# ASTraMut: Learning Java Mutation Operators from Real-World Bug-Fix Patterns

Doyeol Oh
KAIST
Daejeon, Republic of Korea
ohdoyoel@kaist.ac.kr

Hyunji Park
KAIST
Daejeon, Republic of Korea
michellehyunjipark@kaist.ac.kr

Junseo Jang
KAIST
Daejeon, Republic of Korea
lily0402@kaist.ac.kr

Dongjae Lee
KAIST
Daejeon, Republic of Korea
dongjae.lee00@kaist.ac.kr

## Abstract

ASTraMut is a Java mutation operator learner that extracts AST-level edit patterns from real bug-fix commits and applies the learned fix patterns in reverse to generate mutants. Conventional mutation testing tools rely on manually designed operator sets, which are easy to apply but difficult to extend systematically and may generate mutants that do not resemble real faults. ASTraMut addresses this limitation by learning recurring fix patterns from bug-fix corpora, generalizing them through anti-unification, clustering similar edits, and ranking the resulting patterns by support and specificity. In our Defects4J evaluation, ASTraMut automatically learned a mutation operator set that produced a 1.94 % lower average mutation score than the manually designed baseline on the same relevant test suites. These results show that ASTraMut can learn bug-pattern-based mutation operators that are at least competitive with a widely used baseline, and that produce mutants which are slightly harder to kill.

## 1 Introduction

Mutation testing is a fault-based testing technique that evaluates a test suite by introducing small artificial faults, called *mutants*, into a program and checking whether the tests detect them [5, 8]. Because it provides a concrete signal about missing test coverage, mutation testing has been widely studied and adopted in practice, with language-specific tools such as PIT for Java and the JVM [4], Stryker for JavaScript, TypeScript, .NET, and Scala [15], mutmut for Python [6], Infection for PHP [7], Mull for C and C++ [13], and cargo-mutants for Rust [14].

The usefulness of this signal depends on whether the generated mutants are meaningful. Traditional mutation testing tools such as PIT [4] rely on manually designed operator sets. These operators are practical and widely used, but they are not necessarily derived from the way developers actually introduce and fix bugs. Prior studies report two related concerns: generated mutants may not resemble realistic faults [10], and developers may regard many surviving mutants as trivial or unactionable [2]. As a result, improving a test suite only against such mutants can reward tests that kill artificial changes rather than tests that catch real bugs.

A promising solution is to derive mutation operators from real bug-fix history instead of defining them manually. Brown et al. [3] introduced *wild-caught mutants*, which harvest mutation operators from revision histories by reversing likely corrective patches. A patch of the form $Buggy \mapsto Patched$ can be interpreted as a mutation operator $Patched \mapsto Buggy$: the fixed-side shape *Patched* is matched against clean code, and the buggy-side shape *Buggy* is reintroduced as a mutant. This work establishes an important direction for history-driven mutation testing: recurring fixes can be turned into reusable fault-injection rules grounded in defects that developers have actually fixed.

However, wild-caught operators are mostly mined at the lexical-token level. They are also limited to small patch fragments. This makes it hard to learn reusable language-level rewrites. It also creates patterns that are not safely reversible.

ASTraMut treats this problem as AST pattern learning. It extracts fine-grained Java tree edits. It then groups recurring fixes and generalizes them with anti-unification. Before applying the learned patterns in reverse, it removes unsafe and low-quality patterns using mutation-specific checks.

This design produces reusable Java mutation operators. The operators preserve syntactic structure. They also prevent unbound replacement pattern variables and reduce project-specific literal values. The best ASTraMut variant, Merged Top 100, achieves an average mutation score of **69.26%**. In comparison, PIT Default achieves **73.99%**, and PIT Full achieves **70.63%**. These results suggest that learned bug-fix patterns can generate mutants that are competitive with manually designed operators. They can also be slightly harder to kill.

Our contributions are:

- **Learning mutation operators through AST level pattern generalization**: We implement ASTraMut, a mutation operator learning system for Java programs. ASTraMut generalizes AST-level patch patterns using anti-unification algorithm. As far as we know, this is the first anti-unification-based mutation operator learning system.
- **Elimination rules for a high-quality mutation operator set**: We add mutation-specific safety and quality controls that make learned fix patterns usable as reversed mutation operators (§ 3.5). These controls include a new sound pattern elimination rule, the *mutation-safety elimination rule*, that prevents unbound pattern variables, plus oversized ground singleton elimination rule, identifier-only root elimination rule, destructive root-pattern-variable elimination rule, and dataset-aware normalization steps that suppress project-specific noise.

- **Learning high-quality mutation operator set automatically**: Full-scale training produces **436 mutation operators**. These learned operators generate mutants with a mutation score **1.94% – 6.39% lower** than the baseline for the same test suite, indicating that ASTraMut can generate harder-to-kill mutants than conventional operators.
- **ASTraMut is publicly available at GitHub** (https://github.com/duncan020313/cs453-team5-learning-mutation-operator)

## 2 Background

### 2.1 Mutation testing

Mutation testing is a fault-based testing technique. It creates many small syntactic variants of a program, called *mutants*, and checks whether the test suite can distinguish each mutant from the original program [5, 8]. A mutant is *killed* when at least one test fails on the mutated program; otherwise it *survives*. The resulting mutation score measures the fraction of generated mutants killed by the test suite.

The intuition is that tests that catch many simple artificial faults are more likely to expose real faults. In practice, traditional mutation testing depends on manually designed mutation operators, such as replacing arithmetic operators, negating conditionals, or changing return values. Tools such as PIT [4] apply these predefined operators to generate mutants. ASTraMut takes a different approach: it learns mutation operators from existing bug-fix patches, then applies the learned patterns in reverse to generate mutants that reflect real code changes.

### 2.2 Getafix: generalizing patch patterns through anti-unification

Getafix [1] provides the learning algorithm that ASTraMut reuses for mutation operator generation. In its original program repair setting, Getafix learns automated bug-patch patterns from human-written commits. Given a dataset of before/after AST pairs, it (i) extracts multi-granularity *concrete edits*, (ii) generalizes pairs of edits via *anti-unification* [12] into patterns with *pattern variables*, and (iii) organizes the generalizations into a hierarchical dendrogram via agglomerative clustering. The result is a set of patch patterns $Buggy \mapsto Patched$ with shared pattern variables across *Buggy* and *Patched*. For program repair, *Buggy* represents the buggy shape and *Patched* represents the fixed shape. ASTraMut interprets the same pattern in the opposite direction: *Patched* is the clean code to match, and *Buggy* is the realistic fault to reintroduce. Thus, reversing a learned fix pattern turns Getafix's repair abstraction into a mutation operator learned from real patches.

## 3 Mutation Operator Learning

Figure 1 gives an overview of the end-to-end pipeline. The learning stage consumes the extracted tree diffs and produces a ranked set of reusable mutation patterns. Each tree diff is decomposed into multi-granularity concrete edits. We progressively generalize using anti-unification inside a hierarchical agglomerative clusterer, then apply elimination rules to the candidates, and then rank them. As a post-processing step, patterns with identical canonical signatures are merged across training runs, producing the unified mutation operator set consumed by the downstream mutation engine.

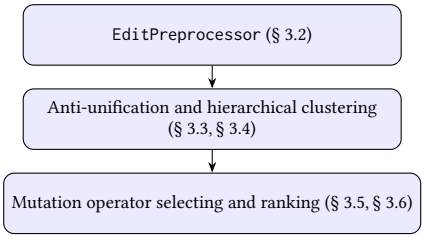


**Figure 1: The ASTraMut pattern-learning pipeline.**

### 3.1 Notation

A *tree pattern* is either a pattern variable or a labelled node with ordered children:

$$\mathsf{Tree} = \left(\mathsf{Label} \times \mathsf{Value} \times (\mathsf{Location} \times \mathsf{Tree}^+)^*\right) \cup \mathsf{PVar}. \quad (1)$$

We use Kleene notation: $X^*$ denotes a finite sequence of zero or more $X$ elements, and $X^+$ denotes one or more $X$ elements. Thus $(\mathsf{Location} \times \mathsf{Tree}^+)^*$ is the ordered child list, which may be empty, while $\mathsf{Tree}^+$ names the non-empty universe of pattern trees. A *pattern variable* $h^k$ is a pattern variable with a unique identifier $k$; two occurrences of the same identifier within a pattern must bind the same subtree. A pattern with no pattern variables is called *ground*.

An *edit pattern* is a quadruple

$$p = (Buggy, Patched, U_{Buggy}, U_{Patched}).$$

Here $Buggy, Patched \in \mathsf{Tree}^+$ are the before/after sides sharing pattern variable identifiers, so the rewrite $Buggy \mapsto Patched$ is variable-coherent. The sets $U_{Buggy}$ and $U_{Patched}$ mark the boundaries of unmodified subtrees retained for the anti-unifier (§ 3.3). A *concrete edit* is an edit pattern in which both *Buggy* and *Patched* are ground. The pipeline starts with concrete edits extracted from individual bug fixes and progressively generalizes pairs of them into patterns with pattern variables.

### 3.2 Preprocessing

`EditPreprocessor` produces a multi-granularity set of edits from a single GumTree diff. Two ideas drive the design. First, a single developer patch can express the same change at multiple AST levels, so the `EditPreprocessor` deliberately emits candidates at several granularities and lets the clusterer pick the granularity that produces the largest, most consistent group. Second, rather than preserving every concrete value in a diff, we want to characterize the reusable structure of the edit. Identifier names and user-defined type names are therefore abstracted to pattern variables, while numeric literals are canonicalized according to their AST context *at extraction time*, before anti-unification generalizes the edits further.

*3.2.1 Multi-level diff extraction.* The `EditPreprocessor` treats a patch not as a single edit, but as a source of edit candidates at multiple AST levels. Starting from the changed region, it compares progressively larger enclosing subtrees and emits a concrete edit

| Identifier | Parent type | Decision |
|---|---|---|
| *Source:* `int result = obj.getItems();` | | |
| `result` | `VariableDeclarationFragment` | `?h0` |
| `obj` | `MethodInvocation` | `?h1` |
| `getItems` | `MethodInvocation` | `?h2` |

**Figure 2: Identifier canonicalization in a single statement.**

whenever the before and after versions of that subtree differ. Thus the same patch can yield candidates at four levels:

(1) *Leaf-level candidates* are extracted from terminal AST nodes, such as constants, variable identifiers, or token-level values.
(2) *Expression-level candidates* treat a complete expression as the edit unit, such as `a + 1`, `x > 0`, `foo(y)`, or `a && b`.
(3) *Statement-level candidates* treat one executable statement as the edit unit, such as `x = a + 1;`, `return foo();`, or an `if` statement.
(4) *Method-level candidates* treat a whole method body or method declaration as the edit unit.

Keeping all four levels lets the clustering stage identify the level at which the edit becomes most reusable.

The `EditPreprocessor` emits a candidate only when the before and after roots have the same AST node type. This constraint keeps the learned rewrite within the same syntactic category when it is later applied as a mutation operator, reducing the chance that a generated mutant replaces an expression, statement, or declaration with an incompatible construct and fails to compile.

*3.2.2 Canonicalization.* Concrete values that are specific to one patch can hide the reusable structure of an edit. For instance,

$$\texttt{foo.length} \mapsto \texttt{foo.size}$$

would only ever match code that uses `foo`. Real bug-fix patterns generalize across receivers: the API rename matters, not the variable name. `EditPreprocessor` therefore canonicalizes program-specific identifiers into pattern variables while preserving the surrounding AST structure. Identifiers that denote the same matched source and destination name are assigned the same pattern variable identifier, so repeated occurrences of the same program binding remain coherent across the before and after sides of the edit.

For a fix like `x.method()` $\mapsto$ `x.method(arg)`, the receiver `x` and the method name denote the same matched identifiers on both sides, so they receive the same pattern variable identifiers. Only the newly introduced argument receives a fresh pattern variable, yielding

$$\texttt{?h0.?h1()} \mapsto \texttt{?h0.?h1(?h2)}.$$

Figure 2 traces identifier canonicalization on a one-statement fix. The receiver, method, and target variable are canonicalized to pattern variables, allowing the learned pattern to match calls of this shape regardless of the concrete names used in the original fix.

Numeric literals follow the same logic, except that boundary values $\{0, \pm 1, 0.0, \pm 1.0, \ldots\}$ carry mutation semantics directly as boundary-condition changes, so they remain literal. Every other numeric-literal label collapses to the canonical sentinel `__MAGIC__`.

| Source | After abstraction | Effect |
|---|---|---|
| `int count;` | `int ?h0;` | literal primitive type |
| `FooBuilder<Bar> fb;` | `?h1<?h2> ?h3;` | pattern variables |

**Figure 3: Primitive and user-defined type treatment during abstraction.**

Without canonicalization, on the fix

$$\texttt{return arr[42];} \mapsto \texttt{return arr[43];}$$

the literals `42` and `43` occupy the same syntactic role but have different labels, so anti-unification falls through Case C (line 16) into Case D (line 17) and emits two *independent* pattern variables:

$$\texttt{return arr[?}h_a\texttt{]} \mapsto \texttt{return arr[?}h_b\texttt{]}.$$

This immediately fails the unbound pattern variable elimination rule (§ 3.5) because $?h_a$ has no counterpart on the after-side. The pattern is therefore dropped, and the same fate awaits the analogous `arr[7]` $\mapsto$ `arr[8]` fix; the underlying non-boundary numeric-literal replacement family yields no operator at all. With canonicalization, both extractions produce

$$\texttt{return arr[__MAGIC__]} \mapsto \texttt{return arr[__MAGIC__]}$$

directly, so Case A (line 11) returns the same pattern verbatim, support accumulates across the training dataset, and the single high-support operator survives. The right-hand-side `__MAGIC__` is later resolved by sampling from the empirical numeric-literal distribution at mutation time (§ 3.7).

*3.2.3 Primitive and user-defined type handling.* Identifier canonicalization is too generous on its own. A project that frequently declares a private class such as `FooContextBuilder` would lift hundreds of patterns specific to that user-defined class into the mutation operator set. The `EditPreprocessor` therefore preserves language-level type information while abstracting project-defined type names. Primitive types such as `int`, `boolean`, and `double` are represented as dedicated AST nodes and remain literal mutation targets. Type identifiers are treated conservatively across simple, qualified, and parameterized type contexts: if they denote user-defined classes or project-specific placeholders, they are converted to pattern variables.

The contrast in Figure 3 illustrates why this matters. Primitive-type changes preserve their mutation semantics. User-defined type names become pattern variables, preventing project-specific class names from polluting the mutation operator set.

## 3.3 Anti-unification

The generalization step follows Kutsia et al.'s anti-unification [12] over pairs of edit patterns. Algorithm 1 formalizes the procedure.

Intuitively the four cases mean: Case A (line 11) keeps structurally identical subtrees verbatim — the developer-intuitive notion of an "unchanged region." Case B (line 13) collapses entire subtrees both marked as unmodified context into a single shared pattern variable. Case C (line 16) aligns isomorphic subtrees structurally and recurses, accumulating shared structure where possible. Case D

**Algorithm 1** AntiUnify: anti-unification of two edit patterns.

1: **Function** AntiUnify($p_1, p_2$) *generalize two edit patterns*
2: RenamePVars($p_1, p_2$)
3: $\alpha \leftarrow$ PatternVariableAllocator()
4: $(B_1, P_1, U_{B_1}, U_{P_1}) \leftarrow p_1$
5: $(B_2, P_2, U_{B_2}, U_{P_2}) \leftarrow p_2$
6: $Buggy \leftarrow$ GeneralizeTree($B_1, B_2, U_{B_1}, U_{B_2}, \alpha$)
7: $Patched \leftarrow$ GeneralizeTree($P_1, P_2, U_{P_1|}, U_{P_2}, \alpha$)
8: **return** $(Buggy, Patched)$
9: **Function** GeneralizeTree($t_1, t_2, U_1, U_2, \alpha$)
10: **if** $t_1 \equiv t_2$ **then**
11: **return** $t_1$ *Case A: identity*
12: **if** $t_1 \in U_1 \wedge t_2 \in U_2$ **then**
13: **return** $\alpha(t_1, t_2)$ *Case B: collapse unmod region*
14: **if** type($t_1$) = type($t_2$) $\wedge$ label($t_1$) = label($t_2$) $\wedge$ |kids($t_1$)| = |kids($t_2$)| **then**
15: $c_i \leftarrow$ GeneralizeTree(kids$(t_1)_i$, kids$(t_2)_i, U_1, U_2, \alpha$) for each $i$
16: **return** TreeNode(type($t_1$), label($t_1$), $c_1, \ldots, c_n$) *Case C: recurse on children*
17: **return** $\alpha(t_1, t_2)$ *Case D: fresh pattern variable*
18: **Function** RenamePVars($p_1, p_2$) *avoid cross-pattern id collisions*
19: **Function** PatternVariableAllocator() *memoize one pattern variable per subtree pair*

(line 17) maps every remaining incompatibility (different type, label, arity, or pattern variable/node mix) to a pattern variable keyed by the pair, so encountering the same $(t_1, t_2)$ a second time returns the same pattern variable identifier. The pattern variable allocator $\alpha$ is shared between the calls on $(Buggy_1, Buggy_2)$ and $(Patched_1, Patched_2)$: when the same subtree pair appears on both sides the allocator returns the same pattern variable on the after-side, so the resulting rewrite rule transports bindings coherently from *Buggy* into *Patched*.

A fresh-rename pass on the inputs (line 1 of Algorithm 1) is required because the `EditPreprocessor` already minted pattern variable identifiers; without renaming, two patterns that independently used `?h0` for different subtrees would silently fuse those bindings during anti-unification.

The following derivation traces Algorithm 1 on two concrete null-pointer-exception guard fixes. Start with two edits that have the same fix shape but different receiver and method names:

$$e_1 : \texttt{x.foo();} \mapsto \texttt{if (x==null) return; x.foo();},$$
$$e_2 : \texttt{y.bar();} \mapsto \texttt{if (y==null) return; y.bar();}.$$

Canonicalization (§ 3.2.2) runs before anti-unification, so the receiver and method names are already canonicalized while preserving the repeated receiver binding inside each edit:

$$\widehat{e_1} : \texttt{?h0.?h1();} \mapsto \texttt{if (?h0==null) return; ?h0.?h1();},$$
$$\widehat{e_2} : \texttt{?h0.?h1();} \mapsto \texttt{if (?h0==null) return; ?h0.?h1();}.$$

**Algorithm 2** Hierarchical agglomerative clustering of concrete edits.

1: **Function** HierarchicalCluster($E, b_{\max}$)
2: $\mathcal{B} \leftarrow$ RootPartitions($E$)
3: $C \leftarrow \emptyset$
4: **for** each $B \in \mathcal{B}$ **do**
5: $\mathcal{S} \leftarrow$ SplitPartition($B, b_{\max}$)
6: **for** each $B' \in \mathcal{S}$ **do**
7: $\mathcal{L} \leftarrow$ SingletonClusters($B'$)
8: $Q \leftarrow \emptyset$
9: **for** each pair $(a, b) \in \mathcal{L} \times \mathcal{L}$ **do**
10: TryEnqueue($a, b, Q$)
11: **while** $|\mathcal{L}| > 1$ **and** $Q \neq \emptyset$ **do**
12: $(a, b, p) \leftarrow$ PriorityDequeue($Q$)
13: **if** $a \notin \mathcal{L}$ **or** $b \notin \mathcal{L}$ **then**
14: **continue**
15: $c.rep \leftarrow p$; $c.members \leftarrow a.members \cup b.members$
16: $\mathcal{L} \leftarrow (\mathcal{L} \setminus \{a, b\}) \cup \{c\}$
17: **for** each $x \in \mathcal{L} \setminus \{c\}$ **do**
18: TryEnqueue($c, x, Q$)
19: $C \leftarrow C \cup \mathcal{L}$
20: **return** Merge($C$)
21: **Function** TryEnqueue($a, b, Q$)
22: $p \leftarrow$ AntiUnify($a.rep, b.rep$) *Algorithm 1*
23: **if** $p.before$ is exactly a root PVar **or** $p.after$ is exactly a root PVar **then**
24: **return**
25: PriorityEnqueue($Q$, $(a, b, p)$, PriorityKey($p$))
26: **Function** RootPartitions($E$) *group edits by anti-unifiable root*
27: **Function** SplitPartition($B, b_{\max}$) *cap partition size for clustering*
28: **Function** SingletonClusters($B'$) *initialize one cluster per edit*

The two normalized edits are structurally identical, so AntiUnify reaches Case A (line 11) at the root and returns the same rewrite:

$$\text{AntiUnify}(\widehat{e_1}, \widehat{e_2}) = (Buggy_{null}, Patched_{null}),$$
$$Buggy_{null} = \texttt{?h0.?h1();},$$
$$Patched_{null} = \texttt{if (?h0==null) return; ?h0.?h1();}.$$

Without extraction-time conversion, the labels `x`/`y` and `foo`/`bar` would clash in Case D (line 17) and reach the same final pattern only after introducing semantically meaningless intermediate pattern variables.

## 3.4 Hierarchical clustering

The clustering stage performs agglomerative clustering of concrete edits (Algorithm 2). The merge priority is

$$\big(U_{\text{after}},\ P_{\text{total}},\ I\big),$$

where $U_{\text{after}}$ is the number of unbound pattern variables on the after-side, $P_{\text{total}}$ is the total number of pattern variables, and $I$ is the insertion sequence. This ordering prefers patterns that bind all of their after-side pattern variables, that is, patterns that will survive the mutation-safety elimination rule, and breaks ties deterministically.

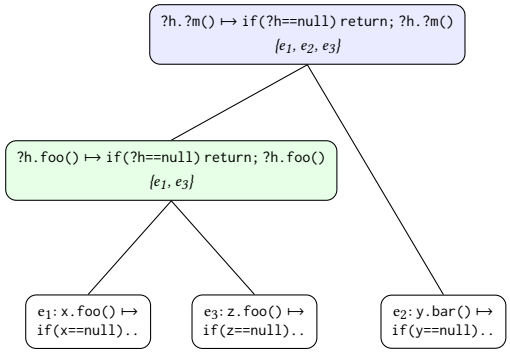


**Figure 4: A two-level dendrogram derived from three null-pointer-exception guard fixes.**

Two engineering extensions accommodate the dataset size. *Root-signature partitioning.* Two edits whose top-level node disagrees on type/label/arity necessarily anti-unify to a root-only pattern variable by Case D (line 17), which the downstream mutation operator set would discard as destructive (the destructive root-pattern-variable elimination rule, § 3.5). We never need to consider such cross-partition pairs, and the $O(n^2)$ initial candidate pool shrinks by an order of magnitude. *Bounded partitioning.* Any partition above a fixed size bound is split into chunks, each clustered independently. Bounded partitioning can split semantically equivalent edits across chunks, so a final pass re-merges clusters whose representatives share a canonical pattern variable-renamed signature, restoring the equivalence classes that the partitioning broke.

Figure 4 shows a two-level dendrogram produced by this procedure on three null-pointer-exception guard fixes. Its leaves are concrete edits and its interior nodes are anti-unified patterns (Algorithm 1). Merging $e_1$ and $e_3$, which share the method name `foo`, yields a single-pattern variable pattern over the receiver. The further merge with $e_2$ generalizes to a two-pattern variable pattern over both receiver and method.

## 3.5 Elimination Rules

Each representative of the cluster is evaluated by five elimination rules before being admitted to the mutation operator set. The no-op elimination rule and unbound pattern variable elimination rule are correctness conditions for reversed mutation rewriting; the remaining three are elimination rules for better quality of mutation operators.

*no-op elimination rule.* Anti-unification can produce trivial identities once enough labels collapse to pattern variables; such patterns generate no mutants and are dropped.

*unbound pattern variable elimination rule.*

$$\text{pvars}(\text{before}) \subseteq \text{pvars}(\text{after}). \tag{2}$$

The mutation engine applies each learned fix in reverse: it matches the after-side *Patched* against the target program (binding pattern variables from the match) and rewrites to *Buggy*. If a pattern variable identifier appears in *Buggy* but not in *Patched*, reversed application would need to materialize that pattern variable. The unbound pattern variable elimination rule rules out such patterns. For example, after the identifier-to-pattern variable step (§ 3.2.2) a fix that drops the second operand of a comparison may yield

```
?h0 < ?h1 ↦ ?h0 < 0;
```

this fails the unbound pattern variable elimination rule because `?h1` appears in the LHS but not in the RHS, so reversed application cannot reconstruct the original expression.

*oversized ground singleton elimination rule.* If a cluster has only one member (support = 1), the representative is fully ground (pvars = 0), and exceeds a small size threshold, it is dropped. Reaching pvars = 0 on a deep pattern is unusual once the identifier-to-pattern variable step (§ 3.2.2) has converted every identifier outside a literal context: the survivors are typically declarations or initializers whose nodes all sit in type, annotation, or import contexts, such as a long fully-qualified-type declaration with no method calls or variables. Such patterns reflect one project-specific edit verbatim and have essentially zero chance of matching anywhere else.

*identifier-only root elimination rule.* A pattern is dropped when both sides are identifier-only trees, with no enclosing expression, declaration, type, or call context. Such patterns degenerate to raw name rewrites; the engine would need typed context to know whether the substitution type-checks.

*destructive root-pattern-variable elimination rule.* If *Buggy* is a pattern variable at the root whose identifier does not appear on the after-side — or symmetrically, *Patched* is a root pattern variable absent from the before-side — the pattern is dropped. This excludes the degenerate case "replace the entire program with a fresh subtree." The check is asymmetric (only when the pattern variable id is *absent* on the opposite side) to preserve legitimate wrap/unwrap patterns such as

```
?h0 ↦ Math.abs(?h0),
```

where the root pattern variable `?h0` *is* present on the opposite side and represents a coherent binding.

## 3.6 Ranking

We score each retained pattern by

$$\begin{aligned} \text{specificity}(p) &= 1 - \frac{|\text{pvars}(p)|}{|\text{nodes}(p)|}, \\ \text{score}(p) &= \text{support}(p) \cdot \text{specificity}(p). \end{aligned} \tag{3}$$

A high-scoring pattern is one that recurs across many bug fixes (high support) *and* retains structural detail (few pattern variables per node). This combination penalizes both one-off project-specific fixes and over-generalized patterns whose entire shape has collapsed into pattern variables.

*Operator-set normalization.* After applying elimination rules, patterns are normalized by a canonical signature that renames pattern variables in preorder and serializes the resulting tree. Duplicate patterns produced by bounded partitioning, per-category learning, or separate training datasets are merged by this signature, with support summed and contributing sources recorded. The same signature criterion is used in § 4.3.1 to count dataset-specific and dataset-shared patterns in the merged operator set.

**Table 1: Training datasets used for learning mutation operators.**

| Dataset | Fix pairs | Patch scope |
|---|---|---|
| ManySStuBs4J [11] | 63 923 | Statement |
| Bugs2Fix [16] | 46 680 | Method |

### 3.7 Mutant generation

The learned mutation operator set is not a set of executable mutants by itself, but a ranked set of tree rewrite patterns learned from Java bug-fix edits. ASTraMut turns these patterns into mutants by applying each learned fix pattern in the reverse direction. Given a learned pattern *Buggy* $\mapsto$ *Patched*, where *Buggy* represents the buggy shape and *Patched* represents the fixed shape, the mutation engine constructs a mutation pattern in the form of *Patched* $\mapsto$ *Buggy*. It then searches the target Java program for code fragments that match *Patched* and rewrites the matched fragment into *Buggy*.

The mutation engine first parses the target Java source file and collects AST fragments that can serve as replacement candidates. Each candidate is converted into the same tree-pattern representation used by the learned operators. The matcher then compares the fixed side of the reversed pattern with each candidate pattern. During matching, pattern variables in the learned pattern are recorded in a binding map from pattern variable identifiers to concrete subtrees in the target program. If the same pattern variable appears again, the matcher checks whether it is equal to the previously bound subtree. This guarantees the consistency of pattern variable bindings.

Once a match is found, ASTraMut instantiates the buggy side of the reversed pattern using the captured bindings and constructs a replacement pattern. This is why the mutation-safety elimination rule in § 3.5 is necessary: every pattern variable that appears in the generated replacement must already have been bound while matching the fixed side. Therefore, if a pattern variable cannot be resolved from the binding map, instantiation fails and the corresponding candidate is skipped.

The engine emits a mutant only when the matched fragment can be replaced by a syntactically valid Java AST fragment generated from the instantiated pattern. Patterns outside the supported translation boundary between learned tree patterns and editable Java syntax are skipped. This helps keep the generated mutants well-defined and directly traceable to real bug-fix patterns.

## 4 Experiments

### 4.1 Experimental Setup

We train on two datasets and combine them with deduplication. They differ in snippet granularity, so we use two granularity configurations:

ManySStuBs4J contains single-statement fixes. Each one-line snippet is wrapped in a synthetic method-body skeleton, and the wrapper nodes are stripped before diff extraction. Bugs2Fix contains method-level bug-fix pairs from the CodeXGLUE small train split and is parsed directly at method granularity. We treat the two datasets as independent training sources with equal status, rather than using one as a corrective supplement for the other.

We evaluate ASTraMut on Defects4J 3.0.1 [9]. For each bug, we:

(1) extract the methods modified by the developer patch and use only these patched methods as mutation targets;
(2) obtain the bug's relevant test suite from the `relevant.tests` metadata field provided by Defects4J;
(3) run mutation testing only on the patched methods and only against these relevant tests;
(4) compute the mutation score as the ratio of killed mutants to generated mutants.

Finally, we report average values across all evaluated bugs.

For the external baseline, we use PIT [4], a widely used mutation-testing system for Java. We evaluate both its default operator set and its full operator set. For each bug, we run the relevant tests and measure the mutation score over the generated mutants. We define mutation score as

$$\text{mutation score} = \frac{|\text{killed mutants}|}{|\text{generated mutants}|}.$$

If no mutant is generated for a bug, we define its mutation score as 1.

### 4.2 RQ1: Does ASTraMut Generate More Difficult Mutants to Kill?

Table 2 reports the main mutation-testing result. Lower mutation score means that the same relevant test suites kill a smaller fraction of generated mutants, so the mutants are harder for the existing tests to kill. For each ASTraMut training source, *Top 100* uses only the 100 highest-ranked learned operators according to the support–specificity score in § 3.6, while *Full* uses all learned operators that remain after applying elimination rules and deduplication. For PIT, *Default* is the standard mutator set and *Full* enables the complete available PIT mutator set. The best ASTraMut variant is the merged top-100 operator set, with an average mutation score of 0.6926. The best PIT variant is the full operator set, with an average mutation score of 0.7063. Thus, ASTraMut's best setting lowers the mutation score by 1.94 % relative to the best PIT setting.

The mutant count comparison shows a slightly different trade-off. The best-scoring ASTraMut setting generates 115.40 mutants on average, while the best-scoring PIT setting generates 135.21. However, the merged full ASTraMut setting generates 158.64 mutants, exceeding PIT full while keeping nearly the same mutation score as the top-100 ASTraMut setting (0.6933 vs. 0.6926). Compared with PIT default, the best-scoring ASTraMut setting generates 2.8× as many mutants and achieves a lower mutation score.

### 4.3 RQ2: Can ASTraMut Learn Mutation Operators Not in PIT?

*4.3.1 Learned Operator Sets.* Training produces three artefacts: a ManySStuBs4J mutation operator set, a Bugs2Fix mutation operator set, and a cross-source merge unified by canonical signature (§ 3.6).

The merged operator set contains 436 learned patterns. The 55 patterns shared across the two training datasets — 12.6 % of the merged set — are elementary rewrite shapes dominated by boolean swaps, numeric-boundary flips, and comparison inversions. These

**Table 2: Mutation-testing results on Defects4J by mutation-operator variant.**

| Variant | Avg. generated mutants | Avg. killed mutants | Avg. mutation score | Δ vs. PIT (Full) |
|---|---|---|---|---|
| ASTraMut (Merged, Top 100) | 115.40 | 70.85 | **69.26 %** | −1.94 % |
| ASTraMut (Merged, Full) | 158.64 | 79.20 | 69.33 % | −1.84 % |
| ASTraMut (ManySStuBs4J, Top 100) | 86.55 | 50.26 | 69.95 % | −0.96 % |
| ASTraMut (ManySStuBs4J, Full) | 90.92 | 52.98 | 69.87 % | −1.08 % |
| ASTraMut (Bugs2Fix, Top 100) | 106.58 | 66.65 | 69.41 % | −1.73 % |
| ASTraMut (Bugs2Fix, Full) | 127.76 | 81.29 | 70.61 % | −0.03 % |
| PIT (Default) | 41.10 | 25.41 | 73.99 % | +4.76 % |
| PIT (Full) | 135.21 | 84.07 | 70.63 % | 0.00 % |

**Table 3: Variants of mutation operator set**

| Variant | # Operators | Coverage |
|---|---|---|
| ASTraMut (Merged, Top 100) | 100 | top-100 of 436 by score |
| ASTraMut (Merged, Full) | 436 | all learned mutation operators from both datasets |
| ASTraMut (ManySStuBs4J, Top 100) | 100 | top-100 of 257 by score |
| ASTraMut (ManySStuBs4J, Full) | 257 | all learned from ManySStuBs4J |
| ASTraMut (Bugs2Fix, Top 100) | 100 | top-100 of 234 by score |
| ASTraMut (Bugs2Fix, Full) | 234 | all learned from Bugs2Fix |
| PIT (Default) | 11 | conditionals, arithmetic, return values |
| PIT (Full) | 26 | default + constructor/method call, switch, `BigDecimal` |

**Table 4: Final mutation-operator-set sizes after all five elimination rules and cross-run deduplication.**

| Training data | Size | Patterns |
|---|---|---|
| ManySStuBs4J | 63 923 | 257 |
| Bugs2Fix | 46 680 | 234 |
| Merged | 110 603 | **436** |

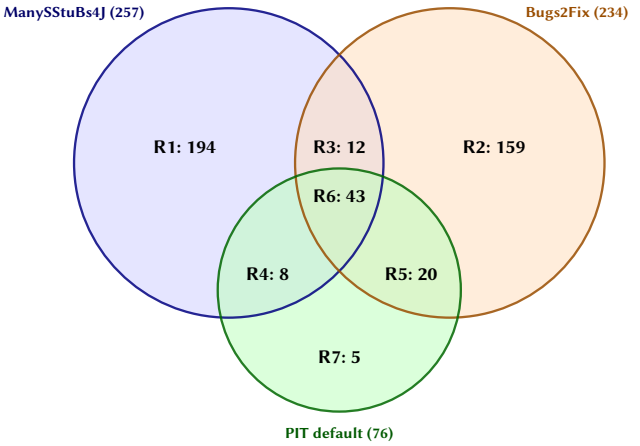


**Figure 5: The Overlap between learned pattern sets and PIT mutation operators**

are the kinds of one-token rewrites that overlap naturally with conventional mutation operators. The remaining dataset-specific patterns are where ASTraMut differs from PIT. We therefore inspect concrete learned rewrites and ask whether an equivalent PIT mutator can produce the same mutant.

*representative patterns not covered by PIT..* The added value of ASTraMut lies where the standard PIT operator set does not reach: identifier-aware API swaps, structural wraps, and guard rewrites. Table 5 lists six concrete patterns drawn from `patterns-merged.json`. All patterns are written as mutation rewrites *Patched* ↦ *Buggy*, meaning that they are already reversed from the learned fix direction. PIT mutators are bytecode-level transformations over a fixed mutator set: its default group covers conditional boundaries, increments, negation of numeric variables, arithmetic replacement, conditional negation, return-value replacement, and void-call removal; optional and experimental groups add call removal, constructor-call replacement, inline-constant changes, conditional forcing, and arithmetic/relational variants [4].

The commonality across these examples is not high frequency in one corpus, but the shape of what was learned from fixes. E1–E3 are identifier-conditioned semantic rewrites: the operator is tied to a recurring source-level name relationship, such as substituting one library class, exception class, or boxed type for another. Such mutations are invisible to a fixed mutator set unless that set explicitly encodes the pair of names. E4 and E5 are predicate rewrites. They preserve a pattern variable-bound expression and change the surrounding guard structure, either by adding a negation around the entire predicate or by removing one disjunctive clause. This differs from PIT's built-in conditional operators, which replace comparison boundaries, negate conditional bytecode, or force a

**Table 5: Representative learned patterns from `patterns-merged.json` that fall outside PIT's built-in operator set.**

| | Pattern (*Patched* ↦ *Buggy*) | Why this is outside PIT |
|---|---|---|
| E1 | `StringBuilder` ↦ `StringBuffer` | Library-specific class names are not part of PIT's fixed bytecode mutator set. |
| E2 | `IllegalArgumentException` ↦ `IllegalState Exception` | Exception classes are semantic identifiers, not relational, arithmetic, return, constant, or call-removal targets. |
| E3 | `Long` ↦ `Integer` | PIT can change primitive return values, but does not rewrite boxed type names in source-level type positions. |
| E4 | `?h0` ↦ `!?h0` | This wraps a whole predicate; `NEGATE_CONDITIONALS` rewrites conditional bytecode rather than learning arbitrary predicate templates. |
| E5 | `?h0 \|\| ?h1` ↦ `?h0` | The learned rewrite weakens a guard by dropping one disjunct, not by replacing a comparator or forcing the entire conditional. |
| E6 | `Block[?h]` ↦ `<empty Block>` | The mutation changes method-body shape; PIT removes selected calls or assignments but does not learn block-level rewrite templates. |

**Table 6: Representative learned patterns for each variant**

| | Region (set membership) | # | Top-3 patterns |
|---|---|---|---|
| R1 | ManySStuBs4J *only* | 194 | `__MAGIC__` → `0`<br>`StringBuffer` → `StringBuilder`<br>`MethodInvocation` → `PrefixExpression` |
| R2 | Bugs2Fix *only* | 159 | `private` → `public`<br>`public` → `private`<br>`protected` → `public` |
| R3 | ManySStuBs4J ∩ Bugs2Fix only | 12 | `1` → `__MAGIC__`<br>`0` → `__MAGIC__`<br>`__MAGIC__` → `1` |
| R4 | ManySStuBs4J ∩ PIT only | 8 | `>>` → `>>>`<br>`<=` → `!=`<br>`+` → `\|` |
| R5 | Bugs2Fix ∩ PIT only | 20 | `==` → `<`<br>`++` → `--`<br>`/` → `+` |
| R6 | ManySStuBs4J ∩ Bugs2Fix ∩ PIT | 43 | `1` → `0`<br>`true` → `false`<br>`false` → `true` |
| R7 | PIT *only* | 5 | `INVERT_NEGS`<br>`VOID_METHOD_CALLS`<br>`EMPTY_RETURNS`<br>`NULL_RETURNS`<br>`RETURN_VALS` |

condition, but do not learn reusable boolean-expression contexts. E6 shows the same distinction at method-body granularity: the learned operator records a block-shape change, while PIT's built-in call and assignment removals operate on selected bytecode instructions. Together, these examples show that ASTraMut generates non-PIT mutants by learning recurring AST-level edit contexts over identifiers, predicates, guards, and blocks, rather than only applying a manually enumerated set of primitive operator replacements.

*Pattern-level overlap.* Some high-ranking patterns duplicate PIT defaults, which is expected because common one-token bug-fix patterns overlap with conventional mutation operators. The non-PIT contribution appears in the mid-tier and dataset-specific patterns: API renames, argument templates, modifier changes, guard rewrites, and method-body rewrites.

*Project-specific identifier noise.* A single variable name can appear hundreds of times within one project, lifting project-specific patterns to high support. The suggested support-ratio elimination rules partially mitigate this, but a stronger fix would normalize identifier frequencies per project.

*Multi-granularity duplication.* The deliberate multi-granularity emission of § 3.2 produces both a leaf and an enclosing-context version of many patterns. Downstream tools must decide whether to apply both or only the most specific.

## 5 Conclusion

We have shown that an anti-unification pattern-learning pipeline, augmented with named mutation-targeted elimination rules and normalization over datasets (identifier-to-pattern-variable conversion, numeric-literal canonicalization, and type abstraction), transfers cleanly to the mutation-testing setting on Java. Trained on ManySStuBs4J and Bugs2Fix, ASTraMut produces **436 reusable mutation operators** (257 from ManySStuBs4J, 234 from Bugs2Fix, merged by canonical signature with 55 shared), surfacing pattern families — API class renames, condition-negation wraps over whole predicates, disjunction broadening, and method-level block-fill rewrites — that lie outside the standard PIT operator set. The full mutation operator set, training code, and mutation engine are released to enable reproduction.

## References

[1] 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).

[2] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. arXiv:2010.13464

[3] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-Caught Mutants. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 511–522. doi:10.1145/3106237.3106280

[4] Henry Coles. 2024. PIT Mutation Testing. https://pitest.org/

[5] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[6] Anders Hovmöller. 2026. mutmut: Mutation testing system for Python. https://github.com/boxed/mutmut
[7] Infection PHP. 2026. Infection: PHP Mutation Testing Framework. https://infection.github.io/
[8] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62
[9] 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 International Symposium on Software Testing and Analysis (ISSTA)*.
[10] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.
[11] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*.
[12] Temur Kutsia, Jordi Levy, and Mateu Villaret. 2014. Anti-Unification for Unranked Terms and Hedges. *Journal of Automated Reasoning* 52, 2 (2014).
[13] Mull Project. 2026. Mull: Practical mutation testing and fault injection for C and C++. https://mull-project.com/
[14] Martin Pool. 2026. cargo-mutants: Mutation testing for Rust. https://mutants.rs/
[15] Stryker Mutator. 2026. Stryker Mutator. https://stryker-mutator.io/
[16] Michele Tufano. 2022. *Bugs2Fix*. doi:10.5281/zenodo.7482755 Dataset of bugs and fixes extracted from GitHub.