that identified them and later used for clustering. Consistent with the intra-procedural focus of our approach, we consider only code changes confined within a single method during dataset construction. Under this setting, PMD produces 31,598 changed warnings, among which 10,606 (about 34%) correspond to method-level code changes, and CodeQL identifies 9,663 changed warnings, of which 7,890 (about 82%) are confined to a single method.

Grouping and Validation. To organize similar defect–fix pairs, we grouped defect–fix pairs into clusters using a hierarchical clustering algorithm that combines checker labels with edit-structure similarity. To ensure validity, the first two authors independently annotated all clusters after aligning on annotation criteria, achieving substantial agreement (Cohen's kappa ≈ 0.8). Disagreements were resolved through discussion.

After this process, the final dataset consists of 1,287 defect–fix pairs organized into 319 clusters, covering 63 defect categories. Further statistics on distribution and diversity are presented in the next section.

Defect-Inducing Commit Identification (RQ2). For RQ2, we further identify the defect-inducing commit for each defect-fix pair following the SZZ algorithm. Specifically, we apply `git blame` to the modified lines in the defective version to trace back to the commits that last modified those lines. We then evaluate these commits in reverse chronological order using the corresponding static checker until the defect-inducing commit is identified.

A.2 Dataset Statistics

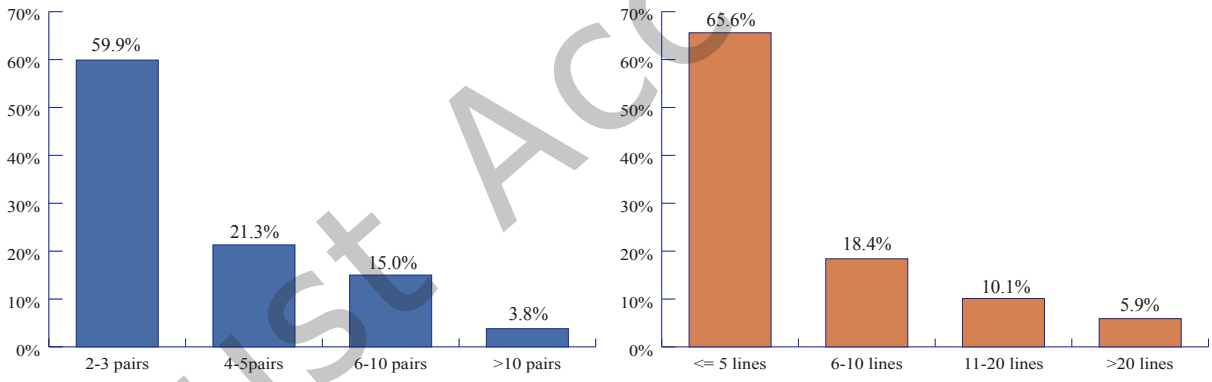
To highlight the diversity and representativeness of Defs, we provide additional statistics on its composition. Table 5 reports the top-5 checkers from each subset (CodeQL and PMD), showing both the number of clusters

Table 5. Top-5 checkers per subset in Defs. In this table, #Clus. denotes the number of clusters for each checker, #Pairs denotes the number of defect-fix pairs for each checker, and the Avg. Size denotes the average number of defect-fix pairs per cluster.

CodeQL				PMD			
Checker	#Clus.	#Pairs	Avg. Size	Checker	#Clus.	#Pairs	Avg. Size
Ineff. primitive constructor	14	52	3.71	Use collection is empty	18	88	4.89
Deprecated method/ctor invocation	10	31	3.10	JUnit5 test pkg private	18	64	3.56
Missing Override annotation	5	28	5.60	Primitive wrapper init	17	74	4.35
Missing catch of NumberFormatEx	4	9	2.25	JUnit use expected	17	53	3.12
Expression always true	3	30	10.00	System println	14	75	5.36

and the total defect–fix pairs they contribute. The CodeQL subset emphasizes defect-oriented categories such as *missing exception handling* and *always-true expressions*, while the PMD subset covers widely recurring unit-test and best-practice rules (e.g., *JUnit conventions*, *logging*, *system output*). This complementary coverage ensures that the dataset is not biased toward a single defect type and reflects both semantic defects and developer practice issues that frequently arise in real-world projects.

Fig. 6a shows the cluster size distribution in Defs. Most clusters are small: 59% contain only 2–3 examples, and another 21.3% contain 4–5 examples. Mid-sized clusters (6–10 examples) account for 15%, while large clusters (10+ examples) make up 3.8%. This long-tail distribution indicates that the dataset captures both rare, fine-grained fixes and highly recurring defect patterns, providing realistic coverage of how defects appear in practice. The prevalence of small clusters further underscores the need for methods that can abstract meaningful patterns from few-shot examples—an ability that DslGen is explicitly designed to provide.



(a) Cluster size distribution.

(b) Edit size distribution.

Fig. 6. Dataset statistics for Defs

Finally, we examine the edit scope of defect–fix pairs. As the results shown in Fig. 6b, most fixes are concise: about 66% of pairs involve no more than five changed lines. Nevertheless, 18% include 6–10 lines, 10% require 11–20 lines, and nearly 6% exceed 20 lines. This distribution confirms that Defs reflects not only the common short edits typical of defect fixes, but also medium and large modifications, thereby offering diversity in modification complexity that is crucial for evaluating generalizable defect patterns.