Frustrated with Code Quality Issues? LLMs can Help!

NALIN WADHWA, Microsoft Research, India

JUI PRADHAN, Microsoft Research, India

ATHARV SONWANE, Microsoft Research, India

SURYA PRAKASH SAHU, Microsoft Research, India

NAGARAJAN NATARAJAN, Microsoft Research, India

ATHARV SONWANE, Microsoft Research, India

ADITYA KANADE, Microsoft Research, India

SURESH PARTHASARATHY, Microsoft Research, India

SRIRAM RAJAMANI, Microsoft Research, India

As software projects progress, quality of code assumes paramount importance as it affects reliability, maintainability and security of software. For this reason, static analysis tools are used in developer workflows to flag code quality issues. However, developers need to spend extra efforts to revise their code to improve code quality based on the tool findings. In this work, we investigate the use of (instruction-following) large language models (LLMs) to assist developers in revising code to resolve code quality issues.

We present a tool, CORE (short for COde REvisions), architected using a pair of LLMs organized as a duo comprised of a proposer and a ranker. Providers of static analysis tools recommend ways to mitigate the tool warnings and developers follow them to revise their code. The *proposer LLM* of CORE takes the same set of recommendations and applies them to generate candidate code revisions. The candidates which pass the static quality checks are retained. However, the LLM may introduce subtle, unintended functionality changes which may go un-detected by the static analysis. The *ranker LLM* evaluates the changes made by the proposer using a rubric that closely follows the acceptance criteria that a developer would enforce. CORE uses the scores assigned by the ranker LLM to rank the candidate revisions before presenting them to the developer.

We conduct a variety of experiments on two public benchmarks to show the ability of CORE: ❶ to generate code revisions acceptable to both static analysis tools and human reviewers (the latter evaluated with user study on a subset of the Python benchmark), ❷ to reduce human review efforts by detecting and eliminating revisions with unintended changes, ❸ to readily work across multiple languages (Python and Java), static analysis tools (CodeQL and SonarQube) and quality checks (52 and 10 checks, respectively), and ❹ to achieve fix rate comparable to a rule-based automated program repair tool but with much smaller engineering efforts (on the Java benchmark). CORE could revise 59.2% Python files (across 52 quality checks) so that they pass scrutiny by both a tool and a human reviewer. The ranker LLM reduced false positives by 25.8% in these cases. CORE produced revisions that passed the static analysis tool in 76.8% Java files (across 10 quality checks) comparable to 78.3% of a specialized program repair tool, with significantly much less engineering efforts.

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

Additional Key Words and Phrases: Code quality, static analysis, code revision, LLMs

1 INTRODUCTION

As software projects progress, assessing reliability, maintainability and security of software assumes paramount importance. Quality of code plays a big role in ensuring these objectives [20, 22]. Many static analysis tools like CodeQL [11], Coverity [2], FindBugs [4], PMD [7] and SonarQube [8] are used in developer workflows to flag code quality issues. However, developers need to spend extra efforts to revise their code to improve code quality based on the tool findings [51, 55].

Recognizing the value of static analysis tools in improving code quality, many approaches [12, 14, 19, 25, 26, 32, 33, 37, 46, 50] use them to detect and localize violations of static checks. To fix the violations, they use either manually designed symbolic program transformations [19, 25, 46, 50], mine symbolic patterns from commit data [12, 14, 32, 33, 37] or learn them from synthetically

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani

Python code flagged by the quality check

```python
def separate_headers(files):
    for file in files:
        lines = iter(file)
        header = next(lines) # Will raise StopIteration
        body = [ l for l in lines ]
        yield header, body
```

Fix recommendation

Each call to `next()` should be wrapped in a `try-except` to explicitly handle `StopIteration` exceptions.

Revised Python code fixing the issue

```python
def separate_headers(files):
    for file in files:
        lines = iter(file)
        try:
            header = next(lines) # Will raise StopIteration
        except StopIteration:
            #Empty file -- Just ignore
            continue
        body = [ l for l in lines ]
        yield header, body
```

(a) Example of a quality check "Unguarded `next` in generator" for Python code from CodeQL

Java code flagged by the quality check

```java
class ComputePrimesThread extends Thread {
    @Override
    public void run() {
        // ...
    }
}
new ComputePrimesThread().run(); // Noncompliant
```

Fix recommendation

If you intend to execute the contents of the `Thread.run()` method within a new thread, call `Thread.start()` instead.

Revised Java code fixing the issue

```java
class ComputePrimesThread extends Thread {
    @Override
    public void run() {
        // ...
    }
}
new ComputePrimesThread().start(); // Compliant
```

(b) Example of a quality check "`Thread.run()` should not be called directly" for Java code from SonarQube

Fig. 1. Examples of quality checks, fix recommendations, and code before and after following the fix recommendations for CodeQL and SonarQube tools for Python and Java languages respectively.

generated data [26]. The code or fix generation capabilities of these symbolic approaches are limited by the space of supported patterns. Learning-based approaches [28, 49, 54, 56] try to overcome this limitation by training neural models to map buggy programs to their fixed versions. However, similar to pattern-mining approaches, these require bug-fixing data for training and it limits the types of bugs they can fix. Setting up these systems and supporting a different programming language, a new quality check or another static analysis tool incurs significant engineering costs. These factors prevent the wide-spread adoption of these automated program repair (APR) tools.

Providers of static analysis tools recommend ways to mitigate the tool warnings. Developers can follow them to manually revise their code when warnings are raised. Figure 1 illustrates two quality checks (a) and (b) from two tools: CodeQL applied to Python code[1] and SonarQube applied to Java code[2]. At the top are code snippets with quality issues. The natural-language fix recommendations for the quality checks are shown in the middle and the revised code that can be obtained after manually following the fix recommendations are shown at the bottom.

The APR tools try to learn mapping between the original and revised snapshots of the code. *To avoid the limitations of APR tools outlined above, we propose to instead make direct use of the clear and concise natural-language instruction (fix recommendation) supplied by the tool providers.* The emergence of large language models (LLMs) (e.g., [15–17, 40, 48]) offers an opportunity to make this possible. LLMs are large neural networks that capture generative distributions of natural languages and source code. These models are trained on very large data in unsupervised manner. Instruction-tuning [41] enhances their utility by finetuning the base LLMs to comprehend and

---

[1]https://codeql.github.com/codeql-query-help/python/py-unguarded-next-in-generator/
[2]https://rules.sonarsource.com/java/RSPEC-1217/

follow natural language instructions. As we show in this paper, it is possible to *instruct* state-of-the-art LLMs to revise a piece of code directly using natural language instructions. These models can sample a variety of code conditioned on instructions, that too without any additional training or finetuning. Unlike symbolic program transformations and neural models trained on specific datasets, they are not limited by the space of patterns or bug-fixing data used for training. This eliminates the need to expend the efforts required in designing symbolic transformation systems or training specialized neural models, and saves on engineering efforts.

We try to realize the promise of LLMs to resolve code quality issues flagged by static analyses in a tool called CORE (short for COde REvision). CORE is architected using a pair of LLMs organized as a duo comprised of a proposer and a ranker. The *proposer LLM* of CORE takes a fix recommendation and applies it to a given source-code file to generate candidate code revisions. The candidates which pass the static quality checks are retained. However, the LLM may introduce subtle, unintended functionality changes which may go un-detected by the static analysis. The *ranker LLM* evaluates the changes made by the proposer using a rubric that closely follows the acceptance criteria that a developer would enforce. CORE uses the scores assigned by the ranker LLM to rank the candidate revisions before presenting them to the developer.

We evaluate CORE on two public benchmarks: CodeQueries [45] and Sorald [46]. CodeQueries is a benchmark of Python files with quality issues flagged by one of the 52 common static checks applied by the CodeQL tool. Sorald comprises of Java repositories with quality issues flagged by one of the 10 common static checks applied by the SonarQube tool. Both the datasets contain code from public GitHub repositories and are representative of real-world quality issues.

We conduct a variety of experiments on these benchmarks to show the ability of CORE: ❶ to generate code revisions acceptable to both static analysis tools and human reviewers (the latter evaluated with user study on a subset of the Python benchmark), ❷ to reduce human review efforts by detecting and eliminating revisions with unintended changes, ❸ to readily work across multiple languages (Python and Java), static analysis tools (CodeQL and SonarQube) and quality checks (52 and 10 checks, respectively), and ❹ to achieve fix rate comparable to a rule-based automated program repair tool, Solard, but with much smaller engineering efforts (on the Java benchmark).

We obtain promising results that bear witness to practical utility of CORE using GPT-3.5-Turbo [41] as the proposer LLM and GPT-4 [40] as the ranker LLM. CORE could revise 59.2% Python files (across 52 quality checks) so that they pass scrutiny by both a tool and a human reviewer. The ranker LLM reduced the false positive rate by 25.8% in these cases. CORE produced revisions that passed the static analysis tool in 76.8% Java files (across 10 quality checks) compared to 78.3% of the specialized program repair tool Solard [46], but with significantly much lesser engineering efforts. The authors of Solard state that "The design and implementation of SORALD already represents 2+ years of full time work." [46], whereas we were able to apply CORE on the Solard benchmark with a few days of efforts by a couple of authors.

There is growing interest in using LLMs in program repair. Many existing techniques [21, 35, 52, 53] aim at fixing bugs characterized by failing test cases. In comparison, we focus on fixing quality issues that are discovered statically and do not have accompanying unit tests for validation. The techniques that repair statically detected errors [29, 30, 42] either target syntactic or simple semantic errors [30], finetune an LLM on specially designed prompts [29] or use less powerful models and prompting [42]. We target a wide range of code quality issues using an instructed-tuned LLM which supports powerful prompting without any finetuning.

We make the following contributions in this paper:

- We identify an emerging opportunity of using instruction-following LLMs to assist developers in resolving code quality issues.

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani
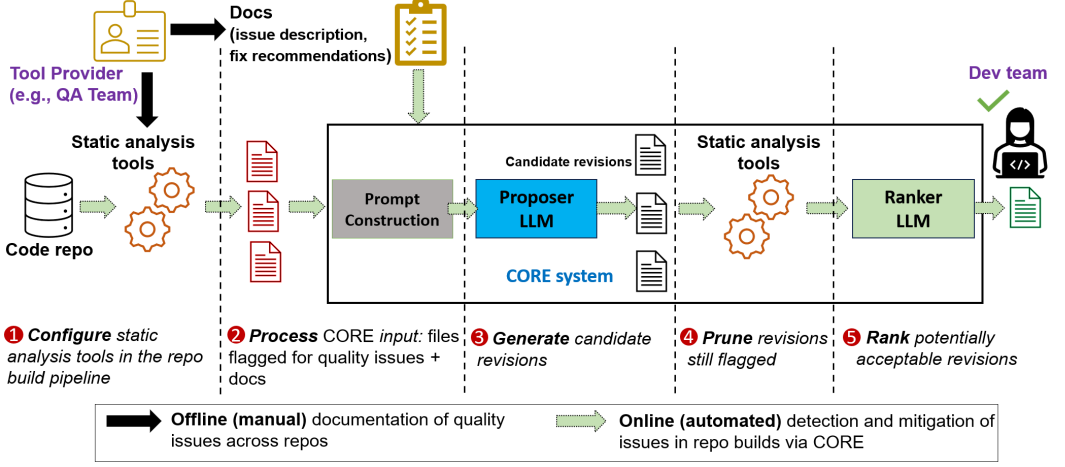
Fig. 2. CORE pipeline: Code quality issues (static checks) across software repositories are documented by the tool provider. CORE is integrated in the repo build pipeline that also runs the suite of static analysis checks. The flagged source files and the documentation are fed as input to the CORE system to automatically produce source file revisions that address the quality issues. The candidate revisions that pass the static checks are further assessed and ranked by a ranker LLM to prevent surfacing spurious fixes to the developer.

- We present a system, CORE, to evaluate this opportunity. We design a multi-step protocol wherein one LLM proposes the code revisions, which are filtered by applying the static analysis and further ranked using a ranker LLM, before they are presented to the developer.
- We conduct extensive experimentation to evaluate the acceptability of the revisions produced by CORE, its ability to control false positives, generalizability to different languages, tools and checks, and its performance compared to a specialized program repair tool. Our results show that CORE is a promising step in bringing LLMs to the help of developers in resolving code quality issues. We also identify opportunities for further improvements.
- We will to release our code and data at https://aka.ms/CORE_MSRI.

2 OVERVIEW

Our goal is to (1) automate code quality revisions in software engineering workflows, which typically comprise large-scale code repositories and various code quality control checks; (2) by taking static analysis tools and documentation of quality checks (natural language instructions) as input; (3) with minimal developer intervention. This section gives an overview of the architecture of CORE using an example code quality issue resolution scenario.

Consider the Python code snippet shown in Code 1 below. The `PersistentDict` class derives from `dict` class, adding `_filename` and `_transact` attributes of its own. It does not override the `__eq__` method. This is an instance of poor code quality that can cause errors: when two objects of `PersistentDict` class are compared, the subclass attributes get ignored. This example code gets flagged by the CodeQL tool with the Eq-Not-Overridden warning [3].

Software engineering workflows involve such quality checks for readability, maintainability, security, etc. Accompanying these checks are the guidelines (natural language instructions) for

fixing the issues in the source files that are flagged by static analysis tools. For the EQ-NOT-OVERRIDDEN check, the CodeQL manual page on the web[3] states *"A class that defines attributes that are not present in its superclasses may need to override the __eq__() method (__ne__() should also be defined)"*. Tool developers or quality assurance (QA), security and compliance teams in organizations write the static quality checks and documentation (fix recommendations), and the repository owners (dev team) are responsible for fixing the quality issues based on the provided guidelines.

```python
class PersistentDict(dict):
    """A class that persists a dict to a file. This class behaves like a dict and adds new
        ↪ functionality to store the dict to a file when writing."""
    def __init__(self, filename, load=True):
        self._filename = os.path.abspath(filename)
        if load: self._load()
        self._transact = False

    @property
    def filename(self):
        'The filepath to write'
        return self._filename
```

Code 1. Example Python code with EQ-NOT-OVERRIDDEN issue.

In the following, we present an overview of the CORE pipeline that is configured with the inputs from the tool provider along with the source-code files flagged by the tools, and produces automatic revisions of the source files to address the issues.

❶ **Configuring the CORE pipeline:** CORE, shown in Figure 2, is a generic pipeline for code revisions. In order to configure CORE to process the EQ-NOT-OVERRIDDEN issue, the tool provider supplies two types of information to CORE: (1) the static analysis tool (e.g., CodeQL) and the check itself (e.g., a .ql file), (2) a description of the code quality issue and instructions to fix the issue in natural language. In our evaluation, we obtain the description of the quality issues and the instructions to fix them from online documentation. Once CORE is configured with the two types of information, it can automatically process static analysis reports corresponding to this issue, along with the files containing the code that needs to be revised to fix the issue, and propose candidate revisions to the code.

> *The aforementioned manual configuration for a code quality issue is a one-time, offline step, that serves to produce revisions (in an automated, online fashion) for the issue arising in various repositories, thereby automating the repetitive task of resolving the code quality issues.*

We describe the components of the CORE pipeline next.

❷ **Constructing prompt:** This component takes in the static analysis report, the flagged source file, and the documentation for the issue (see above), and constructs a "prompt" for querying the large language model (LLM). A *prompt* encodes the natural language instruction to solve a particular task and optionally additional information that the model might use to perform the task such as hints (e.g., lines of interest in the file), constraints (e.g., "do not modify parts of code unrelated to the issue"), and demonstrations (e.g., an example code snippet with __eq__ not overridden and its revised version). Typically, each LLM has a limit on the prompt size (also referred as context size) it supports in terms of the number of tokens. In CORE, the prompt encapsulates the description of the quality issue, fix suggestions, and the line(s) of code the static analysis report attributes the

---

[3]https://codeql.github.com/codeql-query-help/python/py-missing-equals/

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani

issue to. In addition, CORE employs other relevant information that may be helpful to fix the issue, such as fetching relevant blocks of code derived from static analysis reports. The details of prompt construction are presented in Section 3.

❸ **Generating candidate revisions using Proposer LLM:** The proposer LLM takes as input the constructed prompt in natural language along with the code flagged for the quality issue and outputs potential code revisions. We use GPT-3.5-Turbo, which is a state-of-the-art LLM for code generation, in our experiments. GPT-3.5-Turbo supports large prompt sizes (up to 4000 tokens). This lets us input the entire source-code file for many cases. For very large files, we give the largest context block admissible (e.g., the entire method or class surrounding the lines of interest) by the prompt size (details in Section 3). The output code (a block or a full file, as the case may be) is then patched back to the original file. We sample 10 candidate revisions for each input file.

❹ **Pruning revisions with the configured tool:** We run the static analysis check (that CORE is configured with in ❶) against the candidate revisions and filter out the ones where the code quality issue continues to persist (i.e., non-zero violations detected). In this process, we de-duplicate identical code revision suggestions as well as ensure the syntactic validity of code revisions, and reject the ones with syntax errors.

❺ **Ranking admissible candidates using Ranker LLM:** The static analysis tool could pass revisions that are not acceptable to developers, such as introducing unintended changes in the code (e.g., a revision that overrides the `__eq__` method properly, but alters the implementation of `__hash__` unnecessarily). The impressive editing and generative ability of state-of-the-art LLMs also means that they can make alterations (often subtle, but changing the semantics) to existing code even if they are explicitly instructed *not* to do so. Furthermore, if the quality of the static check (usually some form of pattern matching) itself is poor, it is even more important to ensure incorrect revisions are not surfaced to the developer who will eventually accept or reject them. To reduce the burden on the developers, and to improve the acceptance rate of the surfaced fixes, we employ another LLM (GPT-4) to score the candidate revisions in the order of their likelihood of acceptance. To instruct the LLM to do so, we rely on the same rubric, i.e., instructions for issue resolution supplied in ❶, to construct a prompt for the LLM. In particular, we give the candidate revision that passed the static analysis tool (diff with the original source code) along with the rubric in natural language, as input to the Ranker LLM and ask it to assign an ordinal score in a range (from *strong accept* to *strong reject*). We use this score to rank the potential candidates for a file. The details of the ranking strategy are discussed in Section 3.

Sample output: Code snippet shown below is a sample output generated by CORE for Code 1. Due to lack of space, instead of showing the full code, we display the outputs in the standard diff format with respect to Code 1.

```
@@ -128,0 +130,4 @@ class PersistentDict(dict):
+  def __eq__(self, other):
+    if isinstance(other, PersistentDict):
+      return dict.__eq__(self, other) and self._filename == other._filename and self.
      ↪ _transact == other._transact
+    return False
```

Code 2. A correct revision of Code 1 generated by CORE.

The above code snippet is a correct way of revising Code 1. It calls the `__eq__` method of the super class and compares all the member variables of the subclass. Further, it does not make any unnecessary changes to the code.

3 DESIGN

Designing and tuning prompts for querying LLMs is a thriving new area of research [34]. In this section, we describe in detail the prompt construction strategies, guided by static analysis reports. The LLM invocations in our pipeline are for generating candidate revisions, and for scoring and ranking the candidates.

3.1 Proposer LLM: Prompting the LLM to generate code revisions

To generate code revisions for a given code quality issue and an input source file, we devise a prompt template incorporating different types of information, with elaborate natural language instructions, needed to perform the revision task. Our prompt follows the generic structure shown below, with *fixed components* (p₁ and p₂, as per configuration done in ❶ discussed in Section 2) as well as *instance-specific components* (p₃, p₄, and p₅) obtained dynamically:

> **Proposer Prompt Template**
>
> p₁ Description of the quality issue (i.e., static check).
> p₂ Rubric for resolving the quality issue.
> p₃ (Optional) Relevant code blocks for doing the revision.
> p₄ Input source file (in full, or localized to the block containing the issue).
> p₅ Location and warning message given by the static check.

The fixed components of the prompt consist of the name of the quality check, description, and recommended ways to resolve the issue. These are provided at the time of configuring the CORE pipeline. In our empirical evaluation in Section 5, we simply use the content from the documentation webpages [1, 8]. When there are missing or incomplete details about how to fix issues, we furnish the information. The instantiated prompt for our running example Code 1 is given in Figure 3. The text in *italics* is the template, the text in teal correspond to the fixed components obtained from the tool providers, and the text in brickred correspond to instance-specific information retrieved from static analysis (CodeQL for this example) reports.

Handling multiple violations in the input file: The static analysis tool gives us the locations (lines of interest), and in some cases associated warning messages as well, where the issue was flagged in the input source file. There can be multiple locations in a single source file where the check violation is flagged. If the source file is sufficiently small (to fit in the context size of the LLM), we give the entire source file (in p₄) as well as all the flagged locations and warning messages (in p₅) in a single prompt. If not, we use the following procedure to instantiate the prompt(s).

Let $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ denote the flagged locations in the input file. We start with v_1 and take its largest encompassing block, e.g., the class or the method containing v_1, that fits in the available context size. Note that this block might also encompass more violation locations v_i besides v_1. We instantiate a prompt with this block as the input (in p₄), and the subset of flagged violations (in p₅). We remove this location subset from $\mathcal{V}$ and repeat. Thus, the proposer LLM is invoked with possibly multiple instantiated prompts for a single input file, and the generated code block for each prompt is patched back to the original source file appropriately.

Relevant code blocks: During static analysis, tools like CodeQL identify and inspect code blocks that are relevant for determining presence/absence of a property violation. We log this information while running the static analysis and provide it as additional signal in our prompt in p₃. For example, for a CodeQL check "signature mismatch in overriding method", the declaration of the overriden

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya
Kanade, Suresh Parthasarathy, and Sriram Rajamani

PROPOSER PROMPT (output of "Prompt Construction" stage in Figure 2)

p₁ *We are fixing code that has been flagged for the* CodeQL *warning titled "`__eq__` not overridden when adding attributes" which has the following description:*
A class that defines attributes that are not present in its superclasses may need to override the __eq__() method (__ne__() should also be defined).
Adding additional attributes without overriding __eq__() means that the additional attributes will not be accounted for in equality tests.

p₂ *The recommended way to fix code flagged for this warning is:*
Override __eq__ method to also test for equality of added attributes by either calling eq on the base class and checking equality of the added attributes, or implementing a new eq method that checks equality on both self and inherited attributes.

p₄ *Modify the Buggy code below to fix the* CodeQL *warning(s). Output the entire code block with appropriate changes. Do not remove any section of the code unrelated to the desired fix.*

Buggy Code:
```
class PersistentDict (dict) :
    ...
```

p₅ CodeQL *warning(s) for the above buggy code:*
The class 'PersistentDict' does not override "__eq__" , but adds the new attributes "_filename" and "_transact".

The following lines are likely to be of interest:
```
1. class PersistentDict (dict) :
```

Fixed Code:

Fig. 3. Prompt supplied to the Proposer LLM for revising Code 1. This example does not require additional relevant code blocks as context and hence, the corresponding prompt component **p₃** is not present.

method from the superclass is a relevant block because the static check determines the mismatch between the overriden and overriding methods by inspecting their signatures. On the other hand, in the example of Code 1, the CodeQL error message already provides sufficient information. As shown in Figure 3, **p₅** gives the CodeQL diagnostics that `_filename` and `_transact` attributes are added in the subclass and are not covered by the `__eq__` method of the superclass. Thus, the fix can be constructed from local code with this diagnostic information, and there is no need for any other part of the source file. For such checks, we do not supply **p₃**.

3.2 Ranker LLM: Prompting the LLM to score candidate revisions

As we stated in Section 2, static analysis tools could pass revisions that are not acceptable, e.g., introducing unintended changes or otherwise altering functional correctness of the source code.

Ranker Prompt

You are an expert developer. You are verifying the code generated by LLM to fix the warning titled "`__eq__` not overridden when adding attributes" which has the following description: A class that defines attributes that are not present in its superclasses may need to override the __eq__() method (__ne__() should also be defined). ...

The recommended ways to fix code flagged for this warning are:
Override __eq__ method to also test for equality of added attributes by either calling eq on the base class and checking equality of the added attributes, or ...

Your task is to assess the quality of the generated patch and rate it on the following evaluation criteria:
Score 0, if the patch has changes unrelated and unnecessary to fixing the warning (Strong Reject).
Score 1, if the patch has a few correct fixes, but still modifies the original snippet unnecessarily (Weak Reject).
Score 2, if the patch has mostly correct fixes but is still not ideal (Weak Accept).
Score 3, if the patch only makes edits that fix the warning with least impact on any unrelated segments of the original snippet (Strong Accept).

If you find additions or deletions of code snippets that are unrelated to the desired fixes (think LLM hallucinations), it can be categorically scored 0 (Strong Reject). That said, you can make exceptions in very specific cases where you are sure that the additions or deletions do not alter the functional correctness of the code, as outlined next.

Allowed Exceptions:
The following (unrelated) code changes in the diff file can be considered okay and need not come in the way of labeling an otherwise correct code change as accept (score 2 or 3). This list is not exhaustive, but you should get the idea
(a) deleting comments is okay,
(b) rewriting a = a + 1 as a += 1 is okay, even though it may not have anything to do with the warning of interest,
(c) making version specific changes is okay, say changing print ("hello") to print "hello".

The following (unrelated) code changes in the diff file are NOT considered okay, and you should label the diff file as reject (score 0 or 1) even if it is otherwise correct for the query. This list is not exhaustive, but you should get the idea
(a) deleting or adding a print statement,
(b) optimizing a computation,
(c) changing variable names or introducing typos.

Output only the reason and score for the patch below. Do not output anything else.

Diff: ⟨ diff ⟩
Reason:

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani

In the running example of Code 1, we see two kinds of revisions that are likely to be rejected by developers: (1) revisions that override the `__eq__` method properly, but alter the functionality of the code elsewhere, such as changing the implementation of `__hash__`, and (2) revisions that do not quite resolve the quality issue, but by-pass the CodeQL checks anyway — for instance, all the subclass members are explicitly enumerated in the equality check without calling `super().__eq__` for the parent members. We do not want to surface such spurious candidates to the developer.

To this end, we use another instance of the LLM to act as a ranker that scores the candidate revisions that pass the tool, i.e., output of stage ❹ in Figure 2, in the CORE pipeline. We use a prompting strategy similar to the one used for generating the revisions themselves in the previous subsection to query the ranker LLM. The prompt template for scoring candidates is given above (some parts are elided to save space). Note that the prompt is fairly generic, and in particular, is agnostic to the type of code quality check or the static analysis tool.

In addition to information about the quality check and fix recommendation, the prompt provides description of a scoring rubric that asks the LLM to rank the revisions (from strong reject to strong accept) based on how close they are to the intended change while avoiding unrelated changes. Sometimes the LLM enforces coding conventions or styles that it encountered frequently during training and rewrites code to conform to those. We instruct the LLM to overlook such changes (indicated by "Allowed Exceptions" in the prompt) and provide some example scenarios. Similarly, we elaborate more on what kind of revisions should be rejected in the last part of the prompt, before providing the "Diff" for scoring.

4 EXPERIMENTAL SETUP

Datasets: (1) We use a subset of the CodeQueries dataset [45][4] in our experiments. It contains Python files with quality issues flagged by a set of 52 CodeQL queries (i.e., static checks). The 52 CodeQL queries are taken from the standard Python CodeQL suite; these analyze various aspects of code such as security, correctness, maintainability, and readability. In all our experiments, we use the **test split** of the CodeQueries dataset. Due to throttled LLM access, we are able to experiment with a subset comprising 765 files across the 52 queries. We denote this dataset as CQPʏ. Further, we sample 10 files per query from CQPʏ to conduct a user study on revisions generated by CORE. We refer to this subset as CQPʏUS. **(2)** We use a subset of the Sorald dataset [46][5]; this subset consists of a collection of 151 java repositories from Github (out of the 161 repositories in their full dataset) with a total of 483 files, and covers all the 10 SᴏɴᴀʀQᴜʙᴇ checks studied in [46]. We refer to this set as SQJᴀᴠᴀ.

Model configurations: We conduct our experiments using the **GPT-3.5-Tᴜʀʙᴏ** model as the **proposer**. We obtain 10 responses per input source file using the OpenAI inference API. Following recent work [10], to encourage diversity in the sampled responses, we use a combination of *temperature* settings for the model (that controls the stochasticity in the generated responses): 1 response with temperature = 0 (greedy decoding), 6 responses with a temperature of 0.75, and 3 responses with a temperature of 1.0. We use **GPT-4** as the **ranker** (our early investigations suggested that GPT-4 is significantly better in terms of reasoning with code diffs compared to GPT-3.5-Tᴜʀʙᴏ) and obtain a single response (score) per candidate revision, with temperature 0.

Evaluation metrics: For each dataset, we report the number of files flagged and the number of total issues flagged across the files. We measure how many files have at least one revision that passes the static check and how many issues remain in files with no such revision after the Proposer

---

[4]https://huggingface.co/datasets/thepurpleowl/codequeries
[5]https://github.com/khaes-kth/Sorald-experiments

LLM and Ranker LLM stages of CORE are applied. In the user study, we measure how many files have at least one revision that is accepted by the human reviewer and report how many revisions were accepted and rejected across the files by the reviewers. We refer to the number of revisions produced by the CORE pipeline but rejected by the human reviewer as *false positives*.

5 EVALUATION

Our goal is to extensively evaluate the end-to-end CORE pipeline across various quality-improving code revision tasks and to answer the following questions:

RQ1: How *effective* is the *end-to-end* CORE pipeline in mitigating code quality issues and in passing scrutiny by the Ranker LLM on the Python benchmark CQPy?

RQ2: How many of the CORE-generated revisions are also *accepted by human reviewers* on the Python benchmark CQPyUS?

RQ3: How *readily* does CORE pipeline *generalize* to a different programming language (Java) and a static analysis tool (SonarQube)?

RQ4: How *well* does CORE *compare* to a state-of-the-art automatic program repair technique (Sorald) for mitigating static analysis warnings?

5.1 RQ1: How *effective* is the *end-to-end* CORE pipeline in mitigating code quality issues and in passing scrutiny by the Ranker LLM on the Python benchmark CQPy?

We start by looking at the overall performance of the CORE pipeline in terms of (1) fixing the code quality issues as determined by the static analysis tool that CORE is configured with, and (2) acceptances as determined by the Ranker LLM using a detailed evaluation criteria to assess the code revisions (as described in Section 3.2).

The overall evaluation results of the end-to-end CORE pipeline (on the datasets and metrics introduced in Section 4) are presented in Table 1. For RQ1, we will focus on the first row, that corresponds to CORE pipeline configured with CodeQL as the static analysis tool, and the 52 quality checks that are part of the CQPy Python dataset.

The first block of columns shows the dataset statistics. There are 1993 quality issues (i.e., static check violations) flagged in the 765 files of the CQPy dataset, with each file having at least one issue flagged, by the end of stage ❶ in Figure 2. In the second block of columns, we show the effectiveness of the Proposer LLM, *after* the proposed candidate revisions (10 revisions per flagged file) are filtered by the tool (i.e., CodeQL for the first row) by the end of stage ❹. First, we observe that 81.57% of the flagged files get fixed entirely as validated by the static analysis tool, i.e., they have at least one revision that completely passes the static checker with zero issues flagged. Second, we observe that, the average number of issues remaining per revised file, by the end of stage ❹ of the CORE pipeline, is 0.41 compared to over 2.6 issues on average per source file at the beginning of the pipeline. This is particularly remarkable as the Proposer LLM is able to perform revision with just the natural language instructions, without explicitly providing any training examples of the form ⟨*before* code, *after* code⟩ that are commonly needed for automatic program repair tools.

The instruction-following ability of the Proposer LLM to do code revisions, although impressive, can also produce spurious fixes that pass static checks. In the last stage of the CORE pipeline, the Ranker LLM uses elaborate evaluation criteria (in its carefully-constructed prompt presented in Section 3.2) to reject such spurious fixes and accept revisions that are likely to be also accepted by developers. From the last block of columns of Table 1, we see that 583 out of 624 files are ranked high, i.e., strong or weak accept, by the Ranker LLM by the end of stage ❺. In particular, the Ranker LLM (strong- or weak-) rejects every revision (possibly spurious) for 41 files even though they are

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani

Dataset	Dataset statistics		Effectiveness of Proposer LLM		Rankings by Ranker LLM	
	#Files flagged	#Issues flagged (Avg. per file)	#Files passing static checks (%)	#Issues remaining (Avg. per file)	#Files ranked high (%)	low (%)
CQPʏ	765 (100%)	1993 (2.61)	624 (81.57%)	315 (0.41)	583 (76.21%)	41 (5.36%)
CQPʏUS	520 (100%)	999 (1.90)	453 (87.11%)	159 (0.31)	427 (82.11%)	26 (5.00%)
SQJᴀᴠᴀ	483 (100%)	999 (2.06)	397 (82.19%)	270 (0.56)	371 (76.81%)	26 (5.38%)

Table 1. Summary of end-to-end evaluation of CORE on real-world Python and Java files, with 52 and 10 static checks using CodeQL and SᴏɴᴀʀQᴜʙᴇ respectively. "#Files flagged" and "#Issues flagged" correspond to output of static checks (stage ❶ in Figure 2). "#Files passing static checks" and "#Issues remaining" report the number of files having at least one revision that passes the static checks and the issues that remain in files with no such revision (stage ❹ output). "#Files ranked high (low)" is the number of files with at least one revision (no revision, respectively) that is scored as weak/strong accept by the Ranker LLM (stage ❺ output). For files, the percentages are reported with respect to the "#Files flagged".

Stage evaluated	Stage-wise output		Results of user study		
	#Files retained	#Revisions retained	% Files accepted (#)	% Revisions accepted (#)	% Revisions rejected (#)
Stage ❹ (Proposer LLM)	453 (100%)	2397 (100%)	70.64% (320)	44.89% (1076)	55.11% (1321)
Stage ❺ (Ranker LLM, SA)	410 (100%)	1756 (100%)	**72.68%** (298)	**52.45%** (921)	47.55% (835)
Stage ❺ (Ranker LLM, WA)	17 (100%)	228 (100%)	58.82% (10)	36.40% (83)	63.60% (145)
Stage ❺ (Ranker LLM, WR/SR)	26 (100%)	413 (100%)	46.15% (12)	17.43% (72)	**82.57%** (341)

Table 2. Results of user study on the CQPʏUS dataset. "Ranker LLM, SA" denotes all the revisions scored as strong accept by the Ranker LLM; "Ranker LLM, WA" denotes all the revisions scored as weak accept by the Ranker LLM, and "Ranker LLM, WR/SR" denotes all the revisions scored as rejects (strong or weak) by the Ranker LLM. For files and revisions, the percentages are reported row-wise with respect to the numbers in the first block of columns (under "Stage-wise output"). Column-wise maximums are in the bold typeface.

passed by the tool in stage ❹. In the subsequent RQ, we analyse how well the acceptances and the rejections by the Ranker LLM correlate with human reviewers, on a subset of the CQPʏ dataset.

5.2 RQ2: How many of the CORE-generated revisions are also *accepted by human reviewers* on the Python benchmark CQPʏUS?

In this RQ, we investigate the correctness of the revisions produced by CORE, and in particular the effectiveness of the Ranker LLM, by conducting a user study. We use a subset of CQPʏ, called CQPʏUS, with a sample of 10 files per quality check, which already yields 2397 candidate revisions (out of stage ❹) to be manually scrutinized. The CORE pipeline results for this dataset are presented in the second row of Table 1, where the trend closely resembles that of CQPʏ in the first row.

For each of the 453 files in CQPʏUS that comes out of stage ❹ (as seen from row 2, Table 1), we ask a human reviewer to label all the revisions for the file as *accept* or *reject*. We provide the same rubric that we give as prompt to the Ranker LLM (presented in Section 3.2) to the reviewer to assess the correctness of the revisions — the only change is that we ask the reviewer to give a binary accept/reject decision than a graded score that we elicit from the Ranker LLM. Our user group consists of 15 Python developers (intermediate level, with 1-3 years of software engineering experience). Each revision was labeled by only one user, and each user was responsible for labeling

revisions of 2 to 4 (randomly chosen) queries from the dataset. None of the authors of this paper were part of the user group.

The results of the CQPYUS user study are presented in Table 2. The first row of the Table shows the (baseline) metrics for the user study we conducted — all the outputs of stage ❹ were reviewed by the users, and 70.64% of the reviewed files have at least one revision that a human reviewer accepted. However, from the last column, we see that this high acceptance rate comes at a high cost of 1321 false positives, i.e., 55.11% of the revisions that CodeQL passed were rejected by users. This trade-off between acceptance rate and false positives of the pipeline can be crucial in practice. In the following, we show that the Ranker LLM helps achieve a significantly better trade-off.

Equipped with the accept/reject labels given by users for all the CodeQL-passed revisions of the CQPYUS dataset, we ask: *Can the Ranker LLM help tell the correct revisions from the incorrect ones, which would in turn help minimize the review burden of developers?* We answer this question affirmatively in the subsequent rows of Table 2. From the second row, we see that if we surface only the candidates strongly accepted by the Ranker LLM, the rejection rate drops to 47.55%. In an absolute sense, the number of rejections drops to 835 from 1321, which is close to 25% reduction. At the same time, 72.68% of the scrutinized files have at least one revision accepted by a reviewer. Furthermore, from the last row, we see that if we consider only the files for which no revision was (strong- or weak-) accepted by Ranker LLM, the users also rejected over 82% of those revisions; this indicates that dropping the low confidence rejections by the Ranker LLM can indeed help significantly reduce the review burden of developers in practice.

> CORE *pipeline, with the Proposer-Ranker duo LLMs, for resolving code quality issues is effective and can be deployed in real software engineering workflows. Relying only on the (symbolic) tools for filtering revisions is problematic — the Ranker LLM helps reduce the number of false positives greatly, while also ensuring that acceptable revisions which preserve functional correctness are surfaced to the developers.*

5.3 RQ3: How *readily* does CORE pipeline *generalize* to a different programming language (Java) and a static analysis tool (SonarQube)?

CORE can handle different programming languages and static analysis tools out of the box. To demonstrate this, in this RQ, we configure CORE with another widely-used static analysis tool SONARQUBE, and the 10 static checks from the SQJAVA dataset (introduced in Section 4). This configuration was straight-forward; *it took us less than a week to get this done.* In fact, lines of code that needed changes in our CORE implementation (in Python) for this configuration was less than 100. Specifically, we did *not* have to adapt or tune the prompts of the Proposer and Ranker LLMs in our pipeline to accommodate the new tool or the programming language. The authors of the Sorald dataset have made available clear descriptions and fix recommendations for the 10 checks, which we readily use to instantiate our LLM prompts. Further, SONARQUBE provides localization for the check violations (line numbers in the source file) needed to extract code blocks as discussed in Section 3.1.

We report results on the SQJAVA dataset consisting of real-world Java repositories in the last row of Table 1. There are 999 quality issues flagged in 483 files of the dataset, with each file having at least one issue flagged, by the end of stage ❶ of CORE. As in the case of the other datasets (first and second rows), we find that over 82% files have at least one candidate revision that entirely passes the associated SONARQUBE check. Furthermore, the average number of issues that remain by the end of stage ❹ is about 0.56 per file, compared to over 2 issues per file on average to begin

with. From the last column, we see that the Ranker LLM rejects all the (possibly spurious) revisions that passed SonarQube checks for 26 files, and (strong- or weak-) accepts at least one revision for 371 files, which is over 76% of the total files.

5.4 RQ4: How *well* does CORE *compare* to a state-of-the-art automatic program repair technique (Solard) for mitigating static analysis warnings?

	#Files (%)	#Issues remaining (%)
Flagged	483 (100%)	999 (100%)
CORE	371 (76.8%)	270 (27.03%)
Sorald	378 (78.3%)	371 (37.14%)

Table 3. Comparison of CORE with the state-of-the-art automatic program repair method Sorald on the SQJava dataset consisting of 10 static checks, using SonarQube as the static analysis tool. "#Files (%)" for CORE and Solard rows indicate the number of files (% with respect to the Flagged files) that are fixed by the tools respectively.

We compare CORE with the state-of-the-art automatic program repair tool Sorald [46] for fixing static check issues in code. Sorald is a rule-based approach that leverages "metaprogramming templates", which are basically AST-to-AST transformations, that can be applied on the detected violations in code. In particular, for each violation location in the code, Sorald applies one metaprogramming template to the corresponding AST element to fix it. They manually implement one metaprogramming template per static check, based on the fix recommendations for the check, which we directly use in the form of natural language instructions in the CORE pipeline. While their repair tool is extensible to other languages and static analysis tools, their publicly available implementation[6] is for Java and SonarQube. So, for this RQ, we focus on the SQJava dataset.

The comparison results on the Java dataset are presented in Table 3. Of the 483 files in the dataset, CORE, i.e., the output of stage ❺, considering the (strong/weak) accepted revisions by the Ranker LLM, fixes 371 files entirely, at a rate of 76.8%. This is comparable to the manually crafted Sorald tool that fixes 378 files. On the other hand, the number of issues that remain by the end of CORE pipeline is 270 (about 27%), significantly less compared to 371 (over 37%) for the Sorald tool.

6 THREATS TO VALIDITY

A possible threat to validity is that the input code in our dataset might have been seen by the LLM during its training. The LLM is unlikely to have seen the prompts constructed by us paired with the expected code revisions during training. Therefore, our results can be attributed to the ability of the LLMs to follow the instructions, their knowledge of programming languages and the informative details we provide in our prompts. By basing our experiments on hundreds of issues flagged by 52 diverse static checks for Python and 10 diverse static checks for Java from two different static analysis tools, we avoid the possibility of biasing our results to a small dataset, certain code quality issues, or a single tool or programming language. We follow the exact experimental setup as CodeQueries and Solard to avoid any language or tool version mismatch issues.

The code generated by the LLM may pass the previously failing static checks but change the code semantics, e.g., by completely deleting the code. To mitigate this problem, we perform human evaluation for verifying soundness of the revisions, albeit on a subset of our Python dataset, but

---

[6]https://github.com/ASSERT-KTH/sorald

ensuring full coverage in terms of the static checks. This manual labeling could be noisy. All the labels were independently reviewed by one of the authors to avoid such cases.

We found cases where the users were unsure why the tool flagged a violation in the source code in the first place, or whether the fix in the revision had no unintended side effects. There were also a few cases that proved to be challenging to manually verify the correctness of the revisions. For instance, consider the "import * may pollute namespace" static check for Python files[7]. The correct revisions would replace the * with relevant modules. However, verifying if all the required imports are fully enumerated can be challenging, especially for large source files. Looking at multiple revision candidates for some files was helpful to users in this regard — whenever two candidates for a file had a non-overlapping subset of enumerated imports, the user tried to reason about the differences and was able to resolve incompleteness of one or both the revisions. To avoid cases from inflating or otherwise biasing our evaluation results, we instructed the users to *reject* revisions that they were unsure of, as in some of the examples mentioned above, erring on the safer side.

7 RELATED WORK

Automatic program repair is a topic of active research and many tools have been built over the years. Here, we discuss the most closely related work and refer the reader to excellent surveys [23, 24, 38] for wider coverage.

7.1 Repairing static check violations

Among the approaches that target static analysis errors, [19, 25, 46, 50] use manually designed symbolic program transformations to fix specific classes of properties like heap safety [50], security vulnerabilities [25], static quality checks [46] or data races [19]. Other approaches [12, 14, 32, 33, 37, 44] mine symbolic patterns from commit data to learn repair strategies or learn them from synthetically generated data [26]. For instance, SPONGEBUGS [37] uses SonarQube [8] to find bugs and commit data to create paired dataset. Similarly, AVATAR [32, 33] and PHOENIX [14] use FindBugs [4] and commit data. REVISAR [44] mines edits from commit data for PMD [7]. GETAFIX [12] uses Infer [5] and Error Prone [9], and mines general tree edit patterns from commit data using anti unification.

These repair techniques can synthesize only those fixes that are covered by their symbolic patterns. An alternative approach based on learning [28, 49, 54, 56] is to train neural models to map buggy programs to their fixed versions. The neural models learn to directly transform code. However, their scope is determined by the diversity of bug-fixing examples present in the training data and they do not generalize to new classes of bugs not seen during training.

All these approaches require extensive data curation and offline learning efforts, and require redesign when targeting different kinds of bugs. In contrast, the line of work we pursue, using LLMs, does not require any data curation or learning effort. Since LLMs have aleady been pretrained with a large corpus of code and other documents, they can be readily customized to revise code to fix any type of error detected by static analysis, just by suitably authoring prompts.

7.2 LLMs for program repair

The aforementioned advantage of using LLMs has motivated other researchers to use them for program repair. Xia, Wei and Zhang [52] use LLMs with few-shot prompts to generate candidate fixes on buggy code from Defects-4J, QuixBug and ManyBugs benchmarks and use entropy values (the negative log probability of each generated token) to rank candidate fixes. The work relies on the existence of a test suite to validate a candidate fix. In a more recent work, Xia and Zhang [53]

---

[7]https://codeql.github.com/codeql-query-help/python/py-polluting-import/

use a conversational approach, where a test suite is a requirement, and error messages from failed tests are used in a conversational style with the LLM to refine the candidate fix into one that passes the test suite, and present results on the QuixBug benchmarks. Another interesting line of work is to fix bugs in code generated by an LLM using traditional program repair techniques or another LLM [21, 27, 35] .

These approaches aim at fixing bugs identified by failing test cases. In comparison, our work addresses a related but different problem, one of fixing errors flagged by static analysis tools such as CodeQL and SonarQube.

Prompting techniques: RING [30] fixes syntactic and simple semantic errors across multiple languages using an LLM and retrieval-augmented few-shot prompting. The complexity of errors and required fixes in our case is higher. InferFix [29] targets violations flagged by the Infer static analyzer [5, 6] for three types of bugs. However, it constructs prompts augmented with bug type annotation and similar bug-fix pairs, and finetunes the Codex model on these prompts. We use an instruction-based LLM in zero-shot setting (i.e., no ⟨*before* code, *after* code⟩ examples needed) without finetuning. Pearce et al. [42] fix security vulnerabilities using auto-regressive LLMs which are prompted with partial code in which the buggy lines are commented out and the LLM is prompted to generate a "fixed" version of those. We use a more powerful class of instruction-tuned LLMs which benefits from detailed instructions that provide additional context necessary for generating correctly revised code. Our prompts encompass description of the quality issue, suggested resolutions, localization hints and constraints. Due to the auto-regressive nature, the generations in [42] are conditioned only on prefix of the buggy code, whereas we pass the buggy code in the prompt and hence, the code generation can attend to the bidirectional code context, both before and after the buggy lines in the input code.

Validating fixes: The above approaches rely on static analysis to validate the fixes and availability of unit tests to detect regressions. We use a combination of two oracles: (1) the static analyzer, and (2) a second ranker LLM, to ensure that the fix does not vacuously pass the static analysis, to automatically generate acceptable fixes that are both functionally correct and pass static analysis.

7.3 LLMs as verifiers

The issue of plausible but incorrect fixes is well-known [36, 43]. CORE may generate code that passes the static check (a plausible fix) but changes semantics of the input code in unintended ways. The developer can review the statically-validated code-revisions to filter out such cases. Unit tests can also help catch such cases, but they are not always available or may themselves be incomplete. LLMs have been shown to be effective in assessing and supervising quality of output from other LLMs [13, 31], thereby helping reduce the efforts required for human review. Using LLMs, especially GPT-4, for evaluating code generations has been attempted recently. Olausson et al. [39] also have a dual LLM setup, where they use the feedback from GPT-4 in the form of critique to modify the prompt of the proposer LLM for code generation tasks. Zhuo [57] constructs an elaborate prompt for GPT-3.5-Turbo to perform two aspects of evaluation of code generations, namely, code usefulness and evaluation-based functional correctness. Inspired by these findings, to reduce the burden on the developer, we employ a second instance of LLM (GPT-4) as a ranker to score the candidates produced by the proposer LLM based on (1) the correctness of issue resolution, and (2) preserving the functional correctness of the original code. The code generation datasets studied in [57] consist mostly of small code snippets, unlike our setting where we use large real source code files. We work with code diffs in the Ranker LLM prompt, and in our investigations, GPT-4 is substantially better in terms of reasoning with code diffs compared to GPT-3.5-Turbo that Zhuo [57] employs.

8 CONCLUSIONS AND FUTURE WORK

Code quality is a persistent concern in software engineering. Though much progress has been made in detecting these issues statically, fixing them automatically has remained challenging due to the variety of code quality issues that surface in real code. Our proposal in this work is to use the power of large language models, particularly, those that go beyond code completion and can follow natural language instructions, to assist developers in revising and improving their code. Through comprehensive evaluation on two public benchmarks in Python and Java that use 52 and 10 static checks from two different tools, we show the promise of this approach when coupled with carefully crafted prompts. We further show that by employing an LLM instance as a ranker, that assesses the likelihood of acceptance of proposed code revisions, we can effectively catch plausible but incorrect fixes and reduce developer burden.

Our objective for future is to expand the scope of our tool CORE by building more components in the pipeline to not only support more tools and checks but to also improve the quality and correctness of the generated fixes. We believe that feedback-driven continuous improvement is a key to make this work mainstream. For this, we plan to draw upon the traditional static and dynamic analysis techniques for automated feedback generation and use the recent advances in finetuning of the models using techniques based on reinforcement learning and human feedback [18, 41, 47].

REFERENCES

[1] [n. d.]. CodeQL website. https://codeql.github.com/. Accessed: September 15, 2023.
[2] [n. d.]. Coverity Static Analysis. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html. Accessed: September 15, 2023.
[3] [n. d.]. __eq__ not overridden when adding attributes. https://codeql.github.com/codeql-query-help/python/py-missing-equals/. Accessed: September 15, 2023.
[4] [n. d.]. FindBugs Project. https://spotbugs.github.io/. Accessed: September 15, 2023.
[5] [n. d.]. Infer static analyzer. https://fbinfer.com/. Accessed: September 15, 2023.
[6] [n. d.]. InferSharp static analyzer. https://github.com/microsoft/infersharp. Accessed: September 15, 2023.
[7] [n. d.]. PMD: An extensible cross-language static code analyzer. https://pmd.github.io/. Accessed: September 15, 2023.
[8] [n. d.]. SonarQube. https://docs.sonarqube.org/latest/. Accessed: September 15, 2023.
[9] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 14–23.
[10] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram K Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. *arXiv preprint arXiv:2306.10763* (2023).
[11] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
[12] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (oct 2019), 27 pages. https://doi.org/10.1145/3360585
[13] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional AI: Harmlessness from AI Feedback. *arXiv preprint arXiv:2212.08073* (2022).
[14] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952
[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[18] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

[19] Andreea Costea, Abhishek Tiwari, Sigmund Chianasta, Abhik Roychoudhury, and Ilya Sergey. 2023. Hippodrome: Data race repair using static analysis summaries. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–33.

[20] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[21] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. *arXiv preprint arXiv:2205.10583* (2022).

[22] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[23] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[24] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *arXiv preprint arXiv:2303.18184* (2023).

[25] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. 539–554. https://doi.org/10.1109/SP.2019.00071

[26] Naman Jain, Shubham Gandhi, Atharv Sonwane, Aditya Kanade, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2023. StaticFixer: From Static Analysis to Static Repair. arXiv:2307.12465 [cs.SE]

[27] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[28] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[29] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. *arXiv preprint arXiv:2303.07263* (2023).

[30] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair is nearly generation: Multilingual program repair with llms. *arXiv preprint arXiv:2208.11640* (2022).

[31] Tom Kocmi and Christian Federmann. 2023. Large language models are state-of-the-art evaluators of translation quality. *arXiv preprint arXiv:2302.14520* (2023).

[32] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2884955

[33] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandè. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1–12. https://doi.org/10.1109/SANER.2019.8667970

[34] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[35] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. *arXiv preprint arXiv:2304.10423* (2023).

[36] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[37] D. Marcilio, C. A. Furia, R. Bonifacio, and G. Pinto. 2019. Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings. In *2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Los Alamitos, CA, USA, 34–44. https://doi.org/10.1109/SCAM.2019.00013

[38] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[39] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. arXiv:2306.09896 [cs.CL]

[40] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[41] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[42] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE

Computer Society, 1–18.

[43] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 87–102.

[44] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. arXiv:1803.03806 [cs.SE]

[45] Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and Shirish Shevade. 2022. CodeQueries: A Dataset of Semantic Queries over Code. *arXiv preprint arXiv:2209.08372* (2022).

[46] Khashayar Etemadi Someoliayi, Nicolas Yves Maurice Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. 2022. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *IEEE Transactions on Dependable and Secure Computing* (2022).

[47] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* 33 (2020), 3008–3021.

[48] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[49] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[50] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[51] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.

[52] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[53] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[54] 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*. 1506–1518.

[55] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 334–344. https://doi.org/10.1109/MSR.2017.2

[56] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

[57] Terry Yue Zhuo. 2023. Large Language Models Are State-of-the-Art Evaluators of Code Generation. arXiv:2304.14317 [cs.AI]