

LLM-Based Repair of Static Nullability Errors

NIMA KARIMIPOUR, University of California at Riverside, USA
PASCAL JOOS, CISPA Helmholtz Center for Information Security, Germany
MICHAEL PRADEL, CISPA Helmholtz Center for Information Security, Germany
MARTIN KELLOGG, New Jersey Institute of Technology, USA
MANU SRIDHARAN, University of California at Riverside, USA

Modern Java projects increasingly adopt static analysis tools that prevent null-pointer exceptions by treating nullness as a type property. However, integrating such tools into large, existing codebases remains a significant challenge. While annotation inference can eliminate many errors automatically, a subset of residual errors—typically a mix of real bugs and false positives—often persists and can only be resolved via code changes. Manually addressing these errors is tedious and error-prone. Large language models (LLMs) offer a promising path toward automating these repairs, but naïvely prompted LLMs often generate incorrect, contextually inappropriate edits. We present NullRepair, a system that integrates LLMs into a structured workflow for resolving the errors from a nullability checker. NullRepair's decision process follows a flowchart derived from manual analysis of 200 real-world errors. It leverages static analysis to identify *safe* and *unsafe* usage regions of symbols, using error-free usage examples to contextualize model prompts. Patches are generated through an iterative interaction with the LLM that incorporates project-wide context and decision logic. Our evaluation on 12 real-world Java projects shows that NullRepair resolves 63% of the 1,119 nullability errors that remain after applying a state-of-the-art annotation inference technique. Unlike two baselines (single-shot prompt and mini-SWE-agent), NullRepair also largely preserves program semantics, with all unit tests passing in 10/12 projects after applying every edit proposed by NullRepair, and 98% or more tests passing in the remaining two projects.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software verification and validation**; **Automatic programming**.

Additional Key Words and Phrases: Static nullness checkers, static analysis, large language models

ACM Reference Format:
Nima Karimipour, Pascal Joos, Michael Pradel, Martin Kellogg, and Manu Sridharan. 2026. LLM-Based Repair of Static Nullability Errors. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA023 (October 2026), 24 pages. https://doi.org/10.1145/3832114

1 Introduction

As null pointer exceptions (NPEs) are among the most common sources of run-time errors in Java, substantial research has focused on preventing null dereferences [23, 44, 46, 53]. In production environments, NPEs can lead to unexpected crashes and system downtime. Preventing NPEs as early as possible—ideally at compile time—is therefore essential for building reliable software. Static nullness checkers that treat nullability as a type property have emerged as effective tools

Authors' Contact Information: Nima Karimipour, University of California at Riverside, Computer Science & Engineering Dept, Riverside, USA, nima.karimipour@email.ucr.edu; Pascal Joos, CISPA Helmholtz Center for Information Security, SoftwareLab, Saarbrücken, Germany, pascal.joos@outlook.de; Michael Pradel, CISPA Helmholtz Center for Information Security, SoftwareLab, Saarbrücken, Germany, michael@binaervarianz.de; Martin Kellogg, New Jersey Institute of Technology, Computer Science, Newark, USA, martin.kellogg@njit.edu; Manu Sridharan, University of California at Riverside, Computer Science & Engineering Dept, Riverside, USA, manu@cs.ucr.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA023
https://doi.org/10.1145/3832114

for detecting potential NPEs before deployment. Examples include pluggable type checkers like Uber's NullAway [4], Meta's Nullsafe [55], and the Checker Framework's Nullness Checker [14, 53]. These tools offer lightweight integration into existing development workflows, and with JSpecify standardization of a semantics for nullness annotations [3, 49], there is an increasing demand for these tools.

However, adopting such checkers in existing codebases is a significant challenge: for precise and sound checking, the checkers require accurate nullability annotations. To reduce this manual effort, recent work [15, 33, 35, 59] proposes *inference of nullability annotations*, which takes a program without nullability annotations and adds annotations automatically. However, annotations alone are often insufficient. A recent study [31] analyzes versions of programs before and after modules are enrolled into nullness type checkers. The study concludes that beyond adding annotations, developers often perform additional code logic changes as part of the enrollment process. Yet, annotation inference tools only add annotations or suppressions, and do not modify the program logic. Hence, fully satisfying a static nullness checker requires further code changes. Across our benchmark, we find that after applying a state-of-the-art annotation inference technique, 24.8% of the initial nullability errors remain. Thus, fixing these residual nullability errors – if done manually – significantly hinders the adoption of nullness checkers.

This work presents NullRepair, the first automated system for repairing residual type-based nullability errors that annotations alone cannot resolve. Our approach is designed to assist the adoption of static type-based nullness checkers. NullRepair runs *after* existing annotation inference tools, addressing the remaining errors that those tools leave unresolved. We implement and evaluate NullRepair in the context of NullAway [4] and its inference engine [33], showing that it complements state-of-the-art [31] inference techniques by handling hard-to-fix cases through automated code editing. NullRepair guides its repair process using the structure of nullability errors and program semantics. These traditional data sources are augmented with a large language model (LLM), which NullRepair relies on for semantic reasoning and patch generation.

While LLMs have shown strong capabilities for repair [9, 11, 66], we observe that resolving residual nullness checker errors by naïvely prompting an LLM or using general-purpose software engineering agents [67] is insufficient in practice: the LLM often takes the path of least resistance to satisfy the static checker, without resolving the underlying problem. As a result, LLM-generated patches are often "plausible but incorrect" [57] in the parlance of automated program repair: they repair the surface problem (the error from the static checker) but break something else in the process. Nullability is a pervasive and context-sensitive issue: establishing the provenance of a nullable value and reasoning about its appropriate handling may require navigating across multiple classes. In large codebases, assembling the right context for the model is nontrivial. To better understand this challenge, we conduct a preliminary study of 200 residual errors from NullAway after running annotation inference and construct a flowchart that encodes a human's reasoning process for addressing each error. NullRepair uses this flowchart to guide its queries to the LLM.

Our key insight for constructing effective LLM queries is to decompose the codebase into *usage regions*, indexed by the symbols involved in a nullability error. Given these regions, the approach identifies *safe usage patterns* of these symbols across the codebase and uses them to synthesize patches that are both contextually grounded and project-consistent. This novel strategy, of identifying safe usage patterns based on nullability error information, allows us to inject examples directly from the codebase into the LLM prompts, providing the necessary context to identify a suitable fix. Our idea is motivated by the plastic surgery hypothesis [5], which states that fixes can usually be constructed based on existing code-fragments in the same codebase.

Evaluating NullRepair on twelve real-world, previously-unannotated Java projects shows that NullRepair resolves 63% of the 1,119 residual nullability errors that remain after applying a state-of-the-art annotation inference technique. Importantly, NullRepair's patches largely preserve semantics, with all unit tests (held-out from NullRepair) passing in 10/12 projects after applying every edit proposed by NullRepair, and 98% or more tests passing in the remaining two projects. In contrast, two baselines (single-shot prompt and mini-SWE-agent [41]) remove more errors, 72% and 78%, but cause test failures in most (8/12) projects. Further, a manual assessment of patch quality shows that NullRepair's patches are more likely to be acceptable to developers without changes (61-81% more likely) and less likely to be completely unacceptable (23-33% less likely). NullRepair is comparable to the agentic baseline in speed and cost: 1.2 minutes and 15,000 tokens ($0.05 as of current pricing) per error—and *much* faster and cheaper than manual repair. Overall, NullRepair resolves fewer static nullability errors, but has a dramatically lower rate of "plausible but incorrect" patches than the baselines, making it much more palatable for automated deployment without direct human supervision.

In summary, this paper's contributions are:

- We conduct a study of 200 nullability errors that remain after annotation inference and distill a decision flowchart that guides patch generation.
- We introduce NullRepair, an LLM-based and static analysis-based patching system that resolves residual nullability errors that cannot be addressed through annotations alone.
- We evaluate our approach on 1,119 nullability errors from 12 real-world Java projects, showing that it resolves 63% of the errors, while largely preserving correctness and offering mostly human-acceptable patches.
- We release the dataset, implementation, and detailed manual assessments as part of our artifact for reproducibility (Section 10).

2 Background: Java Nullability Checkers

Type-based nullness checkers [4, 14, 53, 55] rely on *type qualifiers* [18] to refine the type system by expressing whether a reference type may include `null`. These qualifiers are written in Java using annotations: `@Nullable` indicates that a reference may be `null`, and `@NonNull` indicates that it must not be. By requiring these qualifiers on parameter, return, and field types, these checkers are fully modular, i.e., each method can be checked independently, resulting in faster type checking. Types without an explicit qualifier are assumed to be `@NonNull` by default, except for local variables whose qualifiers may be inferred automatically [53].

Checkers typically enforce two key rules: (1) a `@Nullable` expression must not be dereferenced without a preceding null check, and (2) a `@Nullable` value must not be assigned to a location declared as `@NonNull`. These rules are enforced through a subtyping relation in which `@NonNull T` is a subtype of `@Nullable T`. Assuming that object fields are properly initialized and all code has been type checked, these constraints collectively ensure that the program is free of NPEs.

For example, consider Figure 1a. Method `bar(Foo)` is called with a `null` argument (line 6), which violates the nullness checker's rules, as the parameter `param` is `@NonNull` by default. The checker would report an error at the call site, indicating that a `@Nullable` value is being passed to a method that expects a `@NonNull` parameter. To resolve this, the developer must either ensure that the argument is non-null or modify the method signature to accept `@Nullable` values as done in Figure 1b. With the changes applied, no type rule is violated, and the nullness checker confirms that no potential null dereferences remain.

While our conceptual contributions could be applied to any Java nullability checker, we build NullRepair on top of NullAway [4], a nullability checker developed at Uber, due to its widespread

```
1  public void bar(Foo param) {
2
3    param.deref();
4
5  }
6  bar(null);  // checker error
```

(a) Nullability error

```
1   public void bar(+@Nullable Foo param) {
2  +  if (param != null) {
3       param.deref();
4  +  }
5   }
6   bar(null);
```

(b) Error resolved

Fig. 1. Example of a nullability error and its resolution.

```
1   package model;
2   class User {
3  +  @Nullable MapView mapView;
4  +  @Nullable public MapView getMapView() {
5       return this.mapView;
6     }
7   }
8
9   package controller;
10  class Controller {
11    public void updateLocation(User user) {
12 -    move(user.getMapView().getCenter()); // Error
13 +    Point2D point = user.getMapView() == null  ?
14 +         new Point2D.Double(-1, -1)   :
15 +         user.getMapView().getCenter();
16 +    move(point);
17    }
18  }
```

```
19  package ui.dashboard;
20  class Dashboard {
21    private User user = new User();
22    public Point2D getCenter() {
23      if (user.getMapView() == null) {
24          return new Point2D.Double(-1, -1);
25      }
26      return user.getMapView().getCenter();
27    }
28  }
```

Fig. 2. Code example to be enrolled into a nullability checker with a residual nullability error (line 12). Annotations in blue are added by an annotation inference tool like Annotator [33], and are part of NullRepair's input. NullRepair proposes the change that removes the red line and replaces it with the green lines (lines 13-16). Lines 23-26 show a safe usage pattern of `user.getMapView()`.

adoption. NullAway has a companion annotation inference tool, Annotator [33], which we use to generate an initial set of annotations for a target program, before invoking NullRepair. We call any nullability error that remains after the automated annotation inference a *residual nullability error*, or for brevity simply *nullability error*.

3 Motivating Example

Figure 2 illustrates the challenges of resolving nullability errors and how our approach addresses these challenges. The goal is to enroll this code in static nullability checking with no remaining errors. The nullability checker reports an error that field `mapView` is assumed to be non-null by default but is not initialized in the constructor. To address this error, automated annotation inference adds `@Nullable` annotations on the field and its getter, which resolves the original errors. However, the getter annotation causes a new error at line 12 due to the dereference of `user.getMapView()`. Since this dereference error cannot be resolved by adding annotations, automated annotation inference cannot repair it: deeper reasoning and non-trivial edits are required.

A null dereference error can be addressed in several ways, but selecting the best fix is often non-trivial. While inserting a null check is a common and desirable approach, determining the correct behavior for the *then* branch—i.e., when the value is null—is particularly challenging. Synthesizing

this behavior often requires domain-specific knowledge to ensure that the resulting patch matches the intended semantics of the program. Naïvely prompting an LLM to fix this error yields a patch that, when `user.getMapView()` is null, simply skips the call to `getCenter()` and returns. However, this patch is not semantically aligned with how the developers handle such a null value in other parts of the codebase.

Instead, for the dereference of `user.getMapView()` on line 12, NullRepair identifies a usage pattern in the `Dashboard` class where the same method is accessed safely via an explicit null check at line 23. This safe usage serves as a guiding example, enabling NullRepair to synthesize a patch that eliminates the dereference error while maintaining semantic fidelity. For this case, NullRepair determines that inserting a null check around the dereference, and for the *then* branch, materializing a default value, is the most appropriate behavior. Among several possible defaults, NullRepair synthesizes a value that aligns with the program's conventions, in this case returning a `Point2D` object at coordinates `(-1, -1)`.

This fix is possible because NullRepair can extract and index both positive (safe) and negative (unsafe) usage patterns for a given code element across the entire codebase. By tracking not only where a symbol is misused (e.g., dereferenced without a check) but also where it is handled safely, NullRepair can retrieve usage-consistent examples to guide the model's synthesis. These usage regions are not constrained to a fixed context window and may span multiple files, classes, or modules. As a result, NullRepair can surface semantically meaningful context even when relevant usages are far from the error site, helping the model generate fixes that are both context-aware and aligned with the project's conventions.

4 Preliminary Study

To better understand the resolution strategies required to resolve residual nullability errors, we conduct a manual analysis of 200 such errors drawn from the three benchmarks that show the least reduction in the number of errors after inference in the original Annotator study [33]. We choose these benchmarks because they contain a significant number of errors that cannot be resolved through annotations alone. While we include these three benchmarks (retrofit, conductor, and litiengine) in our evaluation, excluding them has no impact on our overall conclusions (Section 8).

This preliminary study aims to uncover the types of information required to resolve each error, the decisions a human or automated tool must make, and the code transformations needed to produce a valid patch. For each nullability error reported on the benchmarks after running annotation inference, we manually resolve it, tracing and documenting the following decisions:

- The specific question that needs to be answered.
- The expected form of the answer.
- The resulting action required based on the answer.

We build a decision flowchart using this information incrementally. After analyzing each new error, we check whether our current flowchart accounts for each decision and action involved. If not, we expand the flowchart with new branches or refinements to reflect that error's resolution path. Over time, this iterative refinement converges to the decision flowchart shown in Figure 3, which captures all the decision points and actions needed to address the variety in these 200 nullability errors.

This manual study is the empirical foundation for our approach: it grounds the flowchart in real-world error scenarios so that it can guide the repair process across diverse code patterns and error types. The next section describes how NullRepair implements this strategy to resolve nullability errors through a series of automated decisions and transformations.

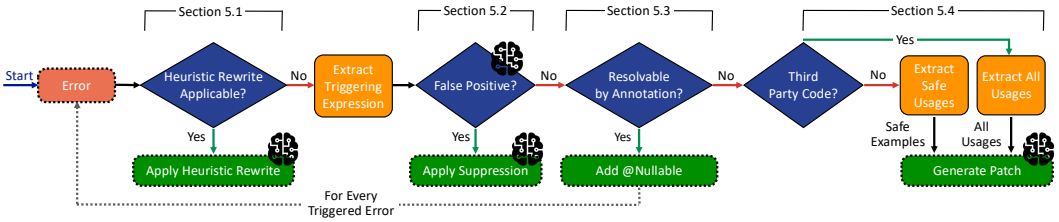


Fig. 3. Decision flowchart guiding NullRepair's patch synthesis. Blue nodes denote decision points, orange nodes indicate action steps, green nodes are terminals where patches are generated.

5 Approach

Our goal is to automatically generate high-quality patches that eliminate nullability errors that remain after annotation inference, by imitating how a developer investigates and resolves such errors. To structure this process, we use the flowchart-based decision protocol (Figure 3) constructed as described in Section 4. Each error reported by the nullability checker has these components:

- **Triggering expression:** the specific expression (e.g., local variable) whose nullability triggered the error.
- **Program point:** the code location (e.g., dereference site) where the error is reported.
- **Error type:** the category of nullability violation (e.g., dereference or return mismatch).
- **Error message:** the human-readable description of the error.
- **Suggested annotation (optional):** an annotation that would resolve the error, if there is one.[1]

The system launches an instance of NullRepair for each error, each of which navigates the Figure 3 flowchart node by node. At each step, NullRepair takes one of three actions: querying an LLM, running static analyses, or applying code transformations. This guided traversal continues until it reaches a terminal node and synthesizes a candidate fix. This protocol-driven interaction is based on the intuition that incorporating surrounding code context and usage patterns can increase the likelihood that generated patches align with the program's intended behavior. The following sections describe the flowchart in detail and explain how NullRepair uses it to synthesize patches.

5.1 Heuristic Code Rewriting

First, NullRepair checks whether a predefined heuristic null-safe rewrite pattern applies at the error site. Each heuristic consists of a precondition and a corresponding rewrite rule, designed to improve robustness against null dereferences while preserving (or minimally impacting) program semantics. Table 1 summarizes NullRepair's rewrite patterns and their associated conditions. If a matching rewrite pattern is applicable, the corresponding transformation is applied immediately. Otherwise, NullRepair proceeds to the next decision node in the flowchart.

For example, when a receiver `x` of a call to `equals` is `@Nullable` (precondition), NullRepair replaces expression `x.equals(y)` with `Objects.equals(x,y)` (rewrite). This avoids a null dereference while preserving behavior when `x` is `@NonNull`. As another example, if a `@Nullable` dereference `x.deref()` occurs in a method with a `@Nullable` return type, and the nullness checker reports no errors at any call site due to the method returning `@Nullable` (precondition), NullRepair inserts an early `return null;` immediately after a null check on `x` (rewrite). Although this transformation alters the method's control flow, it respects the method's nullability contract and leverages existing null-handling at call sites to prevent a potential crash.

[1]Some inference techniques, including Annotator [33], heuristically omit annotations that trigger too many new errors.

Table 1. Null-safe heuristic rewrites with preconditions that ensure semantic safety.

Before	After	Precondition
`x.equals(y)`	`Objects.equals(x, y)`	`x` is `@Nullable`
`x.hashCode()`	`Objects.hashCode(x)`	`x` is `@Nullable`
`x.toString()`	`String.valueOf(x)`	`x` is `@Nullable`
`x.deref()`	`if (x == null) return null; x.deref();`	`x` is `@Nullable`, enclosing method is `@Nullable`, safe call sites

We perform heuristic code rewriting *before* false positive suppression (Section 5.2) for two reasons. First, the transformations given in Table 1 are desirable regardless of whether the error is a false positive, as they provably eliminate the NPE risk, preserve semantics when the variable is not null, and behave appropriately when the variable is null. For example, given a nullability error for `x.equals(y)`, rewriting to `Objects.equals(x, y)` preserves semantics when `x` is non-null and performs the desired comparison if `x` is null. Second, our FP suppression leverages an LLM, and hence there is always some risk of inadvertently suppressing a true-positive report; applying the transformations of Table 1 has nearly no risk.

5.2 Suppressing False Positives

Our preliminary study shows several cases where the checker incorrectly flags an expression as being possibly null, leading to a false positive error. These misclassifications often arise due to limitations in the checker's reasoning—for example, failing to recognize guarding conditions, interprocedural guarantees, or method contracts that ensure non-nullness. In such cases, modifying the code to satisfy the checker may require intrusive or large-scale changes that are unlikely to be acceptable to developers. Therefore, we consider error suppressions (e.g., using casts or annotations) to be a practical and desirable strategy for resolving such errors. Notably, we observe that suppressions and casts are commonly used in existing enrolled codebases to silence similar errors, which confirms the findings of a recent study [24]. Consequently, NullRepair attempts to detect such false positives and applies suppressions when appropriate.

Detecting false positives requires semantic understanding of the code and its context, difficult to capture with deterministic rules. We therefore prompt an LLM, which has shown to be effective at such detection tasks [16, 39, 40]. NullRepair extracts the method containing the reported error and constructs a specialized decision prompt to assess whether the triggering expression is guaranteed to be non-null at the error location. Each decision prompt follows a template that instructs the model to either (i) respond with a definitive answer when confident, or (ii) indicate uncertainty and request additional context. The model's reply is expected to follow a structured XML format. The full structure of these prompts is described in Section 5.5.

Initially, NullRepair includes only the method body in the prompt, as our preliminary study shows that this scope is often sufficient to identify false positives in practice, and eagerly including broader context significantly increases prompt size. If the model replies with uncertainty, it may request specific additional information, such as related method bodies, which NullRepair retrieves and incorporates into a follow-up prompt. If the model response indicates that the reported error is a false positive, NullRepair inserts an explicit non-null cast at the error site, suppressing the error without modifying the program's behavior.

Consider the example in Figure 4. In this case, the model receives a prompt containing the body of `handle()`, where the nullable field `session` is dereferenced. The model replies with uncertainty and requests the implementation of `check()`. If the model infers that a null check within `check()`

```
1   @Nullable Session session;
2   void handle() {
3     if(check()) {
4   -   session.refresh();
5   +   Nullability.castToNonnull(session).refresh();
6     }
7
8   boolean check() {
9     return session != null;
10  }
```

Fig. 4. Example of a false positive suppression.

guarantees `session` is non-null at the dereference site, it classifies the error as a false positive, leading to the inserted cast.[2]

5.3 Handling Triggered Errors

If the error is determined to be a true positive, NullRepair checks whether inserting an annotation can resolve it—which happens when the checker reports a `@Nullable` value being assigned to a `@NonNull` location. In such cases, the fix is to add a `@Nullable` annotation to the destination. After adding the annotation, NullRepair reruns the type checker to identify new errors triggered by this change. Each newly triggered error is then fed back into the flowchart from the beginning and processed independently. To avoid overlapping rewrites and ensure consistency across executions of the flowchart, NullRepair maintains an up-to-date working copy of all previously rewritten regions. The final result is the union of all the changes computed across the original error and any subsequently triggered errors. Since handling triggering errors can, in principle, lead to an infinite sequence of new errors, NullRepair has a budget for resolution attempts (see Section 5.5). This process differs from that followed by the previously run annotation inference tool, since annotation inference only adds annotations that reduce the total number of errors, while NullRepair can address follow-on errors by applying further code changes.

5.4 Safe and Unsafe Usage Regions

In our preliminary study, human developers first exhaust the previously-described techniques—heuristic rewrites, checking for false positives, and adding annotations—before resorting to larger code changes. Following this strategy, only if these techniques fail, NullRepair moves to the next decision node in the flowchart, where it synthesizes a code change by learning from usage patterns in the codebase. The key idea in this step is leveraging the contrast between *safe* and *unsafe* usage patterns of the triggering expression, i.e., code locations where the same element is used without or with triggering a checker error, respectively.

To find these patterns, NullRepair partitions the codebase into discrete *regions*, defined as method bodies and field declarations, which serve as the semantic units within which symbols appear. We formalize this abstraction as a function: $usageRegions : Symbol \rightarrow \{Method\ or\ Field\ Declaration\}$. This level of region granularity in practice preserves enough semantic information to guide the model with contextual clues about program intent. During repair, NullRepair maintains an index that maps each symbol to its corresponding usage regions, and for each region, indicates whether the usage triggers a checker error due to usage of that symbol. Using this index, the system can efficiently retrieve representative *safe* and *unsafe* examples across the project and then provides all

[2]See https://github.com/uber/NullAway/wiki/Suppressing-Warnings#downcasting for more information on NullAway casts.

safe examples as part of the prompt for generating a patch. In the rare case where no safe usages are found for a triggering expression, NullRepair falls back to returning any usages.

To our knowledge, NullRepair is the first repair approach to incorporate statically derived safe and unsafe usage contexts into a structured decision protocol for LLM-driven patch generation. This hybrid strategy increases contextual relevance while reducing overfitting to local code structure.

When retrieving usage examples to guide synthesis, NullRepair distinguishes two cases based on the origin of the triggering expression. If the expression refers to a symbol declared in the codebase, NullRepair directly retrieves safe and unsafe usages of that symbol across all regions. However, if the expression originates from third-party code, NullRepair locates the closest receiver within the expression that *is* declared in the codebase. It then collects all regions where this receiver is used, as these reveal the receiver's lifecycle and usage invariants. For instance, if the triggering expression is `map.get(key)` and `map` is a project-declared field, NullRepair does not analyze all `Map.get()` usages globally. Instead, it focuses on how *this specific* `map` instance is used.

As an example, in line 12 of Figure 2, `user.getMapView()` causes a null dereference error that cannot be resolved by annotations. NullRepair retrieves all usages of `getMapView()` across the project. It discovers that in the `Dashboard` class, the `mapView` field is checked for null before access (line 23) and a reasonable default is returned in the null case. This safe usage is included in the model prompt to guide synthesis. The model then produces a patch that mirrors this safe pattern by inserting a null check and similar default value for the line 12 dereference.

5.5 LLM Setup

Each NullRepair query is stateless and constructed independently, relying only on the context accumulated up to that point. Queries fall into two categories: *decision prompts* and *patch prompts*.

5.5.1 Decision Prompts. Decision prompts assess whether a specific error is a false positive, e.g., whether a method cannot return `null`, to then traverse the flowchart. The model is expected to respond with either `Yes` or `No` to indicate agreement or disagreement, or `Unsure` when it cannot confidently answer. In the `Unsure` case, the model may request additional context—such as the method declaration and body, field declaration, or relevant usages—to make a more informed decision. Decision prompts are only used to decide if errors are false positives. The other decisions in the flowchart are decided based on static analyses, without involving the LLM.

5.5.2 Patch Prompts. Patch prompts request concrete edits to fix a specific error, given contextual evidence. These prompts typically include the triggering expression, safe usage examples extracted from the codebase, and the relevant enclosing method or field. The model is asked to return a rewritten version of the method that resolves the error. Each patch response is validated to ensure that the output is syntactically valid Java code. While this does not guarantee successful compilation, it ensures that the output can be parsed and further analyzed by downstream tools.

5.5.3 Prompt Design and Validation. To construct robust decision prompts, we begin with a labeled dataset of nullability errors drawn from the same benchmarks used in Section 4. Each error is paired with a classification question and a ground truth answer obtained through manual inspection. We first draft prompts and then use the LLM itself to refine them via meta-queries, similar to previous work that optimizes prompts using LLMs [19]. The meta-queries ask the model to suggest clearer phrasing, remove ambiguity, or better align with its reasoning patterns. This self-refinement loop helps make the final prompt templates interpretable and robust across contexts.

Each candidate prompt is tested on at least 20 examples from the benchmarks from Section 4. To minimize overfitting, prompt tuning is limited to this subset. A prompt is accepted only if the model's responses consistently match the expected answer across all test cases. All prompt

Prompt Example (Shortened)

```
### TARGET NULLABILITY ERROR
In line move(user.getMapView().getCenter()); the expression
getMapView() is nullable, causing a null dereference error.
Method:
public void updateLocation(User user) {
  move(user.getMapView().getCenter());
}

### EXAMPLE
See a safe usage example of the method getMapView():
public Point2D getCenter() {
  if (user.getMapView() == null) {
    return new Point2D.Double(-1, -1);
  }
  return user.getMapView().getCenter();
}

### TASK
Based on the example provided above, please rewrite the
method to resolve the error.

### GUIDELINE, RESPONSE FORMAT, OUTPUT EXAMPLE
(Omitted for brevity)
```

Model Response

```
<response>
  <success>
    True
  </success>
  <code>
    public void updateLocation(User user) {
      Point2D point = user.getMapView() == null
        ? new Point2D.Double(-1, -1)
        : user.getMapView().getCenter();
      move(point);
    }
  </code>
</response>
```

Fig. 5. Example of a prompt for patch generation, with the model response. Blue text captures program elements extracted from the codebase; the remaining text is part of the static prompt template.

responses must follow a fixed XML schema, and are parsed and validated; if an output fails structural validation, a repair loop is triggered. This loop supplies the model with the malformed output and parse error, prompting it to retry, up to five times per query. Each error is allocated a fixed interaction budget: a maximum of 50 total LLM queries, with local retry thresholds. In practice, this budget ensures scalability while preserving sufficient reasoning depth for complex cases.

Figure 5 shows an example of NullRepair's prompting strategy. It illustrates a *patch prompt*, where NullRepair asks the model to generate a fix using contextual cues.

6 Implementation

Our implementation is built on top of the Annotator tool [33], which infers annotations to ease the adoption of nullability checkers. Annotator provides an extensible indexing framework over the input codebase, enabling efficient querying of program elements and their usage contexts. We use the JavaParser [61] library to parse Java source code and apply synthesized patches.

Region and Symbol Indexing. Annotator constructs a global index that maps each triggered nullability error to the corresponding *triggering expression*, the underlying *symbol*, and the *code regions* where that symbol appears. We extend this index to further classify regions into *safe* and *unsafe* usage regions, depending on whether they do or do not trigger errors (Section 5.4).

7 Evaluation

7.1 Experimental Setup

Benchmarks. In our evaluation, we use the same benchmarks used in the Annotator paper [33]. These benchmarks span diverse software categories and have not been previously checked with NullAway, making them representative of the usage scenarios NullRepair targets. Table 2 reports the number of nullability errors remaining in each benchmark after applying annotation inference. These residual errors represent the lower bound of what inference alone cannot resolve, highlighting

Table 2. Comparison of NullRepair and the baselines across two modes. **KLoC** is the benchmark size in thousands of lines of code and **Errors** is the number of residual errors in the benchmark. In the **Per Patch** section, **CE** is the number of generated patches with compilation errors, **NE** is the number of compiling patches that trigger at least one new nullability error, and **R** is the number of patches that fully resolve the target error (not triggering new nullability or compilation errors). In the **Combined** section, **R** is the total number of resolved target errors when selectively applying only patches that resolve their error without triggering any new unresolved errors.

Project	KLoC	Errors	Per Patch									Combined		
			NullRepair			mini-SWE-agent			SinglePrompt			NullRepair	mini-SWE-agent	SinglePrompt
			CE	NE	R	CE	NE	R	CE	NE	R	R	R	R
conductor	13.8	25	1	3	17 (68%)	0	2	22 (88%)	1	1	23 (92%)	16 (64%)	23 (92%)	23 (92%)
eureka	9.1	24	0	3	21 (88%)	1	2	21 (88%)	1	4	18 (75%)	21 (88%)	22 (92%)	19 (79%)
glide	27.3	87	7	15	60 (69%)	6	11	65 (75%)	2	14	68 (78%)	63 (72%)	80 (92%)	69 (79%)
gson	8.6	32	0	6	17 (53%)	1	1	28 (88%)	1	1	26 (81%)	18 (56%)	30 (94%)	25 (78%)
jadx	49.7	117	14	35	58 (50%)	14	19	80 (68%)	16	14	82 (70%)	57 (49%)	86 (74%)	82 (70%)
libgdx	97.3	453	14	75	286 (63%)	23	71	343 (76%)	56	71	293 (65%)	301 (66%)	354 (78%)	310 (68%)
litiengine	32.3	152	17	34	95 (63%)	5	23	117 (77%)	17	12	116 (76%)	86 (57%)	89 (59%)	119 (78%)
mockito	19.3	36	5	10	21 (58%)	0	4	28 (78%)	3	6	21 (58%)	19 (53%)	26 (72%)	20 (56%)
retrofit	3.6	13	0	3	10 (77%)	0	0	13 (100%)	0	0	13 (100%)	13 (100%)	12 (92%)	13 (100%)
spring-boot	39.9	84	2	18	61 (73%)	6	13	60 (71%)	11	17	53 (63%)	60 (71%)	69 (82%)	54 (64%)
wala-util	19.7	73	2	16	31 (42%)	6	6	57 (78%)	7	10	52 (71%)	31 (42%)	67 (92%)	56 (77%)
zuul	17.4	23	1	3	19 (83%)	2	2	19 (83%)	5	4	12 (52%)	19 (83%)	20 (87%)	12 (52%)
Total	338.0	1,119	63	221	696 (62%)	64	154	853 (76%)	120	154	777 (69%)	704 (63%)	878 (78%)	802 (72%)

the remaining challenge. The table also includes the number of non-comment, non-whitespace lines of code for each benchmark to indicate the scale of each project.

Baselines. We compare NullRepair to two baselines, SinglePrompt and mini-SWE-agent. *SinglePrompt* prompts the LLM once per error, where the prompt contains the nullability error along with the surrounding code. We use 20 lines before and after the error location for surrounding code, or the entire enclosing method if it is smaller. *mini-SWE-agent* follows an agentic approach [41, 67], and is one of the top-performing SE-agents on the SWE-bench Verified benchmark [7]. The agent can freely explore the project to gather context, modify arbitrary files, and validate changes by running the build and NullAway. We provide the agent with the target nullability error and instructions on how it can apply and validate changes. mini-SWE-agent uses shell commands to interact with the project and can thus perform arbitrary actions.

LLM Configuration. We use `GPT-5.1` from OpenAI without reasoning for all experiments [52]. Both mini-SWE-agent and NullRepair have the same maximum budget of 50 distinct prompts and $0.50 per error.

Research Questions. Ideally, we would expect a static nullness error repair tool to take a codebase with nullability errors and generate code patches that resolve all such errors with minimal code changes, in a way that aligns with the program's logic and structure, resulting in a compilable program with no remaining errors quickly and at a reasonable cost. We evaluate two research questions to judge how well NullRepair performs with respect to this ideal:

- **RQ1:** How effective and high-quality are the patches produced by NullRepair?
- **RQ2:** What are the end-to-end execution time and API cost of applying NullRepair?

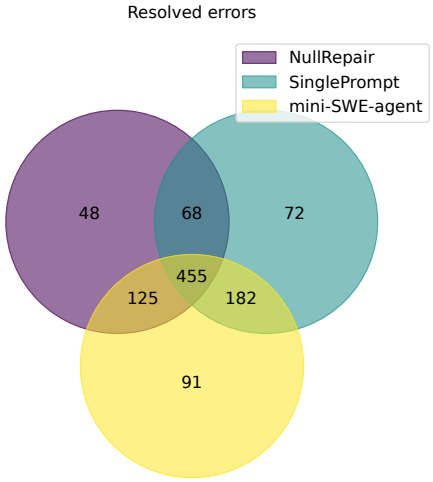


Fig. 6. Venn diagram showing the intersection of patches generated by NullRepair, mini-SWE-agent, and SinglePrompt that fully resolve the target error (not triggering new nullability or compilation errors).

7.2 RQ1: Effectiveness and Quality of Patches

We aim to determine whether the patches generated by NullRepair are acceptable to developers. Determining whether a fix is truly acceptable is a multidimensional question. There are often multiple valid ways to resolve a nullability error, depending on factors such as coding conventions, design intent, and local control flow structure. Rather than relying on a single fixed criterion to judge patch quality, we evaluate effectiveness through a series of criteria each capturing a different aspect of what makes patches useful and consistent with developer expectations.

- **C1: Static Correctness:** Do patches compile successfully and resolve the targeted nullability error without introducing new unresolved errors?
- **C2: Dynamic Correctness:** Do patches preserve the original program behavior, as measured by the success rate of existing test suites?
- **C3: Manual Assessment:** Are patches logically aligned with the surrounding code and likely to be acceptable to a human developer in a code review?

We measure these criteria along two complementary axes: (1) a *patch-level analysis* that evaluates whether each individual patch compiles and resolves its specific error, and (2) an *aggregate-level analysis* that measures the total number of resolved nullability errors after selectively applying only the patches that both compile and fully resolve their targeted error. This simulates a realistic adoption scenario where only statically sound patches are retained. For each residual nullability error, we attempt to generate a patch using both the baselines and NullRepair. We then re-run NullAway on the patched code to re-index any remaining or newly introduced errors.

C1: Static Correctness. Table 2 (*Per Patch*) presents the patch-level results, reporting the number of non-compiling patches each approach generates (*CE*), the number of compiling patches that introduce new nullability errors (*NE*), and those that fully resolve the original nullability error without introducing new errors (*R*). We find that NullRepair generates significantly fewer non-compiling patches compared to SinglePrompt. On average, for only 5.63% of errors NullRepair

```
  @SuppressWarnings("unchecked")
  public <T extends Channel> T getChannel(ChannelDescriptor descriptor) {
    for (Channel array : arrays) {
      if (array.id == descriptor.id) return (T) array;
    }
-   return null; // Error, as getChannel(ChannelDescriptor) is @NonNull
+   throw new IllegalStateException("Channel not found for descriptor id: " + descriptor.id);
  }
```

Fig. 7. Example of incorrect suppression by baselines, breaking program functionality at a call-site.

produces patches that fail to compile, compared to 10.72% for SinglePrompt. This improvement is largely due to NullRepair's use of contextual cues, such as method usage and inferred constraints, which guide patch generation toward semantically valid code. mini-SWE-agent has a similar compilation failure rate (5.72%) to NullRepair, as it can iteratively validate patches and refine them based on compilation feedback. In terms of effectiveness, NullRepair successfully resolves 62% of errors on average (range: 42–88%), SinglePrompt achieves 69% (range: 52–100%) and mini-SWE-agent reaches 76% (range: 71–100%), outperforming NullRepair by 14 percentage points. Figure 6 shows the overlap between the sets of errors resolved by each approach. While NullRepair resolves a significant portion of errors, there is a substantial subset that only the baselines resolve. However, we observe that both baselines frequently rely on overly suppressive local edits that do not fix the underlying problem, which may obscure the actual quality of these fixes. The rate of newly introduced nullability errors also reflects this: NullRepair triggers more (19.75%) than the baselines (both 13.76%). This trade-off suggests that NullRepair often attempts more ambitious edits that engage deeper semantic issues, whereas the baselines prefer minimal, often suppressive changes.

Table 2 (*Combined*) presents the aggregate-level evaluation by measuring the total number of resolved nullability errors after selectively applying only those generated patches that resolve the respective target error. For this, NullRepair iteratively processes errors and applies patches sequentially, only retaining those that reduce the total number of errors. Please note that these are not necessarily the same patches as generated for *Per Patch*, as the codebase changes with each applied patch, such that subsequent patches are generated in a different context, potentially making the task easier or harder. In *Combined* the baselines achieve a higher overall percentage of resolved errors (SinglePrompt 72% and mini-SWE-agent 78% in aggregate) compared to NullRepair (63% in aggregate).

However, this advantage stems primarily from the baselines' aggressive use of guard conditions and explicit throw statements that silence errors without addressing their underlying cause.

Consider the example in Figure 7. Both the SinglePrompt and mini-SWE-agent baselines forcefully silence a nullability error by explicitly throwing a run-time exception. While this suppresses the error from the checker, which correctly interprets the explicit throw as programmer intent, it breaks the code's functionality: A call-site `addChannel(...)` only creates a new channel after checking if `getChannel(ChannelDescriptor)` returns `null` and would therefore always trigger the run-time exception when trying to create a new channel. NullRepair runs out of queries attempting to resolve this error, and so does not propose a patch. In a case like this, no patch is actually better than this (incorrect!) suppression. However, in the error resolution numbers this example looks like a success for the baselines.

In other cases, we observe that the baselines often guess default values for fields that are inconsistent with the program's logic. For example, in a class `TaskModel` a `status` of `null` can

Table 3. Comparison of NullRepair and the baselines in regard to the number of failing unit tests per project after selectively applying patches that compile and resolve the target error without introducing new errors.

Project	Tests	Number of Failing Unit Tests		
		NullRepair	mini-SWE-agent	SinglePrompt
conductor	436	0 (0.0%)	38 (8.7%)	27 (6.2%)
eureka	116	1 (0.9%)	5 (4.3%)	3 (2.6%)
glide	5,335	0 (0.0%)	6 (0.1%)	42 (0.8%)
gson	1,058	0 (0.0%)	3 (0.3%)	0 (0.0%)
jadx	751	0 (0.0%)	628 (83.6%)	647 (86.2%)
libgdx	98	0 (0.0%)	0 (0.0%)	0 (0.0%)
litiengine	508	0 (0.0%)	149 (29.3%)	77 (15.2%)
mockito	2,089	34 (1.6%)	0 (0.0%)	1,136 (54.4%)
retrofit	349	0 (0.0%)	0 (0.0%)	0 (0.0%)
spring-boot	3,920	0 (0.0%)	403 (10.3%)	842 (21.5%)
wala-util	1	0 (0.0%)	0 (0.0%)	0 (0.0%)
zuul	342	0 (0.0%)	1 (0.3%)	48 (14.0%)
Total	15,003	35 (0.2%)	1,233 (8.2%)	2,822 (18.8%)

indicate an uninitialized state. The default value guessed by the baselines, `Status.SCHEDULED`, is wrong, as the `TaskModel` object is not necessarily scheduled at creation time, potentially masking errors and leading to unexpected program behavior.

Multi-file patches. In the per-patch evaluation, NullRepair creates 62 (5.5%) multi-file patches that resolve the targeted error without introducing new errors or test failures (changing up to 8 different files). SinglePrompt produces zero multi-file patches, and mini-SWE-agent produces 12 (1%) (with at most 3 files changed).

C1: NullRepair patches compile and fully resolve the target errors in 62% of cases. When combining patches, NullRepair reduces total nullability errors by 63% overall.

C2: Dynamic Correctness. To measure C2, we evaluate the patches by running each benchmark's unit tests after applying all generated patches. Similar to the aggregated setting in C1, we apply all selected patches from either NullRepair or the baselines to the benchmark code and then execute the tests. The results, shown in Table 3, reveal a striking contrast: NullRepair produces *test-clean* patches (i.e., no test failures) in all but two benchmarks, whereas both baselines introduce at least one failing test in all but four benchmarks. Note that some of the benchmarks have extensive test coverage, with over 1,000 unit tests, and still NullRepair preserves full correctness.

This result highlights a critical limitation of the baselines' patching strategies: while they may achieve a higher reduction in NullAway errors, they often do so by aggressively silencing errors (e.g., via guard statements or broad null checks) without regard for preserving program behavior. These shallow fixes may satisfy the static checker, but they fail to maintain run-time semantics, leading to test failures. In contrast, NullRepair leverages rich contextual signals, such as safe usage patterns, to synthesize semantically appropriate patches. The consistency of passing tests across a diverse set of benchmarks strongly supports the reliability and practical safety of our approach.

C2: NullRepair produces *test-clean* patches in all but two benchmarks, whereas both baselines introduce at least one failing test in all but four benchmarks.

C3: Manual Assessment. To measure C3, we conduct a manual assessment guided by a scoring protocol. Importantly, we do not claim that a patch is definitively acceptable to the original developers of each benchmark—the benchmark projects are not actually adopting NullAway at this point, so there is no incentive for their developers to dedicate time to review patches that fix NullAway errors. Instead, we define a structured rubric and scoring scheme that allows us to assess the *likelihood* of acceptability. Patches are rated as *1 - likely acceptable*, *2 - needs work*, or *3 - likely unacceptable*, based on whether they address the underlying issue correctly, preserve intended semantics, and follow idiomatic usage patterns.

We apply this evaluation to a total of 225 patches, consisting of 75 error cases for which both NullRepair and the baselines produce fixes and remove the error. Each error is associated with three anonymized candidate patches—one from each approach—which are presented side by side in randomized order. All three reviewers are senior developers and authors of this paper, but are blinded to the source of each patch. Each patch is rated by at least two reviewers to ensure reliability and reduce individual bias. After initial independent scoring, reviewers discuss any disagreements to reach consensus. The scoring criteria are:

- Does the patch correctly identify the source of nullability and address it in a justifiable way?
- Are the values or behaviors introduced by the patch consistent with the rest of the program?
- Does the patch rely on suppression, and if so, is the suppression justified?
- Does the patch preserve object protocols and initialization semantics?
- Overall, would a human developer reasonably accept this patch in a code review?

Reviewers initially disagreed on 61 pairs. After discussing differences, the reviewers were able to reach consensus on all disagreements. The Cohen's Kappa [13] value for their initial independent evaluations was 0.18, indicating low initial agreement. However, most of the disagreements (82%) only differed by one point on the three-point scale, indicating that reviewers often had similar overall impressions but varied in their threshold for acceptability.

Our results show that NullRepair produces 29 (39%) patches rated as *likely acceptable*, 16 (21%) patches rated as *needs work*, and 30 (40%) patches that are *likely unacceptable*. In total, 60% of patches could be accepted after minor revisions during code review. The average score is 2.01. In comparison, the baselines SinglePrompt and mini-SWE-agent produce only 16 and 18 (21% and 24%) *likely acceptable* patches, 14 and 18 (19% and 24%) patches that *need work*, and 45 and 39 (60% and 52%) that are *likely unacceptable*. This translates to average scores of 2.39 for SinglePrompt and 2.28 for mini-SWE-agent. Overall, NullRepair outperforms both baselines in 28 cases, ties for the best patch with at least one baseline in 26 cases, and is outperformed by *any* baseline in only 21 cases.

A detailed score distribution (see Figure 8) further illustrates these differences: while all approaches yield a similar number of patches rated as *needs work*, NullRepair produces more patches rated as *likely acceptable* (61-81% increase) and fewer rated as *likely unacceptable* (23-33% decrease) compared to the baselines. Still, all three approaches generate a substantial number of patches that reviewers deem likely unacceptable, highlighting the inherent challenges of automated nullability repair. Relying only on the static correctness via resolved errors or dynamic correctness via existing test-suites is not sufficient to guarantee high quality fixes that are acceptable to developers.

When reviewers prefer patches produced by NullRepair, it is due to their semantic precision and alignment with the program's intent. Rather than failing at the site of the error or setting incorrect default values, NullRepair often applies coordinated changes across the codebase to safely propagate null-handling logic (7/28) or recognizes false positives that it then suppresses (21/28). These patches reflect a deeper understanding of the surrounding logic and are generally perceived as more idiomatic and robust. By contrast, both baselines mostly create very local edits, do not recognize false positives, or introduce hidden bugs.

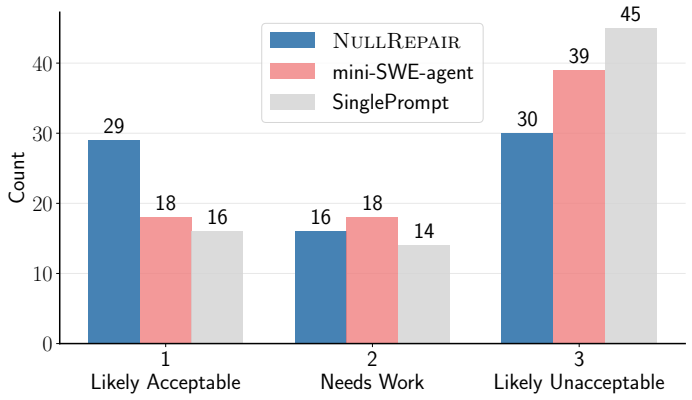


Fig. 8. The distribution of manual scores for patches generated by NullRepair and the baselines.

```
- private Class<?> resourceClass;
+ @Nullable private Class<?> resourceClass;
  ...
  List<Class<?>> getRegisteredResourceClasses() {
    return glideContext.getRegistry()
-       .getRegisteredResourceClasses(model.getClass(),
-           resourceClass, transcodeClass);
+       .getRegisteredResourceClasses(model.getClass(),
+           Nullability.castToNonnull(resourceClass), transcodeClass);
  }
  ... (further location where resourceClass is cast) ...
```

(a) Likely acceptable patch by NullRepair.

```
void clear() {
-  resourceClass = null;
   ...
}
```

(b) Likely unacceptable patch by the SinglePrompt baseline.

Fig. 9. Example where reviewers prefer NullRepair's patch over both baselines.

For example, consider the NullRepair patch in Figure 9a. NullRepair correctly identifies that the field `resourceClass` can be `null`, as a method `clear()` exists in the class that resets it to `null`. The approach therefore adds the `@Nullable` annotation and handles multiple locations where `resourceClass` must not be null, by casting them to `@NonNull` (e.g., in the method `getRegisteredResourceClasses()`). This coordinated change preserves program semantics while resolving the nullability error. In contrast, both baselines alter the behavior of the `clear()` method, by for example removing the line that sets `resourceClass` to `null` (Figure 9b), effectively breaking the resetting behavior. This is an unintended change that can lead to incorrect downstream program behavior, as well as memory leaks.

However, in some cases reviewers favor one or both of the baseline approaches. Sometimes, NullRepair creates overly complex fixes that reviewers cannot fully comprehend (3/21), or where they consider one of the baseline patches as more idiomatic (7/21). In 7 further cases, NullRepair addresses the target error correctly but introduces an unrelated change or superfluous check. In the last 4 cases, NullRepair makes an incorrect decision when determining whether the error is a false positive. At least one of the baselines is preferred in each of these cases. By producing conservative and fail-fast fixes—such as raising exceptions or adding explicit null checks—the baselines ensure safety even without a full understanding of the broader context.

In ties, we observe three common scenarios. First, all approaches occasionally fail to recognize complex initialization or assignment patterns (e.g., a local variable that can be `null` in corner

Table 4. Comparison of NullRepair and the baselines in regard to cost and performance.

Metric	NullRepair		mini-SWE-agent		SinglePrompt	
	Per Project	Per Error	Per Project	Per Error	Per Project	Per Error
Execution Time (min)	113	1.2	133	1.4	24	0.3
Number of Prompts	897	9.6	855	9.2	93	1.0
Total Tokens (k)	1,411	15	5,632	60	79	0.8
Cost (USD)	$4.23	$0.05	$4.05	$0.04	$0.32	$0.003

cases), leading to incomplete or unnecessary changes (12/26). Second, in 5 further cases, all three approaches fail to recognize that suboptimal annotations added by the annotation framework would need to be corrected. Third, all three tools sometimes produce similarly effective local edits at the error site, requiring minimal context to resolve the issue acceptably (9/26).

Correctness of Classification. We also analyze the performance of NullRepair's error classification step (Section 5.2), on the same 75 error cases, by inferring the ground-truth classification from the reviewers' comments and disagreement-resolution discussions. NullRepair correctly classifies 27/33 true positive errors, and 33/42 false positive errors. This translates to a precision of 75%, a recall of 82%, and an F1 score of 78%. For some incorrect classifications, reviewers also expressed uncertainty about classification during their review, or initially disagreed with each other, indicating that these cases are inherently challenging even for human experts.

C3: NullRepair produces higher-quality patches in 28 of 75 cases, ties for the best patch in 26 further cases, and 39% of patches are likely acceptable to developers compared to baselines' 21% and 24%.

Considering all three criteria together, NullRepair can produce patches that are not only statically correct and test-clean, but also semantically aligned with the surrounding code. The baselines, while effective in some cases, often resort to suppressive strategies that compromise run-time behavior. This leads to a higher rate of test failures and lower overall patch quality.

RQ1: NullRepair's patches are effective, with few test failures, and higher-quality than baseline patches, as judged by manual review.

7.3 RQ2: Cost and Performance

While the goal of NullRepair is to produce developer-acceptable fixes, it must also do so efficiently and at reasonable cost. In this part of the evaluation, we examine the computational and resource-related cost of applying NullRepair across the benchmarks.

We consider three performance metrics:

- **Execution Time.** We report wall-clock time for NullRepair and both baselines.
- **Number of Prompts.** We track the number of prompts sent to the LLM per error. For mini-SWE-agent this corresponds to the number of agent cycles.
- **Token Count and API Cost.** We report the number of tokens used per error to compute the API cost of running NullRepair and the baselines.

Together, these metrics provide a holistic view of the performance and scalability of our approach.

The results are presented in Table 4. The total execution time of NullRepair is 113 minutes, on average per project. However, this average is strongly impacted by the libgdx project, which

contains many more nullability errors than the other projects. When excluding libgdx, the average execution time is roughly halved to 59 minutes. Per error the average execution time is 1.2 minutes. mini-SWE-agent is a bit slower with 1.4 minutes per error. Given that this is a one-time analysis, the execution time is reasonable for practical use. NullRepair creates on average 9.6 prompts per error. This is similar to mini-SWE-agent, which completes after 9.2 cycles per error. In contrast, the SinglePrompt baseline only sends one prompt per error. NullRepair has an average token usage of 15 k per error and 1,411 k per project. Most of these tokens are uncached input tokens. Based on current API pricing, this translates to a cost of $0.05 per error. We believe this is a reasonable overhead. mini-SWE-agent uses significantly more tokens (60 k per error), but as the agent's prompts have large repetitive parts (agent history), most of these tokens are cached input tokens (45 k per error), which are ten times cheaper. Therefore, the cost of mini-SWE-agent is similar to NullRepair at $0.04 per error.

RQ2: NullRepair is reasonably fast, averaging 113 minutes per project, and moderately priced, costing just $0.05 per error.

8 Threats to Validity

The most important threat to the validity of our experiments is that three of twelve benchmarks in Tables 2 and 3 are used as the source of the residual patches we study in Section 4 to design our flowchart. There is a risk that our technique will overfit to the residual errors in these three benchmarks, inflating our results. To mitigate this threat, we re-evaluate our experiments on just the nine benchmarks that are not considered in our preliminary study, and find results that are nearly identical to the overall numbers in Tables 2 and 3: the per-patch repair rate (column R of Per Patch in Table 2) is 61.8% on this subset of benchmarks vs. 62.2% on all benchmarks; the combined repair rate (column R of Combined in Table 2) is 63.4% on the subset, vs. 62.9% on all benchmarks; and the overall test failure rate is 0.26% vs. 0.23% (in Table 3). None of these minor differences would impact our scientific conclusions in any way, so we are confident that the preliminary study's use of some of the same benchmarks does not bias our overall results.

Another threat is generalizability: we evaluate our approach with only one LLM, one nullability checker, one annotation inference tool, and on twelve projects in a single language. Other LLMs, checkers, inference tools, projects, or languages may require further innovation. A further potential threat is data leakage. Although the LLM's training data may include examples of nullability errors, it is unlikely to have encountered the exact nullability errors present in our benchmark projects, or fixes addressing them, as, at the time of the model's training data cut-off date, only one of these projects has adopted NullAway in their public repositories. Another threat is our proxies for patch correctness: Tests can show the presence but not the absence of errors, so passing tests are only weak evidence of correctness. However, tests are an improvement over only measuring error reduction, as much of the prior work on annotation inference does. Our manual evaluation of patch quality could also be biased. We mitigate this threat by 1) blinding ourselves to what approach patches stem from and randomizing the order in which each triplet of patches is presented, 2) having two authors independently review each patch, and 3) releasing the patches and judgments in our artifact, so other researchers can check our reasoning.

9 Related Work

Automated Program Repair. Researchers have proposed many techniques for repairing general software bugs [37], some of which are successfully deployed in practice [2, 48, 65]. These approaches include heuristic, generate-and-validate approaches [36], constraint-based search for semantically

similar code [34], and template-based repair [42]. Recent work builds on neural models, e.g., sequence-to-sequence models [10, 45, 68, 69], language models fine-tuned for the repair task [26, 60], general-purpose LLMs [66], and via agentic LLM prompting [9, 11]. Unlike our work, all these techniques repair run-time failures, typically visible via a failing test case. NullRepair cannot rely on a failing test as a validation oracle, and also must assess whether each error is a false positive. Further work uses general-purpose agents for solving GitHub issues [67, 71]. We compare NullRepair to mini-SWE-agent (Section 7), which is one of the leading openly available agents on SWE-bench Verified [27].

Fixing Compilation Errors. To help developers, especially novices, handle compilation errors (syntax errors, type errors, etc.), researchers have proposed approaches that rely on sequence-to-sequence models trained for this purpose [20] and LLMs [29], or combine a symbolic, error-correcting parser and a neural model [58]. Unlike NullRepair, these approaches assume that editing a single location is sufficient to fix an error. Another key difference is that compiler errors must be fixed from the start, whereas we are looking at situations where one is adopting a type system later during the development process.

Fixing Static Analysis Errors. Our work also relates to prior effort toward automatically fixing other kinds of static analysis errors. One line of work learns from past fixes, e.g., by generalizing past fixes into a transformation rule [2], by expressing patches in a domain-specific language [6], or by fine-tuning a sequence-to-sequence model [8, 12, 28]. Unlike these works, NullRepair does not require past fixes or analysis-specific fine-tuning. For instance, InferFix [28] targets potential null-dereference errors reported by the Infer static analyzer and relies on fine-tuning with a corpus of fixes for Infer warnings. In contrast, NullRepair addresses arbitrary NullAway type system errors, which go beyond null-dereferences (e.g., assigning `@Nullable` to `@NonNull`). It handles false positives, and employs safe usage regions from the codebase being repaired. Similarly, NPEX [38] generates repairs based on null-handling patterns learned from a corpus, which is conceptually related to NullRepair's safe usage regions. However, NPEX is designed to repair null-dereference errors observed at runtime using stack traces as input, whereas NullRepair addresses arbitrary statically reported NullAway errors. Consequently, NullRepair can generate suppressions or annotation insertion repairs, which are important but out of scope for NPEX.

Another line of work defines code transformation rules by hand [17], which is inherently limited to a small set of rules and requires significant effort. Finally, a third line of work deeply integrates a repair technique with a specific static analysis tool, e.g., by using an analysis that reasons about heap properties [62], internal feedback from a static analyzer [70], or symbolic execution to repair errors reported by a Datalog-based analysis [43]. Unlike these deeply integrated approaches, NullRepair is designed to work with any static analysis that reports nullability errors, such as NullAway, in a black-box manner, and focuses on nullability errors rather than general memory errors. Another difference between our work and the above is that we evaluate NullRepair not only based on how many errors it removes, but also on its impact on the test pass rate, which we find to be more informative.

Other Related Work. NullRepair can be seen as another stepping stone to enable static nullability checking in existing code bases, which relates to prior work on type inference, especially for nullability annotation [15, 25, 33, 35, 59], but also for other kinds of type annotations [1, 22, 47, 56, 63]. Parts of our approach relate to prior efforts to automatically distinguish true from false positives in static analysis errors, which has been shown to be a difficult problem [30], and can be addressed through a combination of static dependency analysis and LLM querying [64]. The Llm4sa work [64] has not been applied to Java or nullability yet, but might further improve NullRepair's

effectiveness at deciding when to suppress an error. Suppressions of static analysis errors have been the subject of a recent study [24], which shows that suppressions are relatively common in practice, confirming our design decision to support suppressions in NullRepair. MSA is a technique to port C code to Checked C, i.e., a memory-safe dialect of C [50]. Like our work, MSA also asks an LLM to add annotations or rewrite code to pass static checks that go beyond the core language, but for another language and another family of static checks. NullRepair relates to work on addressing run-time type errors in Python, with template-based [51] and learning-based approaches [21, 54]. ReTyper [21] also retrieves context from the current codebase to assist an LLM, but unlike NullRepair, uses code similarity instead of static analysis to decide what context to retrieve. Finally, NullRepair's false positive decision-making allows the LLM to request additional context if needed. This relates to work [44] on expanding the scope of static analysis on demand, based on whether the current context is sufficient for a conclusive decision.

10 Conclusions

NullRepair advances the goal of fully automated migration to type-based nullability checkers by resolving nullability errors that remain after annotation inference. While our primary focus is on supporting from-scratch enrollment of existing codebases, NullRepair is equally valuable in post-migration cleanup. For example, Uber has already used Annotator to migrate large codebases to NullAway, but many hundreds or even thousands of error suppressions remain. NullRepair can help eliminate these suppressions by synthesizing safe repairs. By continuing the path laid down by annotation inference tools, NullRepair brings us one step closer to the full automation of making software more reliable and robust against null-pointer exceptions.

Data Availability

NullRepair is publicly available at https://github.com/Pascal-Joos/NullRepair, and archived with a DOI on Zenodo [32] to allow easy reproduction of our results.

Acknowledgements

This research was supported in part by the National Science Foundation under grants CCF-2223826, CCF-2312262, and CCF-2312263, the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073), a gift from Oracle Labs, and a Google Research Award. The views expressed herein are those of the authors and do not necessarily reflect those of our funders.

References

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*. 91–105. doi:10.1145/3385412.3385997

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27. doi:10.1145/3360585

[3] David P. Baker. 2024. JSpecify blog: Release 1.0.0. https://jspecify.dev/blog/release-1.0.0/. Accessed July 2026.

[4] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical type-based null safety for Java. In *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia, 740–750. doi:10.1145/3338906.3338919

[5] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 306–317. doi:10.1145/2635868.2635898

[6] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *ESEC/FSE*. 613–624. doi:10.1145/3338906.3338952

[7] SWE bench Team. 2024. SWE-bench Leaderboards. Accessed July 2026. https://www.swebench.com/

[8] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin T. Vechev. 2024. DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models. *CoRR* abs/2402.13291 (2024). arXiv:2402.13291 doi:10.48550/ARXIV.2402.13291

[9] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[10] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959. doi:10.1109/TSE.2019.2940179

[11] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. 2025. Agentic Bug Reproduction for Effective Automated Program Repair at Google. doi:10.48550/arXiv.2502.01821

[12] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 87:1–87:13. doi:10.1145/3597503.3639184

[13] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[14] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 681–690. doi:10.1145/1985793.1985889

[15] Jens Dietrich, David J. Pearce, and Mahin Chandramohan. 2023. On Leveraging Tests to Infer Nullable Annotations. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:25. doi:10.4230/LIPIcs.ECOOP.2023.10

[16] Xueying Du, Jiayi Feng, Yi Zou, Wei Xu, Jie Ma, Wei Zhang, Sisi Liu, Xin Peng, and Yiling Lou. 2026. Reducing False Positives in Static Bug Detection with LLMs: An Empirical Study in Industry. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '26)*. Association for Computing Machinery, New York, NY, USA, 716–727. doi:10.1145/3786583.3786910

[17] Khashayar Etemadi, Nicolas Harrand, Simon Larsén, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikström, and Martin Monperrus. 2023. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *IEEE Trans. Dependable Secur. Comput.* 20, 4 (2023), 2794–2810. doi:10.1109/TDSC.2022.3167316

[18] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. 192–203. doi:10.1145/301618.301665

[19] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024. Connecting Large Language Models with Evolutionary Algorithms Yields Powerful Prompt Optimizers. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 34133–34156. https://proceedings.iclr.cc/paper_files/paper/2024/file/9156b0f6dfa9bbd18c79cc459ef5d61c-Paper-Conference.pdf

[20] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 31, 1 (Feb. 2017). doi:10.1609/aaai.v31i1.10742

[21] Sichong Hao, Xianjun Shi, and Hongwei Liu. 2024. RetypeR: Integrated Retrieval-based Automatic Program Repair for Python Type Errors. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 199–210. doi:10.1109/ICSME58944.2024.00028

[22] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*. 152–162. doi:10.1145/3236024.3236051

[23] David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal) *(PASTE '05)*. Association for Computing Machinery, New York, NY, USA, 13–19. doi:10.1145/1108792.1108798

[24] Huimin Hu, Yingying Wang, Julia Rubin, and Michael Pradel. 2025. An Empirical Study of Suppressed Static Analysis Warnings. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE014 (June 2025), 22 pages. doi:10.1145/3715729

[25] Piyush Jha and Werner Dietl. 2024. OppropBERL: A GNN and BERT-Style Reinforcement Learning-Based Type Inference. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 131–136. doi:10.1109/SANER60148.2024.00021

[26] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *ICSE*. 1430–1442. doi:10.1109/ICSE48619.2023.00125

[27] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? doi:10.48550/arXiv.2310.06770

[28] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1646–1656. doi:10.1145/3611643.3613892

[29] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press, 5131–5140. doi:10.1609/AAAI.V37I4.25642

[30] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: how far are we?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 698–709. doi:10.1145/3510003.3510214

[31] Nima Karimipour, Erfan Arvan, Martin Kellogg, and Manu Sridharan. 2025. A New Approach to Evaluating Nullability Inference Tools. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE017 (June 2025), 22 pages. doi:10.1145/3715732

[32] Nima Karimipour, Pascal Joos, Michael Pradel, Martin Kellogg, and Manu Sridharan. 2026. *Reproduction Package for "LLM-Based Repair of Static Nullability Errors"*. doi:10.5281/zenodo.20645902

[33] Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. 2023. Practical Inference of Nullability Types. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1395–1406. doi:10.1145/3611643.3616326

[34] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 295–306. doi:10.1109/ASE.2015.60

[35] Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. 2023. Pluggable Type Inference for Free. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1542–1554. doi:10.1109/ASE56229.2023.00186

[36] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* 38, 1 (Jan. 2012), 54–72. doi:10.1109/TSE.2011.104

[37] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. doi:10.1145/3318162

[38] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. *Proceedings - International Conference on Software Engineering* 2022-May (7 2022), 1532–1544. doi:10.1145/3510003.3510186

[39] 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 (4 2024). Issue OOPSLA1. doi:10.1145/3649828

[40] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. doi:10.48550/arXiv.2405.17238

[41] Kilian Lieret and contributors. 2025. mini-SWE-agent: The 100 line AI agent that's actually useful. Accessed July 2026. https://mini-swe-agent.com

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *ISSTA*. ACM, 31–42. doi:10.1145/3293882.3330577

[43] Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program Repair Guided by Datalog-Defined Static Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1216–1228. doi:10.1145/3611643.3616363

[44] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. 2008. Verifying Dereference Safety via Expanding-Scope Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 213–224. doi:10.1145/1390630.1390657

[45] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA*. ACM, 101–114. doi:10.1145/3395363.3397369

[46] Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 1033–1052. doi:10.1145/2048066.2048144

[47] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. doi:10.1109/ICSE.2019.00045

[48] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, 269–278. doi:10.1109/ICSE-SEIP.2019.00039

[49] JSpecify members. 2026. JSpecify: Standard annotations for Java static analysis. https://jspecify.dev/. Accessed July 2026.

[50] Nausheen Mohammed, Akash Lal, Aseem Rastogi, Rahul Sharma, and Subhajit Roy. 2025. LLM Assistance for Memory Safety. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1717–1728. doi:10.1109/ICSE55347.2025.00023

[51] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 922–934. doi:10.1145/3540250.3549130

[52] OpenAI. 2024. OpenAI API. https://openai.com/. Accessed July 2026.

[53] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212. doi:10.1145/1390630.1390656

[54] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. doi:10.1145/3597503.3608132

[55] Artem Pianykh, Ilya Zorin, and Dmitry Lyubarskiy. 2022. Retrofitting null-safety onto Java at Meta. https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/. Accessed July 2026.

[56] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. doi:10.1145/3368089.3409715

[57] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*. Baltimore, MD, USA, 24–36. doi:10.1145/2771783.2771791

[58] Georgios Sakkas, Madeline Endres, Philip J. Guo, Westley Weimer, and Ranjit Jhala. 2022. Seq2Parse: neurosymbolic parse error repair. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1180–1206. doi:10.1145/3563330

[59] Kazi Amanul Islam Siddiqui and Martin Kellogg. 2025. Inferring Pluggable Types with Machine Learning. doi:10.48550/arXiv.2406.15676

[60] André Silva, Sen Fang, and Martin Monperrus. 2024. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. doi:10.48550/arXiv.2312.15698

[61] Federico Tomassetti and contributors. 2025. JavaParser: Visitable Java 1.8+ Abstract Syntax Tree. https://javaparser.org/. Accessed July 2026.

[62] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. 151–162. doi:10.1145/3180155.3180250

[63] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Hkx6hANtwH

[64] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* 18, 7, Article 168 (June 2024), 34 pages. doi:10.1145/3653718

[65] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. 2024. User-Centric Deployment of Automated Program Repair at Bloomberg. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 81–91. doi:10.1145/3639477.3639756

[66] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 819–831. doi:10.1145/3650212.3680323

[67] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 50528–50652. doi:10.52202/079017-1601

[68] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. doi:10.1145/3510003.3510222

[69] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 13 pages. doi:10.1145/3597503.3623337

[70] Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2025. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 69 (Feb. 2025), 27 pages. doi:10.1145/3705310

[71] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

Received 2026-01-30; accepted 2026-04-16