CORE: Resolving Code Quality Issues using LLMs

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
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: (1) 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), (2) to reduce human review efforts by detecting and eliminating revisions with unintended changes, (3) to readily work across multiple languages (Python and Java), static analysis tools (CodeQL and SonarQube) and quality checks (52 and 10 checks, respectively), and (4) 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. We release code, data, and supplementary material publicly at http://aka.ms/COREMSRI.

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

Authors' addresses: Nalin Wadhwa, Microsoft Research, India, t-nalwadhwa@microsoft.com; Jui Pradhan, Microsoft Research, India, juipradhan2k@gmail.com; Atharv Sonwane, Microsoft Research, India, t-asonwane@microsoft.com; Surya Prakash Sahu, Microsoft Research, India, suryaprakashsahuat1@gmail.com; Nagarajan Natarajan, Microsoft Research, India, nagarajan.natarajan@microsoft.com; Aditya Kanade, Microsoft Research, India, kanadeaditya@microsoft.com; Suresh Parthasarathy, Microsoft Research, India, supartha@microsoft.com; Sriram Rajamani, Microsoft Research, India, sriram@microsoft.com.

ACM 2994-970X/2024/7-ART36
https://doi.org/10.1145/3643762

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

ACM Reference Format:
Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proc. ACM Softw. Eng.* 1, FSE, Article 36 (July 2024), 23 pages. https://doi.org/10.1145/3643762

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 [21, 23]. Static analysis tools like CodeQL [12], Coverity [2], FindBugs [4], PMD [7] and SonarQube [8] are used in developer workflows to flag code quality issues. For instance, CodeQL can be integrated in GitHub workflows and is estimated to be used in tens of thousands of repositories. However, developers need to spend extra efforts to revise their code to improve code quality based on the tool findings [59, 63].

Recognizing the value of static analysis tools in improving code quality, many approaches [13, 15, 20, 27, 28, 37, 38, 43, 53, 58] use them to detect and localize violations of static checks. To fix the violations, they use either manually designed symbolic program transformations [20, 27, 53, 58], mine symbolic patterns from commit data [13, 15, 37, 38, 43] or learn them from synthetically generated data [28]. The code or fix generation capabilities of these symbolic approaches are limited by the space of supported patterns. Learning-based approaches [30, 56, 62, 64] 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., [16–18, 46, 55]) 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 [47] enhances their utility by finetuning the base LLMs to comprehend and 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.

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

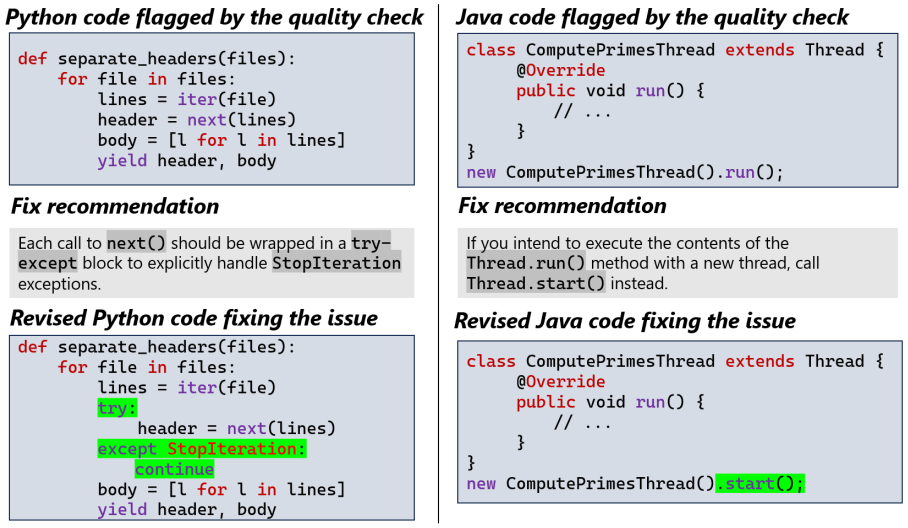

Fig. 1. Examples of quality checks (left) "Unguarded `next` in generator" and (right) "`Thread.run()` should not be called directly", fix recommendations, and code before and after following the fix recommendations for (left) CodeQL and (right) SonarQube tools for (left) Python and (right) Java languages.

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 [51] and Sorald [53]. 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, Sorald, 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 [47] as the proposer LLM and GPT-4 [46] 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 Sorald [53], but with significantly much lesser engineering efforts.

The authors of Sorald state that "The design and implementation of SORALD already represents 2+ years of full time work." [53], whereas we were able to apply CORE on the Sorald benchmark within a week's time. Of course, the authors of Sorald must have spent time to build the dataset and reusable artifacts, which we get readily. However, the key point we want to highlight is that unlike Sorald, we do not incur the cost of engineering an AST-to-AST transformation system, since the LLM does code rewrites itself. Further, we analyzed the experimental results to identify strengths and weaknesses of CORE to inform future research.

There is growing interest in using LLMs in program repair. Many existing techniques [22, 40, 60, 61] 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 [31, 32, 48] either target syntactic or simple semantic errors [32], finetune an LLM on specially designed prompts [31] or use less powerful models and prompting [48]. We target a wide range of code quality issues using instruction-tuned LLMs which support powerful prompting without any finetuning.

We make the following contributions in this paper:

(1) We identify an emerging opportunity of using instruction-following LLMs to assist developers in resolving code quality issues.
(2) 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.
(3) We conduct extensive experimentation to evaluate (a) acceptability of the revisions produced by CORE, (b) its ability to control false positives, (c) generalizability to different languages, tools and checks, (d) its performance compared to a specialized program repair tool, (e) ablations of key choices in CORE, and (f) a qualitative study. Our results show that CORE is a promising step in bringing LLMs to the help of developers in resolving code quality issues.
(4) We release code, data, and supplementary material publicly at http://aka.ms/COREMSRI.

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.

```
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.

Consider the Python code snippet shown above. The `PersistentDict` class derives from `dict` class, adding `_filename` and `_transact` attributes of its own. It does not override the `__eq__`

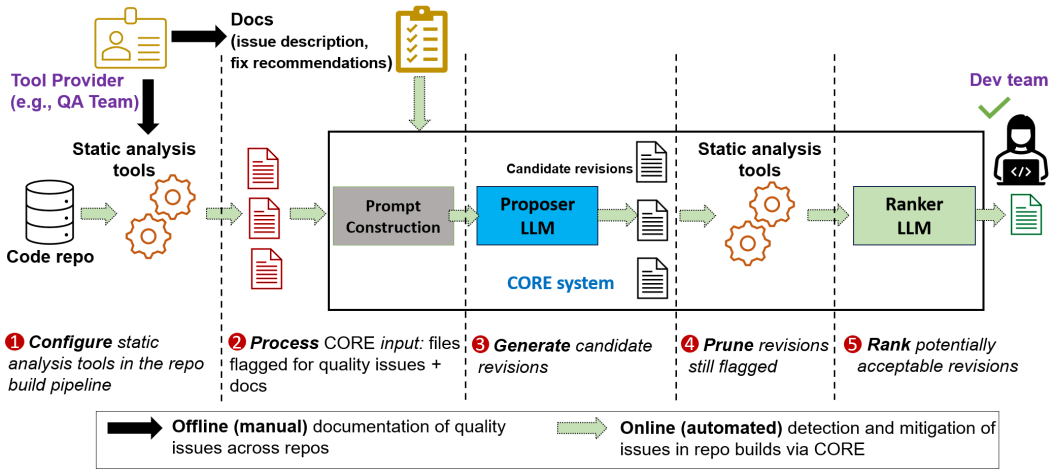

Fig. 2. CORE pipeline: Code quality issues (static checks) across software repositories are documented by the tool provider. CORE is integrated in the repo build pipeline that also runs the suite of static analysis checks. The flagged source files and the documentation are fed as input to the CORE system 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.

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 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.

CORE pipeline 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 configured, CORE 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 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 [39]. 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* (p_1 and p_2, as per configuration done in ❶ discussed in Section 2) as well as *instance-specific components* (p_3, p_4, and p_5) obtained dynamically:

Proposer Prompt Template

- p_1 Description of the quality issue.
- p_2 Recommendations for resolving the quality issue.
- p_3 (Optional) Relevant code blocks for doing the revision.
- p_4 Input source file (in full, or localized to the block containing the issue).
- p_5 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.

We use remediation details verbatim from the webpages [1, 8] for 41/62 checks used in our study (see Section 4); for the remaining 21 checks, we use our domain expertise to write fix remediation, as there is no clear information in the webpages (see supplementary material for details). Note that validity of all the fixes generated by the LLM is checked automatically by the static analysis tool (step 4, Figure 2), including for fixes generated using the remediation instructions written by us. Once the pipeline is configured, CORE programmatically extracts warning messages from the static analysis reports needed for prompt construction. The instantiated prompt for our 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.

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.

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 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 Algorithm 1 to first extract code blocks of a predetermined size, each containing one or more more issue violations. We then instantiate p4 and p5 with a concatenation of the returned pairs of code blocks and the corresponding warning messages.

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 method from the superclass is a relevant block because the static check determines the mismatch

Algorithm 1: Psuedocode for handling multiple violations in the Proposer LLM prompt

Data: source file to fix F, set of issue violation locations $\mathcal{V}$ flagged in F, maximum size of a code block T (in terms of number of tokens).
let *block_prompt_groups* be an empty $Dict(CodeBlock, List(Issues))$;
for v_i *in* $\mathcal{V}$ **do**
  $c_i \leftarrow LargestEncompassingBlock(v_i, T, F)$ // Returns the largest encompassing Class or method or window around the issue location, of size smaller than the threshold T
  if c_i *in block_prompt_groups* **then** $block\_prompt\_groups[c_i].append(v_i)$
  else $block\_prompt\_groups[c_i] = [v_i]$
return *block_prompt_groups*

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. 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.

Ranker Prompt Template

r1 Description of the quality issue. *(same as p1 in the Proposer template.)*
r2 Recommendations for resolving the quality issue. *(same as p2 in the Proposer template.)*
r3 Rubric for scoring the revisions on the scale of "Strong Reject", "Weak Reject", "Weak Accept", "Strong Accept".
r4 Input candidate revision as "Diff" with its source file.

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 4 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. Note that the prompt is fairly generic, and in particular, is agnostic to the type of code quality check or the static analysis tool.

Ranker Prompt (Details)

r1 *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). ...

r2 *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 ...

r3 *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.

r4 *Output only the reason and score for the patch below. Do not output anything else.*

Diff: ⟨ diff ⟩
Reason:

In addition to information about the quality check and fix recommendation, the prompt includes description of a scoring rubric for the LLM to rank revisions, from strong accept (score 3) to strong reject (score 0). The scoring criteria are (a) whether the revision addresses the issue(s) at hand and (b) whether it introduces any unrelated changes. On the other hand, the proposer LLM could

enforce certain coding convention or style that it encountered frequently during training. We instruct the ranker LLM to overlook such changes as long as they do not impact the functional semantics of the code. Similarly, we also give examples of changes that should be rejected (as shown in the full ranker LLM prompt above).

4 EXPERIMENTAL SETUP

Datasets: (1) We use a subset of the CodeQueries [51] dataset[3] 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 (version 2.5.0) 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 use a subset of 2752 files across 52 static checks. We denote this dataset as CQPy. Further, we sample 10 files per query from CQPy to conduct a user study on revisions generated by CORE. We refer to this subset as CQPyUS. **(2)** We use a subset of the Sorald [53] dataset[4]; this subset consists of a collection of 151 java repositories from Github. The original dataset contains an additional 10 large repositories, that substantially increase the number of flagged files for 3 of the 10 issues (75% total increase) in the dataset. So, to avoid biasing the results and findings towards these specific issues and files, we exclude the 10 repositories from our experiments. Our dataset has a total of 483 files, and covers all the 10 SonarQube checks studied in [53]. We refer to this set as SQJava.

The static checks used in these two datasets cover a diverse spectrum of analyses targeting type checking, exception handling, class inheritance, control-flow and data-flow properties, concurrency errors and others, across two popular programming languages on real-world code.

Model configurations: We conduct our experiments using the **GPT-3.5-Turbo** model as the **proposer**. We obtain 10 responses per input source file using the OpenAI inference API. Following recent work [11], 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. In Section 5.6, we consider the effect of a different LLM choice in the pipeline. 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-Turbo) and obtain a single response (score) per candidate revision, with temperature 0. We set `max_tokens` to 4000 for GPT-3.5-Turbo and 1000 for GPT-4.

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 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:

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

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?
RQ5: Where does CORE *succeed* and where does it *fail*?
RQ6: What is the effect of using a *less powerful LLM* in the CORE pipeline?

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 5389 quality issues (i.e., static check violations) flagged in the 2752 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 88.81% 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 0.25 compared to over 1.95 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 2325 out of 2752 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 119 files even though they are passed by the tool in stage ❹. In the subsequent RQ, we analyze how well the acceptances and the rejections by the Ranker LLM correlate with human reviewers, on a subset of the CQPy dataset.

> CORE *pipeline is effective for resolving code quality issues in real software engineering workflows that rely on static tools for quality assurance.* CORE *produces a list of the candidate revisions in decreasing order of confidence for a given file, using an elaborate grading criteria including checking for unintended side-effects and semantic correctness which the static tools miss.*

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 SonarQube 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".

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 (%)	#Files ranked low (%)
CQPy	2752 (100%)	5389 (1.95)	2444 (88.81%)	693 (0.25)	2325 (84.48%)	119 (4.32%)
CQPyUS	520 (100%)	999 (1.90)	453 (87.11%)	159 (0.31)	427 (82.11%)	26 (5.00%)
SQJava	483 (100%)	999 (2.06)	397 (82.19%)	270 (0.56)	371 (76.81%)	26 (5.38%)

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

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 CQPy, called CQPyUS, 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 CQPy in the first row.

For each of the 453 files in CQPyUS 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) static checks from the dataset. On average, a user spent about 4 hours to finish the labeling assignment (difficulty of labeling varies significantly across static checks and, in some cases, across files within checks). They were given a tutorial on the CodeQL static checks and fix recommendations. They had access to the internet to refer to the CodeQL online documentation. 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 passed CodeQL 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

Table 2. Results of user study on the CQPyUS 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.

Stage evaluated	Stage-wise output		Results of user study		
	#Files retained	#Revisions retained	% Files accepted (#)	% Revisions accepted (#)	% Revisions rejected (#)
Stage 4 (Proposer LLM)	453 (100%)	2397 (100%)	70.64% (320)	44.89% (1076)	55.11% (1321)
Stage 5 (Ranker LLM, SA)	410 (100%)	1756 (100%)	**72.68%** (298)	**52.45%** (921)	47.55% (835)
Stage 5 (Ranker LLM, WA)	17 (100%)	228 (100%)	58.82% (10)	36.40% (83)	63.60% (145)
Stage 5 (Ranker LLM, WR/SR)	26 (100%)	413 (100%)	46.15% (12)	17.43% (72)	**82.57%** (341)

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.

It is evident from our user study that 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 applied to Java code 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 1 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 4 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.

Table 3. Comparison of CORE with the state-of-the-art APR technique 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.

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

CORE *can be readily extended to new static tools and programming languages with minimal engineering efforts and lines of code changes.*

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

We compare CORE with the state-of-the-art automatic program repair (APR) technique Sorald [53] for fixing static check issues in code. We use the latest verion of Sorald (v0.8.5) [9]. 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 [9] 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 5, 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 (about 37%) for the Sorald tool. Note that since we tested CORE against the latest version of Sorald (v0.8.5) [9], the results we reported for Sorald in Table 3 are better than the results reported in the study by the authors (refer to Table 4 [53]).

CORE *is competitive to state-of-the-art automatic program repair tool* Sorald *with significantly less engineering efforts, and with absolutely no tuning of the pipeline for the benchmark.*

5.5 RQ5: Where does CORE *succeed* and where does it *fail*?

We present qualitative analysis of CORE outputs, with a deep-dive of some of the results presented in the above subsections.

(i) Prompt size needed vs. the performance on various static checks: We present a more fine-grained analysis of the results in Tables 1 and 2 in Figure 4. We show how the performance of CORE varies by query and by the prompt size (i.e., Proposer and Ranker LLM prompts) needed per file. Larger source files tend to be more challenging in general as LLMs have to reason with very long contexts. For the Proposer LLM (top table), we use the performance metric of Table 1, i.e., the fraction of files (falling in the bin) passing static checks (for the query). For the Ranker LLM (bottom table), we use the file acceptance rate after human review (as in Table 2) as the metric.

Static checks are numbered by their order of CodeQL pass rate in the 0-50th perc. bin

Size\Check	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0-50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
50-100	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.88	0.86	0.86	0.83
Overall (Norm.)	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02

Size\Check	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	Overall (Norm.)
0-50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.90	0.86	0.86	0.83	0.83	0.83	0.80	0.75	0.75	0.67	0.67	0.67	0.67	0.60	0.60	0.40	0.00	0.45
50-100	0.80	0.80	0.80	0.75	0.75	0.67	0.67	0.67	0.60	NA	1.00	0.00	0.75	0.50	0.50	0.80	0.67	0.67	1.00	1.00	0.75	0.57	0.80	0.60	0.40	0.50	0.42
Overall (Norm.)	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.02	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.87

Static checks are numbered by their order of (user) acceptance rate in the 0-50th perc. bin

Size\Check	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0-50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
50-100	NA	1.00	1.00	1.00	1.00	1.00	0.88	0.88	0.86	0.86	0.86	0.86	0.83	0.80	0.67	0.60	0.57	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.40	0.33
Overall (Norm.)	0.01	0.02	0.01	0.02	0.00	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.01	0.02	0.02	0.01	0.02	0.02	0.02	0.01	0.01	0.01	0.01	0.01

Size\Check	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	Overall (Norm.)
0-50	1.00	1.00	0.86	0.86	0.80	0.75	0.75	0.75	0.71	0.67	0.67	0.67	0.60	0.60	0.57	0.50	0.40	0.25	0.20	0.00	0.00	0.00	NA	NA	NA	NA	0.37
50-100	0.33	0.00	1.00	0.50	NA	1.00	0.80	0.60	1.00	0.86	0.33	0.33	0.50	0.40	0.33	0.50	0.60	0.33	0.33	0.60	0.50	0.00	0.83	0.71	0.43	0.29	0.35
Overall (Norm.)	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.01	0.02	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.01	0.00	0.72

Fig. 4. Performance of the proposer LLM (top) and the ranker LLM (bottom) stages in CORE, on CQPyUS, by static checks and prompt size needed (0th-50th percentile and 50th-100th percentile bins). The last rows (columns) of the tables are row (column) sums normalized by total number of files (520 for the top table and 453 for the bottom table). "NA" indicates that the cell did not have any files. Appendices A, B in the supplementary material give the mapping from the column headings (numbers assigned to the checks) to names of the CodeQL static checks.

Multiple observations are in order from the two tables presented in the figure: 1) there are some checks that are easier to resolve than others, no matter the prompt size. For instance, "Duplicate key in `dict` literal" check (number 4 in the top table, see Appendix A in the supplementary material) is resolved by simply updating the associated `dict` variable[5] whose precise location is given by the CodeQL error message that we include in the prompt in p_5 as discussed in Section 3.1; 2) in case of the Proposer LLM (top), the larger prompt size needed (i.e., 50th-100th percentile bin) results in reduction in performance for 19 out of 52 checks; the bin-wise average performance of CORE (last column) is 42% in the larger bin compared to 45% in the smaller bin; 3) in case of the Ranker LLM (bottom), the larger prompt size needed (i.e., 50th-100th percentile bin) results in reduction in performance for 26 out of 52 checks.

(ii) Common failure modes: Figure 5 shows the histogram of the failure modes of CORE, i.e., the reasons given by the human reviewers in our user study presented in Section 5.2 for rejecting the candidate revisions. From Table 2, we have that 1321 revisions were rejected out of the 2397 revisions produced by CORE. We also requested users to provide a terse reason for their decision, whenever they chose to reject a revision. The users provided reasons for 1108 (about 84%) of the rejected 1321 revisions. The most common failure mode observed in Figure 5 is introducing semantic changes, unrelated to the issue of interest, that could potentially impact the functional correctness of the code (e.g., changing the definition of `__hash__` while revising the implementation of `__eq__` for the Eq-Not-Overridden issue). Many of the presented failure modes align with the prevailing wisdom on the pitfalls of LLMs, including 1) hallucinations (such as creating superfluous classes and methods, rewriting exception handler messages, etc.), and 2) going above and beyond the given brief (via the prompt) to make unnecessary additions or deletions or edits. For instance, we find that in about 7% of the cases, LLMs try and fix seeming semantic inconsistencies in the source file — such as renaming variables consistently, adding or removing logging statements in certain code branches, etc. The user study also underscores the unreliability of static tools even for the quality checks, let alone functional correctness — in about 8.94% of the cases, fixes in the revised code were incorrect, and in about 1.44% of the cases, the fixes were incomplete, yet those revisions passed the associated CodeQL checks.

[5] https://codeql.github.com/codeql-query-help/python/py-duplicate-key-dict-literal

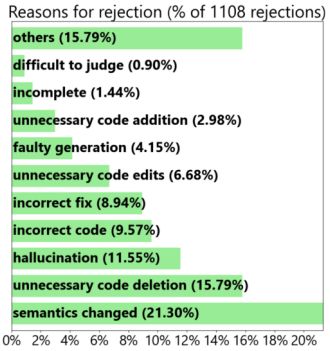

Fig. 5. Reasons given by reviewers for rejecting the candidate revisions produced by CORE on CQPyUS.

Table 4. **Ablation of Proposer LLM**: Comparing GPT-3.5-Turbo and WizardCoder-15B as the Proposer LLM in the CORE pipeline over a subset of CQPy consisting of small files fitting within a prompt budget of 1000 tokens. The first row gives the dataset statistics. The number of candidate revisions sampled per file is denoted by n.

	#Files (%)	#Issues remaining (%)
Flagged	224 (100%)	417 (100%)
WizardCoder-15B ($n = 10$)	192 (85.71%)	88 (21.10%)
GPT-3.5-Turbo ($n = 10$)	215 (95.98%)	15 (3.60%)
WizardCoder-15B ($n = 1$)	133 (59.37%)	171 (41.01%)
GPT-3.5-Turbo ($n = 1$)	186 (83.03%)	56 (12.71%)

Our qualitative analysis reveals that size of the file affects the performance of CORE on some static checks more than others. Close scrutiny of the feedback from human reviewers shows that many erroneous revisions come from hallucinations and the propensity of the LLMs to make unrelated changes. These insights should be useful to improve CORE and other similar efforts.

5.6 RQ6: What is the effect of using a *less powerful LLM* in the CORE pipeline?

We perform ablative study with a less powerful LLM for generating candidate revisions in the CORE pipeline. For this study, we use a recent open-source coding model WizardCoder-15B [42] [6], an order of magnitude smaller in size, but competitive on widely-used coding benchmarks such as HumanEval[7], compared to GPT-3.5-Turbo that we have used as the Proposer LLM in our results thus far. The only other change in CORE for this ablation is that we encapsulate the prompt template described in Section 3.1 in a more general format that WizardCoder-15B is trained on, as recommended by its authors [42]. We do this study on a smaller subset of CQPy where less powerful models are likely to succeed — we consider source files that are smaller in size, that fit within a needed budget of 1000 tokens for the total prompt size. We sample up to 5 such files per each of the 52 static checks in the dataset, yielding 224 flagged source files for this study.

In Table 4, we compare the performance of CORE pipeline (by the end of stage ❹) on this dataset with WizardCoder-15B or GPT-3.5-Turbo as the Proposer LLM in stage ❸. There are 417 issue violations in the 224 flagged files in total. From the second and the third rows, we see that WizardCoder-15B is fairly competitive to GPT-3.5-Turbo, and resolves over 85% of the files, compared to about 96% by GPT-3.5-Turbo. The instruction-following ability of WizardCoder-15B has been documented for various benchmarks [42], and through our study, we find promise for code reasoning and issue resolution as well.

Finally, in the last two rows of the table, we ablate on the number of revisions per file used in CORE. If we sample only one (with temperature = 0, as given in Section 4) candidate revision per file instead of 10 (which is the default in CORE), we see that GPT-3.5-Turbo still resolves over 83% of files while WizardCoder-15B is relatively poorer at about 59%. While the gap is clear, these results are promising in general — we can consider deploying substantially smaller models as proposer in CORE without compromising much on the performance.

[6] https://huggingface.co/WizardLM/WizardCoder-15B-V1.0
[7] https://github.com/openai/human-eval

A recent trend on program synthesis benchmarks shows that smaller yet powerful LMs can be competitive to LLMs. Our study (on a relatively simpler subset of CQPy) gives initial evidence that this may also apply to the different task of fixing code quality issues that we address in this paper.

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.

While the input code in our datasets may have been part of LLM training, it is highly unlikely for the LLMs to have seen the prompts constructed by us paired with expected code revisions. We analyzed 3535 Python repos from the CQPy dataset and found that less than 5% of the repos have CodeQL in their GitHub workflows. Thus, it is unlikely that the CodeQL warnings in CQPy were addressed in the repos, and so, unlikely that the LLM training included the fixes.

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 CodeQL and Sorald 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 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[8]. 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. From Figure 5, we see that less than 1% of the cases analyzed by the reviewers were difficult to judge for them, and those revisions were rejected.

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 [24, 26, 44].

7.1 Repairing static check violations

Among the approaches that target static analysis errors, [20, 27, 53, 58] use manually designed symbolic program transformations to fix specific classes of properties like heap safety [58], security

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

vulnerabilities [27], static quality checks [53] or data races [20]. Other approaches [13, 15, 37, 38, 43, 50] mine symbolic patterns from commit data to learn repair strategies or learn them from synthetically generated data [28]. For instance, SpongeBugs [43] uses SonarQube [8] to find bugs and commit data to create paired dataset. Similarly, Avatar [37, 38] and Phoenix [15] use FindBugs [4] and commit data. Revisar [50] mines edits from commit data for PMD [7]. GetAFix [13] uses Infer [5] and Error Prone [10], 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 [30, 56, 62, 64] is to train neural models to map buggy programs to their fixed versions. The 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 [60] 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 [61] 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 [22, 29, 40]. These approaches aim at fixing bugs identified by failing test cases. In comparison, our work addresses a related but different problem of fixing errors flagged by static analysis tools.

Prompting techniques: RING [32] 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 [31] 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. [48] 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 [48] 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.

7.3 Automating code reviews

A related line of research is automating code review activities [25, 34–36, 57]. However, this line of research has important differences from the focus of our work. While humans can give feedback on code quality, the common best practices are often distilled as automated checks. We focus on the latter and take advantage of the automated tools already in the engineering pipelines to automatically validate the LLM-generated fixes. This prevents unnecessary back-and-forth between the reviewer and the code author for the same. Automated code review tools aim to assist human-raised issues and help reduce load for reviewers. They exist alongside or after our tools in a typical software development and management pipeline; and they work in tandem between reviewer and author.

7.4 LLMs as verifiers

The issue of plausible but incorrect fixes is well-known [41, 49]. 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 [14, 33], 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. [45] 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 [65] 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 functional correctness. The code generation datasets studied in [65] 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 [65] 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 based on reinforcement learning and human or tool feedback [19, 47, 52, 54].

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] [n. d.]. Sorald Tool Source. https://github.com/ASSERT-KTH/sorald/releases/tag/sorald-0.8.5.
[10] 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.
[11] 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).
[12] 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.
[13] 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
[14] 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).
[15] 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
[16] 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.
[17] 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).
[18] 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).
[19] 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).
[20] 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.
[21] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[22] 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).
[23] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.
[25] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022. CommentFinder: a simpler, faster, more accurate code review comments recommendation *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 507–519. https://doi.org/10.1145/3540250.3549119
[26] 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).
[27] 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

[28] 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]

[29] 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.

[30] 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.

[31] 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).

[32] 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).

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

[34] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. CodeEditor: Learning to Edit Source Code with Pre-trained Models. *ACM Transactions on Software Engineering and Methodology* 32, 6 (Sept. 2023), 1–22. https://doi.org/10.1145/3597207

[35] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. AUGER: Automatically Generating Review Comments with Pre-training Models. arXiv:2208.08014 [cs.SE]

[36] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-training. arXiv:2203.09095 [cs.SE]

[37] 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

[38] 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

[39] 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.

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

[41] 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.

[42] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL]

[43] 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

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

[45] 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]

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

[47] 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.

[48] 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.

[49] 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.

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

[51] Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and Shirish Shevade. 2024. CodeQueries: A Dataset of Semantic Queries over Code. In *17th Innovations in Software Engineering Conference*.

ACM.
[52] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI]
[53] 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).
[54] 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.
[55] 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).
[56] 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.
[57] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. arXiv:2201.06850 [cs.SE]
[58] 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
[59] 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.
[60] 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.*
[61] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).
[62] 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.
[63] 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
[64] 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.
[65] Terry Yue Zhuo. 2023. Large Language Models Are State-of-the-Art Evaluators of Code Generation. arXiv:2304.14317 [cs.AI]

Received 2023-09-29; accepted 2024-01-23