Write Your Own Code Checker: An Automated Test-Driven Checker Development Approach with LLMs

Jun Liu*[†]
School of Advanced Interdisciplinary Sciences, University of Chinese Academy of Sciences
Beijing, China
liuj2022@ios.ac.cn

Yuanyuan Xie*
Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences
Hangzhou, China
xieyy@ios.ac.cn

Jiwei Yan[‡]
Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences (ISCAS)
Beijing, China
yanjiwei@otcaix.iscas.ac.cn

Jinhao Huang
Technology Center of Software Engineering, ISCAS
Beijing, China
huangjinhao@otcaix.iscas.ac.cn

Jun Yan[‡][†]
University of Chinese Academy of Sciences
Beijing, China
yanjun@ios.ac.cn

Jian Zhang[†]
University of Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

Abstract

With the rising demand for code quality assurance, developers are not only utilizing existing static code checkers but also seeking custom checkers to satisfy their specific needs. Nowadays, various code-checking frameworks provide extensive checker customization interfaces to meet this need. However, both the complex checking logic of rules and massive API usages of large-scale checker frameworks make this task challenging. To this end, automated code checker generation is anticipated to ease the burden of checker development. In this paper, we propose AutoChecker, an innovative LLM-powered approach that can write code checkers automatically based on only a rule description and a test suite. To achieve comprehensive checking logic, AutoChecker incrementally updates the checker's logic by focusing on solving one selected case each time. To obtain precise API knowledge, during each iteration, it leverages fine-grained logic-guided API-context retrieval, where it first decomposes the checking logic into a series of sub-operations and then retrieves checker-related API-contexts for each sub-operation. For evaluation, we apply AutoChecker, five baselines, and three ablation methods using multiple LLMs to generate checkers for 20 randomly selected PMD rules. Experimental results show that AutoChecker significantly outperforms others across all effectiveness metrics, with an average test pass rate of 82.28%. Additionally, checkers generated by AutoChecker show performance comparable to that of the official checkers when applied to real-world projects.

*Both authors contributed equally to this research.
[†]Also with Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, ISCAS.
[‡]Corresponding authors.

CCS Concepts

• **Software and its engineering → Software safety**; *Automatic programming*.

Keywords

Checker Generation, API Retrieval, Test-driven Development, LLMs, static analysis

ACM Reference Format:
Jun Liu, Yuanyuan Xie, Jiwei Yan, Jinhao Huang, Jun Yan, and Jian Zhang. 2026. Write Your Own Code Checker: An Automated Test-Driven Checker Development Approach with LLMs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764560

1 Introduction

Static code-checking tools play a crucial role in ensuring code quality by generating security reports based on a set of predefined rules. In practice, users often need to customize checkers to meet specific requirements [29]. Recent studies [37, 52, 60] also emphasize the importance of tailoring code-checking tools to specific contexts, such as individual projects and security scenarios. For example, a survey of experienced developers [60] found that up to one-third of participants highlighted the need for project-specific rules. Thus, customizing static code checkers is important for quality assurance.

To meet this demand, many static analysis tools support custom checkers. For instance, PMD [9] and SonarQube [11] allow users to write custom checkers in Java, while CodeQL [2] and other DSL-based tools [67] support custom queries in DSL formats. However, creating custom checkers remains a significant challenge. An empirical study [25] highlights this, revealing that only 8% of developers write them in practice. This difficulty stems from several inherent obstacles: the high complexity of checking frameworks [20] (e.g., PMD's framework alone exceeds 30 KLOC), the need for massive framework-specific API knowledge, incomplete or unclear API documentation, and the non-trivial checking logic. These barriers make checker customization time-consuming and difficult, especially for users with urgent needs but limited tool familiarity.

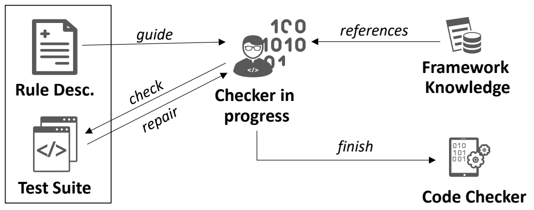


Figure 1: Pipeline of the Manual Checker Development

Recently, the booming of Large Language Models (LLMs) has significantly advanced automatic code generation [50, 61, 63]. Inspired by this, we explore leveraging LLMs to auto-generate checker code, aiming to alleviate the burden on developers in writing custom checkers. Notably, some recent studies have combined LLMs with static checking tools for security issue detection. Specifically, some studies [44, 62] leverage LLMs to infer source-sink specifications for specific projects and CWEs, while others [22, 41, 42] use LLMs to filter false positives reported by static checkers. However, these works focus on enhancing existing checkers rather than creating new ones for specific requirements. As far as we know, we are the first to automate custom checker development from scratch.

Checker generation is more challenging and distinct from typical code-generation tasks. Fig. 1 illustrates the manual checker development process. When a custom checker is needed, the project manager provides an overall rule description for the rough goal and an adequate test suite for validation. Here, each test case refers to a rule-specific validation code that demonstrates compliance/violation of a particular rule. Developers then interpret the rule description and test suite to derive the correct checking logic and implement it using framework APIs based on their knowledge. Unlike typical code generation tasks, which often involve clear and common targets (e.g., algorithms or function implementations) [47, 63], automated checker generation is significantly more difficult due to the intricacy of the checking logic and the scale of the required framework APIs. Specifically, we have to cope with two main challenges:

C1: **Generating comprehensive checking logic covering diverse scenarios.** When using LLMs to generate the comprehensive checking logic, both the rule description (for the overall goal) and test suite (covering diverse scenarios) should be included as input. However, as the number of checking scenarios grows, the input volume can become excessive. This not only challenges the LLM's ability to synthesize coherent logic across all cases but also risks exceeding the model's token limit. Thus, the comprehensive checking logic is hard to generate at once.

C2: **Retrieving precise API knowledge from high-level rule descriptions.** Developing code checkers requires a deep understanding of the framework's APIs. However, with thousands of APIs, identifying the precise ones for a specific checker is challenging. A common approach is to retrieve relevant APIs based on the rule description. However, this often fails due to the *granularity mismatch* between high-level rule descriptions and specific API functionalities. This discrepancy makes precise API retrieval difficult, as also shown by the results of the Retrieval Augmented Generation (RAG) baseline in Section 4.2.

To address above challenges, we propose **AutoChecker**, a novel approach to automatically generate static checkers from rule descriptions and test suites. First, to cover diverse scenarios, we mimic the manual checker development process (Fig. 1), where developers iteratively validate and refine the checker against a test suite. We introduce the Test-Driven Checker Development (TDCD) approach, enabling AutoChecker to refine the checker case by case, incrementally building comprehensive checking logic (***C1***). Second, to retrieve precise API knowledge, AutoChecker employs **Logic-guided API-context Retrieval** (***C2***). Unlike common RAG approaches that simply use the rule descriptions as queries, AutoChecker decomposes the checking logic into discrete sub-operations and respectively retrieves corresponding API contexts from two specialized databases: *Meta-API DB* (semi-automatically built) and *Full-API DB* (automatically built). This fine-grained method retrieves precise API knowledge on the sub-op level for accurate checker generation.

In this paper, we implement AutoChecker on PMD [9], a widely-used static analysis tool[1]. To evaluate AutoChecker, we randomly select 20 PMD built-in Java rules (10 easy and 10 hard). Experimental results show that our approach outperforms baselines across all metrics. Specifically, AutoChecker-generated checkers achieve an average test pass rate of 82.28% (84.70% for easy rules and 79.86% for hard ones), which is 2.93× and 2.11× higher than the simplest baseline NoCaseLLM and the best baseline NoCaseLLM^{RC}, respectively. Also, we further evaluate practicality by applying AutoChecker-generated checkers (that pass all tests) to five large-scale Java projects. The results show that AutoChecker can write checkers performing equivalently to official ones when sufficient test cases are provided. We conclude our main contributions as follows:

- We propose an automated test-driven checker development approach (TDCD), which uses an iterative generation pipeline to cope with the complex checking logic case by case.
- We develop a logic-guided API-context retrieval strategy and design a general Meta-Op set for fine-grained and precise API retrieval, which contains 354 atomic checking operations.
- We implement our approach into AutoChecker, which automatically develops custom code checkers based on the given rule and test suite. Experimental results show that AutoChecker outperforms baseline methods across all effectiveness metrics. Compare to official checkers, they also achieve expected results on real-world, large-scale projects.

Both the code and the dataset of AutoChecker are available at https://github.com/SQUARE-RG/AutoChecker. To demonstrate intermediate LLM-generated checkers and results in each step of the checker-development cycle, we also provide a replay website for visualization at https://autochecker.maskeduser.party.

2 Background and Motivation

In this section, we first briefly introduce the background of custom static code checkers, with a focus on the specific type (AST-based checkers) targeted in this paper. Then, we illustrate the challenges and our proposed solutions through a motivating example.

[1]AutoChecker can be readily adapted to other AST-based tools and programming languages with manageable human effort, which is further discussed in Section 5.

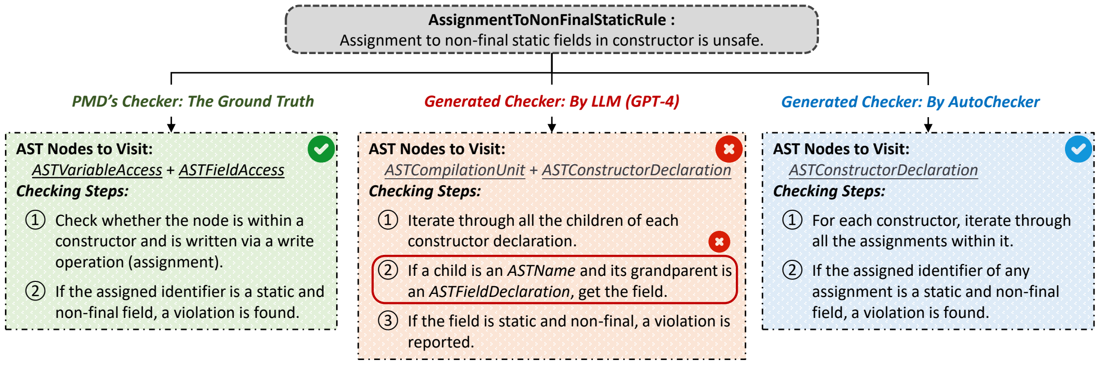

AssignmentToNonFinalStaticRule :
Assignment to non-final static fields in constructor is unsafe.
PMD's Checker: The Ground Truth
Generated Checker: By LLM (GPT-4)
Generated Checker: By AutoChecker
AST Nodes to Visit:
ASTVariableAccess + ASTFieldAccess
Checking Steps:
① Check whether the node is within a constructor and is written via a write operation (assignment).
② If the assigned identifier is a static and non-final field, a violation is found.
AST Nodes to Visit:
ASTCompilationUnit + ASTConstructorDeclaration
Checking Steps:
① Iterate through all the children of each constructor declaration.
② If a child is an ASTName and its grandparent is an ASTFieldDeclaration, get the field.
③ If the field is static and non-final, a violation is reported.
AST Nodes to Visit:
ASTConstructorDeclaration
Checking Steps:
① For each constructor, iterate through all the assignments within it.
② If the assigned identifier of any assignment is a static and non-final field, a violation is found.

Figure 2: A motivating example showing the concrete steps of the ground truth and auto-generated checkers for *AssignmentToNonFinalStaticRule*. Specifically, the logic of the checker (as checking steps) generated directly by LLM is incomplete.

2.1 Custom Static Code Checker

Static code checkers are designed in static analysis tools to analyze code without executing it [16, 24, 56]. Many existing tools, such as PMD [9], SonarQube [11], and CodeQL [2], support custom code checkers. These custom checkers can be broadly categorized into two groups based on their analysis workflow: **AST-based** (by traversing Abstract Syntax Tree [18]), and **flow-based** (by traversing control- and data-flow). Flow-based checkers are heavyweight, and their customization typically involves enhancing specifications on predefined data-tracking patterns [44, 62]. Compared to them, AST-based checkers are more lightweight with a straightforward checking process: traverse the AST of the target code, apply checking rules to relevant AST nodes, and report potential issues when a match is found. So, they are easier to customize. To meet new customization demands, experienced developers can write new AST-based checkers from scratch (as shown in Fig. 1). These advantages also make AST-based checkers a preferred choice for software companies in quality assurance. Therefore, this paper specifically focuses on automating the development of AST-based checkers.

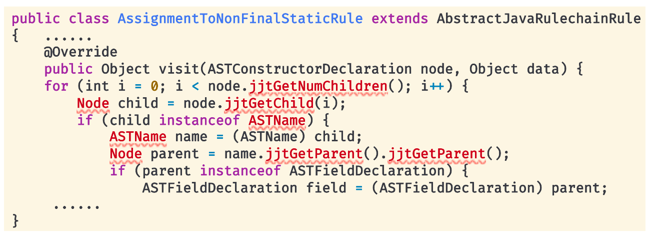
2.2 Motivating Example

PMD [9] is a popular static checking tool supporting 18 programming languages (primarily Java and Apex) with over 400 built-in rules. We use a PMD Java rule, *AssignmentToNonFinalStaticRule*, as a motivating example. Its description states: *"Assignment to non-final static fields in constructors is unsafe."* The corresponding checker should report all unsafe assignments described by the rule.

First, we prompt multiple LLMs (GPT-4 [5], DeepSeek-V3 [45], etc.) to generate checkers for this rule by providing its description and full test suite. However, all generated checkers fail due to incomplete logic and compilation errors caused by hallucinated APIs. This highlights two key challenges in automated checker generation: (1) generating comprehensive checking logic (at the **Abstract Level**), and (2) invoking correct framework APIs (at the **Implementation Level**). Below, we detail the results from GPT-4.

At the **Abstract Level**, we compare the checking steps from implementations of the LLM-generated checker and the ground truth. As shown in Fig. 2, the ground truth checker locates variable and field accesses within constructors and verifies if the referenced symbols are static and non-final. In contrast, the LLM-generated checker identifies unsafe fields starting from constructor declarations but only checks fields in `ASTFieldDeclaration`, missing unsafe fields in re-assignment expressions, resulting in incomplete logic. Despite providing sufficient test cases, the LLM struggles to generate comprehensive logic due to information overload from presenting many test cases at once. To address this, AutoChecker introduces test-driven checker development, refining the checker's logic case by case. As shown in Fig. 2, AutoChecker resolves the soundness issue by examining all assignment expressions within constructors, producing correct checking logic from a unique perspective compared to the ground truth.

At the **Implementation Level**, we analyze the LLM-generated checkers' code. As shown in Fig. 3, when directly prompted to write a checker, the LLM often guesses framework APIs, leading to hallucinations like undefined method `jjtGetNumChildren` and class `ASTName`. Specifically, **41.7% (5 out of 12)** of the APIs used are hallucinated, causing compilation errors. To address this, we then follow the common RAG pipeline [30, 40], retrieving framework APIs using the rule description as a query. However, due to the granularity mismatch between the high-level rule description and specific API functionality, **29.4% (5 out of 17)** of the APIs remain hallucinated. Finally, by introducing fine-grained logic-guided API retrieval, AutoChecker successfully generates a correct checker with **26** valid APIs, compiling and passing all tests. Notably, as API knowledge is provided, the number of APIs in the generated checker increases, as guessed APIs (often higher-level abstractions) are replaced with multiple concrete valid APIs.



```
public class AssignmentToNonFinalStaticRule extends AbstractJavaRulechainRule
{   ......
    @Override
    public Object visit(ASTConstructorDeclaration node, Object data) {
    for (int i = 0; i < node.jjtGetNumChildren(); i++) {
        Node child = node.jjtGetChild(i);
        if (child instanceof ASTName) {
            ASTName name = (ASTName) child;
            Node parent = name.jjtGetParent().jjtGetParent();
            if (parent instanceof ASTFieldDeclaration) {
                ASTFieldDeclaration field = (ASTFieldDeclaration) parent;
     ......
}
```

Figure 3: A snippet of the LLM-generated checker for *AssignmentToNonFinalStaticRule*, using the rule description and test suite as input, includes multiple hallucinated APIs.

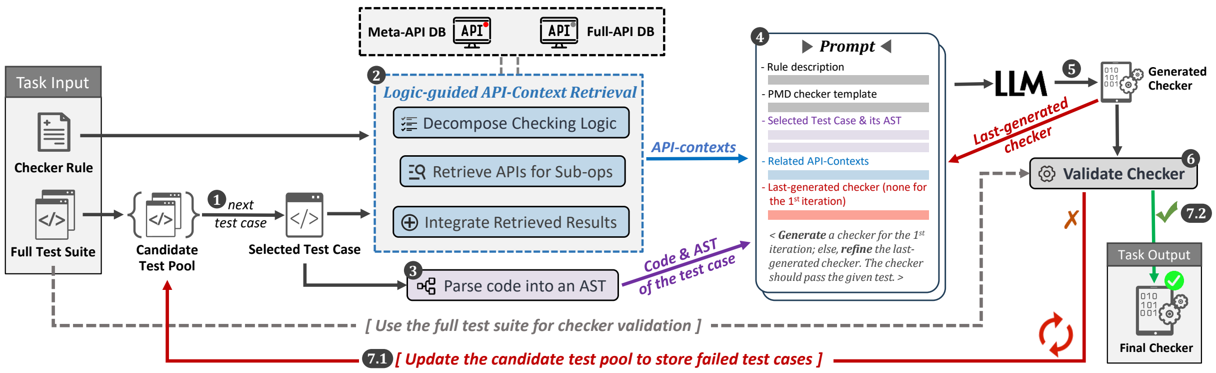

Meta-API DB
Full-API DB
Task Input
Checker Rule
Full Test Suite
Candidate Test Pool
1 next test case
Selected Test Case
2 Logic-guided API-Context Retrieval
Decompose Checking Logic
Retrieve APIs for Sub-ops
Integrate Retrieved Results
3 Parse code into an AST
API-contexts
Code & AST of the test case
4 Prompt
- Rule description
- PMD checker template
- Selected Test Case & its AST
- Related API-Contexts
- Last-generated checker (none for the 1st iteration)
< Generate a checker for the 1st iteration; else, refine the last-generated checker. The checker should pass the given test. >
LLM
5 Generated Checker
Last-generated checker
6 Validate Checker
7.2
Task Output
Final Checker
[Use the full test suite for checker validation]
7.1 [Update the candidate test pool to store failed test cases]

Figure 4: Overview of the LLM-powered Test-Driven Checker Development in AutoChecker

3 Methodology

This section presents the detailed methodology of our proposed AutoChecker. After showing the overall pipeline in Section 3.1, Section 3.2 and Section 3.3 introduce the API-context retrieval and checker development approaches in detail.

3.1 Overview

Given a checker rule and its full test suite, AutoChecker is designed to automatically generate the static checker following the **Test-Driven Checker Development (TDCD)** process. As shown in Fig. 4, AutoChecker generates and refines the checker case by case.

To start with, AutoChecker maintains a candidate test pool to store test cases that have not yet been verified or passed. During each round of TDCD, a single test case is selected from this pool 1. Using the selected test case and given checker rule, AutoChecker employs the **Logic-guided API-Context Retrieval** approach to collect relevant API-contexts 2. To ensure precision, AutoChecker breaks down the checking logic into fine-grained sub-operations using LLM and retrieves the corresponding APIs respectively. Additionally, to obtain the accurate AST-based information of the test case, AutoChecker utilizes a parser to get its AST 3.

After preparing all the necessary input information, AutoChecker constructs the checker-generation prompt 4, which consists of the *rule description*, *PMD checker template*, *selected test case (both source code and AST)*, *related API-contexts* and *last-generated checker (not for the first round)*. By passing on the prompt to the LLM, a checker is generated for this round 5. To verify whether the generated checker is correct, it will then be validated with the full test suite 6. If the checker fails to pass all tests, AutoChecker will update the candidate test pool to keep all the failed test cases and start the next iteration 7.1. Otherwise, once the generated checker passes all tests or reaches a test-passing bottleneck, AutoChecker will terminate the TDCD process and output the final checker 7.2.

3.2 Logic-guided API-Context Retrieval

As shown in Fig. 4, API-context Retrieval serves as a crucial module within the TDCD process, which is designed to provide accurate and sufficient API knowledge for checker generation. Inspired by Chain-of-Thought [43, 65] and Compositional API Recommendation [51], we propose a fine-grained Logic-guided API-Context Retrieval approach. Specifically, AutoChecker first uses the LLM to decompose the checker rule into a checking skeleton with sub-operations. Then, each sub-operation is used for individual API-context (API signatures and usages) retrieval and finally makes up the whole API-contexts. In this section, we sequentially explain the Logic-guided API-Context Retrieval approach in three parts: API Collection, Database Construction, and the Retrieval Process.

3.2.1 *Framework API Collection.*

In general, framework APIs for AST-based checkers can fall into the following three categories:

- **Node-related APIs** perform concrete operations for specific AST nodes, e.g., obtaining the name of a method, etc.
- **Edge-related APIs** deal with connections and transitions between nodes, e.g., finding the closest parent AST node, etc.
- **Util-related APIs** offer utility functions that can be invoked anywhere, e.g., checking whether a type is abstract, etc.

In PMD specifically, APIs are defined in AST Node Classes (e.g., `ASTMethodDeclaration`) and Utility Classes (e.g., `JavaAstUtils`). Thus, we identify node- and edge-related APIs from AST Node Classes, while Util-related ones are collected from Utility Classes.

☞ **Collecting Node-related and Edge-related APIs from AST Node Classes**. First, we map each AST Node Class (ANC) to its available APIs, including methods declared within the class and those inherited from its superclasses. Among all APIs, edge-related APIs, which handle general node-traversal functions, are primarily defined in the abstract ANC, `JavaNode`. From the available APIs of `JavaNode`, we identify edge-related APIs as those whose return value is another node. After filtering out these edge-related APIs, the remaining ones are categorized as node-related APIs.

☞ **Collecting Util-related APIs from Utility Classes**. Each util-related API is a static method within a utility class characterized by a final modifier and a private constructor. By searching all the utility classes, we collect the util-related APIs.

Overall, the number of collected Java-checking framework APIs of PMD in each type is shown in Tab. 1. The significant number of APIs (over 11k) also underscores the necessity of precise retrieval.

Table 1: PMD's Framework APIs of Each Type

API Type	Collect From	Number
Node-related APIs	Concrete ANCs	11,243
Edge-related APIs	Abstract ANC	21
Util-related APIs	Utility Classes	377

Table 2: Descriptive Text Generation for All Types of APIs

API Type	Return Type	Descriptive Text (*prefix+basic phrase+comments*)
Node, Edge	`Boolean`	Check whether [className]^s [methodName]^s //cmt.
Util	`Boolean`	Check whether [methodName]^s //cmt.
Node, Edge	`non-Boolean`	[methodName]^s of [className]^s //cmt.
Util	`non-Boolean`	[methodName]^s //cmt.

^{s} denotes splitting the name into individual words according to the CamelCase rule. cmt. denotes the comments of each API for simplicity.

3.2.2 API-Context Database Construction. Based on the collected framework APIs, we construct two API-context databases: **Full-API DB** and **Meta-API DB**. An API-context is defined as either an API's signature or usage snippet, paired with descriptive text (used as query for semantic retrieval). The Meta-API DB is built based on the Full-API DB, according to a crafted Meta-Op Set. We explain the process in three steps: *Full-API DB Construction*, *Meta-Op Set Preparation*, and *Meta-API DB Construction*.

☞ **Full-API DB Construction**. The Full-API DB is constructed using all three types of APIs. To generate the descriptive text for each API, we leverage the semantic information embedded in its signature. As demonstrated in Tab. 2, each descriptive text consists of three parts: the prefix, basic phrase, and comments.

First, we determine the ***prefix*** of the descriptive text based on the API's return type. For an API with a `Boolean` return type, used for judgment, we add "*check whether*" as the prefix of the descriptive text. For an API with a `non-Boolean` return type (e.g., `String`), used for data acquisition, the method name usually starts with an action word like "*get*", so no additional prefix is required.

Then, we generate the ***basic phrase*** based on the API's class and method names. Specifically, we split these names into individual words based on the *CamelCase* naming rule and remove unnecessary or repetitive terms (e.g., AST). For example, the class `ASTStringLiteral` yields the basic phrase "*String Literal*", while the method `isEmpty` produces "*is empty*". Notably, for util-related APIs, class names (e.g., `JavaAstUtil`) are typically omitted, as they often lack relevance to the API's concrete functionality.

To enhance the descriptive text, we also extract ***comments*** (docs) of the APIs and append them to the end of the description text, prefixed with "//". Irrelevant comments, such as those related to exceptional conditions or authorship, are filtered out.

Finally, the prefix, basic phrase, and comments are combined to form the descriptive text of each API. Based on that, we construct the Full-API DB, where each element is a **description-signature** pair with the descriptive text and signature of an API. Fig.5 gives an example element for the API `isEmpty` in the Full-API DB.

➽ **Description-Signature Pair:**
Description (*descriptive text*): "Check whether string literal is empty"
API–context (*API signature*):
"net.sourceforge.pmd.lang.java.ast.ASTStringLiteral:
public java.lang.Boolean isEmpty() //True if the constant value is empty."

Figure 5: An Example Element in Full-API DB

When using the Full-API DB for retrieval, we focus retrieval efforts on node- and util-related APIs and directly include all the edge-related API-contexts to the retrieved result. Edge-related APIs, which provide AST-traversing functions, are usually limited in number (21 for PMD as shown in Tab. 1) but fundamental. Thus, we treat them as essential information to be provided by default.

☞ **Meta-Op Set Preparation**. For real-world scenarios, framework APIs vary widely in encapsulation granularity, both within and across frameworks. This inconsistency makes it hard to reliably find the correct APIs solely based on the Full-API DB, which may lead to mismatches or overlaps. Thus, we need a more standardized API-context database (Meta-API DB). To meet this, we propose an abstraction layer, the Meta-Operation Set (Meta-Op Set), designed to unify API-context granularity across frameworks.

Specifically, the Meta-Op Set contains meta-operations (meta-ops) with basic functionalities commonly used for code-checking tasks. To get a comprehensive Meta-Op Set, we invited three developers with more than two years of checker-development experience for the collection. The first developer collected and organized most meta-ops into categories according to their experience across various checking frameworks (mainly based on PMD and CodeQL), and the other two brainstormed to supplement them. As shown in Fig. 6, the Meta-Op Set contains **354** meta-ops in **14** categories. We have open-sourced the Meta-Op Set in our project repository.

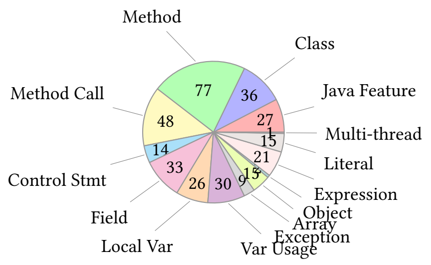


Figure 6: Category of Operations in the Meta-Op Set

☞ **Meta-API DB Construction**. Using the Meta-Op Set as a foundation, we construct the Meta-API Database (Meta-API DB), where each entry pairs a meta-operation (meta-op) with its corresponding API-context (either API signature or usage snippet).

For each meta-op, we first search the Full-API DB to identify API descriptions that semantically align with the meta-op's functionality. Once a match is found, we extract the associated API signature as the API-context for that meta-op. Otherwise, if no API descriptions match the given meta-op, we manually craft an implementation code snippet to fulfill the meta-op's functionality as its API-context. Overall, the API-contexts in Meta-API DB are in the form of **operation-signature** pairs and **operation-snippet** pairs. We provide two examples in Fig. 7.

➽ **Operation-Signature Pair:**
Meta–op: "Get the name of class"; Category: "Class".
API–context (*API signature*):
"net.sourceforge.pmd.lang.java.ast.ASTClassOrInterfaceDeclaration:
public java.lang.String getSimpleName()"

➽ **Operation-Snippet Pair:**
Meta–op: "Check whether the return type of method is int"; Category: "Method".
API–context (*code snippet*):
```
"import net.sourceforge.pmd.lang.java.ast.ASTMethodDeclaration;
import net.sourceforge.pmd.lang.java.types.JPrimitiveType;
public boolean isReturnValueIntType(ASTMethodDeclaration m) {
   return m.getResultTypeNode().getTypeMirror()
              .isPrimitive(JPrimitiveType.PrimitiveTypeKind.INT);
}"
```

Figure 7: Example Elements in Meta-API DB

3.2.3 *API-Context Retrieval Process.*

With the constructed DBs, AutoChecker retrieves related API-contexts based on the checker rule and a given test case. To start with, all 21 edge-related API-contexts from the Full-API-DB are directly added to collected API-contexts, as mentioned in Section 3.2.2. Then, AutoChecker leverages the Logic-guided API-context Retrieval approach to retrieve related node- and util-related API-contexts, which is shown in Fig. 8.

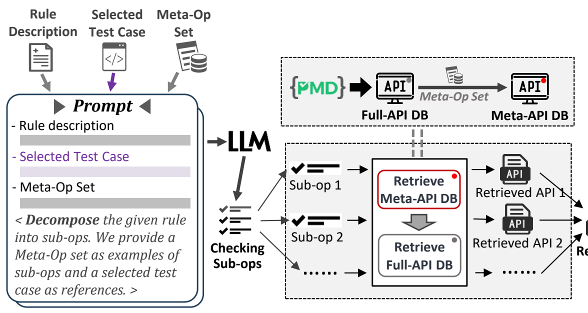


Figure 8: Pipeline of the Logic-guided API-context Retrieval

First, AutoChecker decomposes the checker rule into a series of sub-operations (sub-ops) as the checking skeleton. Given the checker rule, test case and Meta-Op Set as inputs, AutoChecker leverages the LLM to make the split. Specifically, the Meta-Op Set serves as references to sub-ops, which guides the LLM to generate sub-operations under the similar granularity of meta-ops. The overall decomposition prompt is also demonstrated in Fig. 8.

Next, AutoChecker fetches API-contexts for each sub-op using both the Meta-API and Full-API DBs. During each retrieval process, the sub-op serves as the query to find the API-context with the highest semantic similarity score. If the score falls below a set threshold, the retrieval fails and returns `None`. AutoChecker first queries the Meta-API DB. If unsuccessful, it then searches the Full-API DB. Note that, before querying the Full-API DB, AutoChecker filters out irrelevant node-related APIs for higher precision and efficiency. Here, APIs defined in AST node classes that don't appear in the test case's AST are deemed irrelevant. Finally, all relevant API-contexts, both foundational and retrieved, are gathered.

3.3 Test-Driven Checker Development

This section focuses on the technical details of the TDCD process. While Figure 4 shows the high-level pipeline, Algorithm 1 presents the detailed, step-by-step implementation of the methodology.

3.3.1 *Prompt Settings.*

In each round of TDCD, AutoChecker writes a checker based on a selected test and the checker rule. There are two types of prompts in TDCD: one for initial checker generation and the other for iterative checker refinement.

☞ **Prompt for Initial Generation**. In the 1^{st} round, the prompt instructs the LLM to generate a rule-specific checker capable of passing the provided test using the following input on line 11.

★ ***Rule description***. It is derived from the original input.

★ ***Test case code***. It is picked from the candidate test pool (line 5).

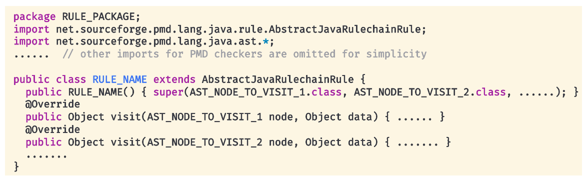
★ ***Test case AST***. Since AST information is crucial for AST-based checking, AutoChecker extracts the test's AST using PMD's built-in parser (line 7). To clearly link AST nodes to their source code, AutoChecker also retains the concrete names of AST nodes parsed from identifiers. For instance, the AST node `ASTClassDeclaration` parsed from the method name "length" is augmented as "ASTMethodDeclaration(length)".

★ ***Related API-contexts***. AutoChecker adopts the API-Context Retrieval to retrieve related API-contexts based on the checker rule and cleaned test case on line 6, introduced in Sec. 3.2.

★ ***Checker template***. We manually summarize a PMD checker template from existing checkers, which is shown in Fig. 9.

Algorithm 1 Algorithm of Test-Driven Checker Development (TDCD)

```
Input: r: the checker rule description, T_a: the full test suite with all tests
Output: c_f: the final checker, pr_f: the test pass rate for the final checker
 1: Load the checker template C̄
 2: T_c ← T_a, c ← None          ▷ initialize the candidate test pool T_c and checker c
 3: T_p ← {}, T_s ← {}           ▷ record the passed tests in T_p and skipped tests in T_s
 4: while |T_c| > 0 do
 5:     t ← pickNextTest(T_c)
 6:     K_api ← retrieveAPIContexts(r, t)       ▷ use logic-guided API-context retrieval
 7:     ast ← parseAST(t)
 8:     j ← 0                                   ▷ the number of retries for t
 9:     while j < MAX_RETRY_TIMES do
10:         if c = None then
11:             c ← genInitialChecker(r, t, ast, C̄, K_api)          ▷ LLM-based generation
12:         else
13:             c ← refineLastChecker(r, t, ast, C̄, K_api, c)       ▷ LLM-based refinement
14:         end if
15:         rep ← validateChecker(c, T_a)                ▷ get the validation report
16:         if t ∈ rep.passedtests and rep.failedtests ∩ T_p = ∅ then
17:             break                  ▷ the checker passes t without regression errors
18:         end if
19:         j ← j + 1
20:     end while
21:     if j = MAX_RETRY_TIMES then
22:         T_s.add(t)                          ▷ skip t if it reaches the retry limit
23:     end if
24:     T_p ← rep.passedtests, T_c ← rep.failedtests \ T_s             ▷ update test sets
25: end while
26: c_f ← c, pr_f ← rep.pr                  ▷ return the final checker and test pass rate
```



```
package RULE_PACKAGE;
import net.sourceforge.pmd.lang.java.rule.AbstractJavaRulechainRule;
import net.sourceforge.pmd.lang.java.ast.*;
......  // other imports for PMD checkers are omitted for simplicity

public class RULE_NAME extends AbstractJavaRulechainRule {
  public RULE_NAME() { super(AST_NODE_TO_VISIT_1.class, AST_NODE_TO_VISIT_2.class, ......); }
  @Override
  public Object visit(AST_NODE_TO_VISIT_1 node, Object data) { ...... }
  @Override
  public Object visit(AST_NODE_TO_VISIT_2 node, Object data) { ....... }
  .......
}
```

Figure 9: Simplified PMD Checker Template

☞ **Prompt for Iterative Refinement**. In subsequent rounds, the prompt is designed for checker refinement. It instructs the LLM to refine a given rule-specific checker to pass the selected test case. Compared to the initial generation prompt, this one also includes the ★ ***last-generated checker*** as input.

Notably, after generating the checker using the above prompts, AutoChecker employs a simple strategy to prevent import errors. Specifically, it replaces the import section of the generated checker code with default imports, matching those in the template (Fig. 9). This ensures that all required packages are correctly imported.

3.3.2 *Checker Development Cycle.*

The TDCD cycle follows an iterative refinement process. Throughout the cycle, AutoChecker dynamically maintains three test sets as follows.

- T_c is the candidate test pool with unprocessed and failed tests.
- T_p is a test set that records all passed tests.
- T_s is a test set that records all skipped tests. In a single round, sometimes the LLM may fail to generate a checker that passes the given test case within allowed attempts, AutoChecker then skips this test to prevent blocking the cycle.

To start with, the cycle begins by initializing T_c with all tests from the full suite T_a. Then, AutoChecker selects a single test from T_c in each round of the cycle to guide the checker development process on lines 5-19. For each round, the generated checker will be validated with the full test suite on line 15. Note that AutoChecker ensures that each newly generated checker in every iteration should pass the given test case without affecting the already passed test cases (without regression errors). If not, AutoChecker will re-query the LLM to re-generate the checker within allowed retry attempts on lines 8-20. After validation, all test sets are updated on lines 21-24. Specifically, passed tests are moved to the T_p, while persistently failing tests (after maximum attempts) are added to T_s. Besides, T_c is updated with the failed tests, excluding skipped ones in T_s.

Finally, the cycle terminates when T_c becomes empty, indicating all tests have been either validated or skipped. The final checker c_f and its test pass rate pr_f are derived from the last validation results.

4 Evaluation

We conduct extensive experimental evaluations of AutoChecker to address the following research questions:

- **RQ1 (Effectiveness)**: Can AutoChecker effectively generate high-quality code checkers?
- **RQ2 (Ablation Study)**: How do different strategies contribute to AutoChecker's effectiveness?
- **RQ3 (Cost)**: Can AutoChecker develop checkers cost-effectively?
- **RQ4 (Practicality)**: How do AutoChecker-generated checkers perform on real-world projects?

4.1 Evaluation Setup

4.1.1 Implementation Settings. In this paper, we build AutoChecker specifically for *PMD*, an open-source AST-based code-checking tool known for its effectiveness and ease of use [46]. Specifically, we used the latest version `7.0.0-rc4` when we started our work.

We implemented AutoChecker on *LangChain* [7], a widely-used framework for LLM applications. For the API-context retrieval module, it employs vector databases with the SOTA open-source *bge-large-en-v1.5* [66] embedding model from BAAI [1]. Drawing from our experience and prior work [48, 71], we set similarity score thresholds to 0.85 for Meta-API matching and 0.8 for API-context searching. In the checker development cycle, we set `MAX_RETRY_TIMES` as 5 for each round of checker generation. Currently, AutoChecker supports two working modes: *writing checkers from scratch* and *incrementally*. In the incremental mode, developers can enhance existing checkers by providing additional test cases, which will continuously trigger the TDCD process.

To evaluate the effectiveness of AutoChecker, we use multiple popular LLMs, including self-hosted and official ones, as follows:

- Self-hosted LLMs: Llama3.1 (`Llama-3.1-8B-Instruct`) [6] and Qwen2.5-Coder (`Qwen2.5-Coder-32B-Instruct-AWQ`) [35].

Table 3: Basic Information of the Benchmark RuleSet

Category	Easy Rules		Hard Rules	
	Rule Name	**#TC**	**Rule Name**	**#TC**
Method Decl.	SignatureDeclareThrowsException	22	MethodNamingConventions	12
Method Call	InefficientEmptyStringCheck	18	LiteralsFirstInComparisons	33
Class Decl.	ExcessivePublicCount	7	ClassWithOnlyPrivateConstructors ShouldBeFinal	22
Variable Decl. and Usage	UseStringBufferForStringAppends	28	AssignmentToNonFinalStatic	6
Exception	ExceptionAsFlowControl	7	AvoidThrowingNullPointerException	9
Expression	NullAssignment	19	BrokenNullCheck	25
Control Stmt	IdenticalCatchBranches	7	EmptyControlStatement	31
Object Inst.	StringInstantiation	10	AvoidInstantiatingObjects InLoops	23
Import	ExcessiveImports	2	UnnecessaryImport	73
Literal	AvoidUsingOctalValues	8	AvoidDuplicateLiterals	11

#TC: the number of test cases. **Abbr.**: Decl.⟶Declaration, Inst.⟶Instantiation

- Official LLMs: GPT-4 (`gpt-4-0613`) [5] and DeepSeek-V3 [45].

4.1.2 Benchmark RuleSet. The benchmark ruleset for evaluation is derived from the official built-in rules in PMD 7.0.0-rc's open-source repository [10]. Initially, there are 132 built-in PMD Java rules. We exclude four rules that are either deprecated or undocumented[2]. The remaining 128 rules are classified based on the primary ASTNode they check, as defined in their official implementations. Fig. 10 shows the distribution of rules across these reclassified categories.

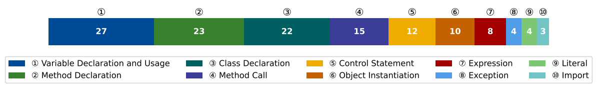


Figure 10: Distribution of Classified PMD's Built-in Rules

For a clearer evaluation, we also divide the collected rules into ***easy rules*** and ***hard rules*** based on the implementation complexity of their official checkers. Statistically, we measure complexity by analyzing specific elements in the checker code: a rule is labeled as *easy* if its checker's *line count*, *import statements*, *method calls*, and *control statements* are all below the average values across all built-in checkers, and if it uses fewer than one semantic class (from `pmd.lang.java.types` or `pmd.lang.java.symbols`). Rules not meeting these criteria are labeled as *hard*.

Overall, we have 128 rules across 10 categories, evenly split into 64 easy and 64 hard rules. For evaluation, we randomly select 10 easy and 10 hard rules, ensuring each represents a unique category. Since PMD provides official test cases for each rule, we extract the default test suites for these 20 rules from PMD's website [9]. By default, these test cases are generally ordered by their difficulty, and we retain this order for AutoChecker. Finally, the benchmark ruleset's details are summarized in Tab. 3.

4.1.3 Baselines and Ablation Methods. According to our knowledge, AutoChecker is the first LLM-based approach for automated code checker generation, specifically for AST-based ones. Thus, we manually develop comprehensive baseline and ablation methods based on LLMs to demonstrate the effectiveness of AutoChecker.

For **RQ1**, we design five baselines to generate the checker at one time inspired by common practices in LLM-powered SE tasks [34]:

[2]Excluded rules are *ExcessiveMethodLength*, *ExcessiveClassLength*, *BeanMembersShould-Serialize*, and *AbstractNamingConvention*.

Table 4: Overall Performance Results of AutoChecker and Baselines Using Different LLMs on the Benchmark RuleSet.

Method + LLM	**#Rule**$_{pc}$ **(/20)**	**#Rule**$_{pot}$ **(/20)**	**#Rule**$_{pat}$ **(/20)**	**#TC**$_{pass}$ **(/373)**	**TPR**$_{avg}$
NoCaseLLM					✎ *naive baseline without test cases*
+ Llama3.1	0	0	0	0	0.00%
+ Qwen2.5-Coder	5	5	1	40	19.41%
+ GPT-4	7	7	1	62	27.92%
+ DeepSeek-V3👍	8	8	1	56	28.06%
AllCasesLLM					✎ *naive baseline with all test cases*
+ Llama3.1	0	0	0	0	0.00%
+ Qwen2.5-Coder	4	4	1	17	14.40%
+ GPT-4	5	5	2	36	21.53%
+ DeepSeek-V3👍	6	6	2	43	24.60%
NoCaseLLM^R					✎ *enhanced baseline with RAG*
+ Llama3.1	2	2	0	16	4.71%
+ Qwen2.5-Coder	9	9	2	60	30.68%
+ GPT-4	10	10	1	108	30.82%
+ DeepSeek-V3👍	9	9	2	92	32.05%
NoCasesLLM^C					✎ *enhanced baseline with COT*
+ Llama3.1	0	0	0	0	0.00%
+ Qwen2.5-Coder	6	6	1	45	21.18%
+ GPT-4	8	8	1	94	27.26%
+ DeepSeek-V3👍	9	9	0	66	29.40%
NoCaseLLM^{RC}					✎ *enhanced baseline with RAG + COT*
+ Llama3.1	2	2	0	7	6.25%
+ Qwen2.5-Coder	9	9	1	60	30.49%
+ GPT-4	9	9	1	105	27.74%
+ DeepSeek-V3👍	11	11	1	101	38.93%
AutoChecker					✎ *our approach*
+ Llama3.1	**3**	**3**	**1**	**22**	**8.41%**
+ Qwen2.5-Coder	**20** ✿	**20** ✿	**4**	**257**	**79.01%**
+ GPT-4👍	**20** ✿	**20** ✿	**6** ✿	**278** ✿	**82.28%** ✿
+ DeepSeek-V3	**19**	**19**	**4**	**278** ✿	**80.86%**

We keep the result with higheset TPR$_{avg}$ across three runs for each method. #TC$_{pass}$ denotes the number of passed test cases in total; ✿ marks the best result of each metric across all methods; 👍 is the best LLM (based on TPR$_{avg}$) for each method.

- **NoCaseLLM**: generates checkers using only the rule description and PMD's checker template, without test cases.
- **AllCasesLLM**: generates checkers with the rule description, PMD checker template, and the full test suite. If the test suite exceeds the LLM's token limit, excess cases are dropped.
- **NoCaseLLM$^{\mathbf{R}}$**: enhances NoCaseLLM with RAG, adding the top-k (default k=19, the mean API count of PMD's built-in checkers) APIs retrieved from the Full-API DB using the rule description as query.
- **NoCaseLLM$^{\mathbf{C}}$**: enhances NoCaseLLM with Chain-of-Thought (CoT) prompting, the LLM is asked to "*first create a comprehensive checking skeleton and then generate the checker*".
- **NoCaseLLM$^{\mathbf{RC}}$**: enhances NoCaseLLM with both RAG and COT strategies.

For **RQ2**, we evaluate the impact of AutoChecker's two key strategies: the logic-guided API-context retrieval and the TDCD cycle (case-by-case iteration). We designed three ablation methods:

- **AutoChecker$^{\mathbf{WoI}}$**: removes the TDCD cycle, providing all test cases, their ASTs, and API-contexts at once. Excess tests are dropped, similar to AllCaseLLM.
- **AutoChecker$^{\mathbf{WoR}}$**: removes the API-context retrieval but retains the TDCD cycle, prompting LLMs without API-contexts.
- **AutoChecker$^{\mathbf{WoM}}$**: removes Meta-Op Set and Meta-API DB. For API-context retrieval, it splits logic into sub-ops based on the rule and test case and retrieves solely on Full-API DB.

In our evaluation, we run each method (including baselines and AutoChecker) **three** times to account for LLM's randomness, and the best performance from each is collected for fair comparison.

4.1.4 Metrics. We design four types of metrics to evaluate a given approach in developing static code checkers.

◆ **Rule**$_{pc}$: A rule is counted as $Rule_{pc}$ if the approach successfully generates a $\underline{p}$ass-$\underline{c}$ompilation checker for it. For the approach, the total number of such rules is recorded as $\#Rule_{pc}$.

◆ **Rule**$_{pot}$: A rule is counted as $Rule_{pot}$ if the approach generates a checker that $\underline{p}$asses at least $\underline{o}$ne of its $\underline{t}$est case. The total number of such rules is recorded as $\#Rule_{pot}$.

◆ **Rule**$_{pat}$: A rule is counted as $Rule_{pit}$ if the approach generates a checker that $\underline{p}$asses $\underline{a}$ll the $\underline{t}$est cases in its test suite. The total number of such rules is recorded as $\#Rule_{pat}$.

◆ **TPR** and **TPR**$_{avg}$: For each rule, we record the test pass rate ($\frac{number\ of\ passed\ test\ cases}{number\ of\ all\ test\ cases}$) of the generated final checker as TPR. TPR_{avg} denotes the average pass rate across all rules.

4.2 RQ1: Effectiveness Evaluation

Tab. 4 shows the main evaluation result of AutoChecker and other baseline methods on the benchmark ruleset based on metrics defined in Section 4.1.4. For each method, we record the result with the highest TPR$_{avg}$ across three runs for fair comparison.

When paired with GPT-4, AutoChecker outperforms all other baselines across different LLMs on all metrics. Specifically, it successfully generates checkers that can pass all tests for six rules, and at least one for all 20 rules (passing 278 test cases in total). Though the generated checkers cannot pass all tests for all the rules, they attain an 82.28% average test pass rate (TPR$_{avg}$), indicating the method's remarkable effectiveness in generating usable checkers.

In general, the performance of all methods (excluding ablation methods in this RQ) across various LLMs follows these rankings:

- ***LLM Rank***: Llama3.1 $<$ Qwen2.5-Coder $\lesssim$ GPT-4 $\lesssim$ DeepSeek-V3
- ***Method Rank***: AllCasesLLM $<$ NoCasesLLM $<$ NoCaseLLM$^{\mathrm{C}}$ $<$ NoCaseLLM$^{\mathrm{R}}$ $<$ NoCaseLLM$^{\mathrm{RC}}$ $<$ AutoChecker.

The LLM-rank result generally aligns with other LLM-evaluation studies [35, 45, 47]. The smallest model, Llama3.1, with limited code-related capability, often leads to compilation failures caused by syntax errors. In contrast, the other three LLMs, being more powerful, can generate test-passing checkers. Among them, DeepSeek-V3 excels in all baselines, while GPT-4 gets the best result for AutoChecker (checkers generated with DeepSeek-V3 and GPT-4 pass the same number of tests but vary in test distribution over rules, leading to the difference in TPR$_{avg}$). Notably, AutoChecker with the self-hosted LLM (Qwen-Coder-2.5) also achieves a considerable TPR$_{avg}$ of 79.01%, making it promising for privacy-sensitive and resource-constrained code-checking applications.

Based on the method rank, AutoChecker significantly outperforms all baselines. Specifically, it achieves 2.93× the performance of NoCaseLLM, 3.34× of AllCasesLLM, 2.57× of NoCaseLLM^R, 2.80× of NoCaseLLM^C and 2.11× of NoCaseLLM^{RC} on TPR$_{avg}$. Though the performance of NoCaseLLM can be augmented with prompt engineering techniques (COT and RAG), the metric TPR$_{avg}$ is still below 40%, and most generated checkers cannot even pass compilation. Compilation errors primarily stem from

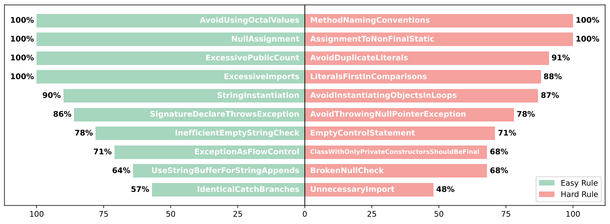


Figure 11: TPR Distribution for Checkers Generated by AutoChecker+GPT-4 on Easy Rules and Hard Rules

insufficient API knowledge, leading to API hallucinations such as incorrect class names and method calls. These results also prove that simply retrieving API-contexts based on the rule description (in AutoChecker^{R} and AutoChecker^{RC}) is coarse-grained, often resulting in retrieval failures and, eventually LLM hallucinations.

To further analyze AutoChecker's performance on easy and hard rules, we collect the TPR distribution for all rules using GPT-4, the best-performing LLM. As shown in Fig. 11, the results align with expectations: hard rules are more challenging, with average TPRs of 84.60% for easy rules and 79.90% for hard rules. Specifically, the generated checkers pass all tests for 4 easy rules and 2 hard rules.

Failure Discussion. From the results, checkers generated by AutoChecker with GPT-4 fail on 95 test cases, which are skipped after reaching the retry limit. We randomly sample about half (45) from different rules and categorize the failures into compilation errors (due to hallucinated APIs), selected test failures (failing the current test), and regression test failures (failing previously passed tests). Besides API retrieval precision, LLM capability is also a key reason for these failures, as we observed LLMs using deprecated or wrong APIs even when correct ones are provided (e.g., using deprecated API `jjtGetNumChildren` for rule *ExecptionAsFlowControlRule* even the correct API `getNumChildren` has been provided in prompts).

➠ **Answering RQ1:** AutoChecker outperforms both naive and enhanced baselines, achieving the highest 82.28% TPR$_{avg}$ with GPT-4. It indicates that our approach can effectively help developers to write their own checkers only with the rule and test suite.

4.3 RQ2: Ablation Study

To evaluate the effectiveness of specific strategies in AutoChecker, we conduct ablation experiments. As GPT-4 and DeepSeek-V3 achieve comparable performance (discussed in RQ1), we use both for the ablation study. Tab. 5 gives the overall results.

We start by analyzing the effectiveness of retrieval and iteration settings. In terms of TPR$_{avg}$, AutoChecker^{WoI} achieves better performance using DeepSeek-V3, while AutoChecker^{WoR} performs better using GPT-4. Compared to them, AutoChecker with GPT-4 improves TPR$_{avg}$ by 53.97% and 22.31%, respectively. This shows that both API-context retrieval and the TDCD cycle are essential, with API-context retrieval being particularly crucial. As shown in the second column, AutoChecker^{WoI} has fewer pass-compilation checkers than AutoChecker^{WoR}. Without accurate API knowledge, AutoChecker and any other LLM-based methods use hallucinated APIs and will fail due to compilation errors.

To validate the effectiveness of the meta-settings (Meta-Op Set and Meta-API DB) in AutoChecker, we introduce the ablation method AutoChecker^{WoM}. As shown in Tab. 5, while it gets good performance on TPR$_{avg}$ of around 70% only based on the Full-API DB, it is still at least 10 percent point lower than AutoChecker. This result highlights the critical role of meta-settings in retrieval.

Table 5: Results of AutoChecker and Ablation Methods using GPT-4 and DeepSeek-V3 on the Benchmark Ruleset.

Method + LLM	#Rule$_{pc}$ (/20)	#Rule$_{pot}$ (/20)	#Rule$_{pat}$ (/20)	#TC$_{pass}$ (/373)	TPR$_{avg}$
AutoChecker^{WoI}			✎ *ablation method without iterations*		
+ GPT-4	8	8	2	65	29.37%
+ DeepSeek-V3👍	14	14	4	141	53.44%
AutoChecker^{WoR}		✎ *ablation method without API-context retrieval*			
+ GPT-4👍	18	18	2	231	67.27%
+ DeepSeek-V3	15	15	2	221	59.17%
AutoChecker^{WoM}	✎ *ablation method without Meta-Op Set and Meta-API DB*				
+ GPT-4	17	17	3	256	66.42%
+ DeepSeek-V3👍	18	18	1	258	72.92%
AutoChecker					✎ *our approach*
+ GPT-4👍	**20** ✿	**20** ✿	**6** ✿	**278** ✿	**82.28%** ✿
+ DeepSeek-V3	**19**	**19**	**4**	**278** ✿	**80.86%**

➠ **Answering RQ2:** Both the *Retrieval* and *Iteration* strategies are necessary for AutoChecker. Also, with the meta-settings, its average test pass rate increases by around 10 percentage points.

4.4 RQ3: Cost of AutoChecker

We evaluate AutoChecker's time and financial costs respectively. Our observations show consistent time and token costs across different LLMs, as they are all accessed via official or self-hosted APIs. Since AutoChecker struggles to achieve good results with Llama3.1, we analyze the average costs across the other three LLMs.

For time cost, we measure the average duration across three runs for easy rules, hard rules, and all rules combined. As shown in Fig. 12, AutoChecker takes 70 minutes to generate the final checker per rule on average: 40 minutes for easy rules and 100 minutes for hard rules. As a comparison, traditional checker development in practice needs both manual implementation and may suffer multi-day delays from cross-role coordination between managers and developers. Thus, AutoChecker is more efficient and able to generate checkers automatically once tests are prepared.

For financial cost, we calculate the token usage using the corresponding tokenizers, accounting for 121k input and 388 output tokens on average. Generating a checker costs approximately \$3.65 for GPT-4 and \$0.035 for DeepSeek-V3 per rule. As Tab. 4 shows, AutoChecker achieves comparable performance across LLMs, enabling users to opt for cheaper options (DeepSeek-V3) or API-free ones (Qwen2.5-Coder). For enterprises that need custom checkers, the financial cost of AutoChecker is far more affordable than hiring expertise for manual development.

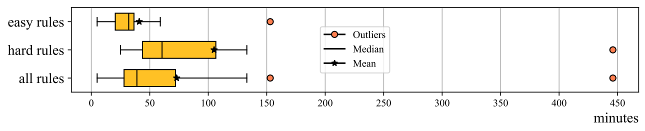


Figure 12: Time Cost of AutoChecker on Different Rule Set

➠ **Answering RQ3:** The time and financial cost of AutoChecker is more affordable compared to traditional checker development.

4.5 RQ4: Practicality in Real-world Projects

To evaluate the applicability of AutoChecker-generated checkers, we apply them to scan real-world projects and compare their performance with official

Table 6: Violations Reported by the Official and AutoChecker-Generated Checker on Real-world Projects

Checker Rule	$\#TC_+$	#Violations on Five Projects		
		official checker	$\text{Checker}_{AutoChecker+GPT\text{-}4}$ with TS_{orig}	$\text{Checker}_{AutoChecker+GPT\text{-}4}$ with TS_{aug}
NullAssignment	**+5**	2,560	1,632 (↓928)✘	**2,562** (↑2)
ExcessivePublicCount	**+6**	389	330 (↓59)	**389** (=0)
ExcessiveImports	**+0**	3,321	3,321 (=0)	**3,321** (=0)
AvoidUsingOctalValues	**+7**	58	0 (↓58)	**58** (=0)
MethodNamingConventions	**+1**	11,562	11,560 (↓2)	**11,562** (=0)
AssignmentToNonFinalStatic	**+0**	8	8 (=0)✘	**8** (=0)
StringInstantiation	**+0**	347	347 (=0)	**347** (=0)
InefficientEmptyStringCheck	**+2**	16	28 (↑12)	**16** (=0)

TS_{orig}(original test suite) + **TC_+**(new test cases) ⟶ **TS_{aug}**(augmented test suite);
Checker with TS_{orig}/TS_{aug}: generated checker based on a rule and its TS_{orig}/TS_{aug};
✘ denotes that the checker meets crash during project scan.

checkers. For project selection, we choose five popular Java projects[3] from GitHub, each with over 50K stars and ranging from 50 to 1,517 KLOC. For checker selection, we use the generated ones that pass all tests and identify them as successful. Instead of selecting from a single run, we collect the successful checkers from all three runs. Specifically, we select the eight[4] successful checkers generated by AutoChecker with GPT-4, as this exceeds the number with DeepSeek-V3 (six) and other LLMs.

Table 6 shows the number of reported violations by both official and AutoChecker-generated checkers for each project. As shown in the fourth column, only three of the generated checkers based on the original test suite achieve the same performance compared to official ones. Among all the eight checkers, we observe missing reports (FNs) for four checkers and mistaken reports for one checker (FP), while two checkers encounter crashes during code scanning.

Through careful manual analysis, we identified two main reasons for the performance gap: implementation bugs (crash) and omitted checking logic for corner cases (FPs and FNs). Implementation bugs are mostly simple, missing null checks and failing to perform type checking before casting. They are quickly fixed by directly asking LLMs to repair with bug reports. For FPs and FNs, they can be reduced by augmenting the original test suite. To address this, we craft test cases to cover missing checking scenarios. The number of added cases is shown in the fifth column in Table 6.

After bug fixes and test augmentation, the newly generated checkers successfully report all violations, matching the performance of official ones. Additionally, the *NullAssignment* checker reports two more violations, which are repeated ones at the same location (other reports are not repeated). As they are redundant true violations, we do not take them as FPs.

➠ **Answering RQ4:** Given an adequate test suite, AutoChecker can generate checkers with comparable real-world performance to official ones. It shifts the development effort from the challenging task of writing checkers to the more manageable task of designing test suites.

5 Threats to Validity

A primary threat to validity is the generalizability of AutoChecker. As our implementation targets PMD for Java, it may not easily apply to other code-checking tools and programming languages. To address this, we design AutoChecker with framework- and language-agnostic strategies. Specifically, the core test-driven development cycle is LLM-based and language-independent, while the Meta-Op Set for API-context retrieval is conceptually universal. Ideally, AutoChecker can be adapted to any tool that supports custom AST-based checkers and all languages. During extension, the main practical hurdle is the one-time, manual effort to construct the API-context databases. This cost is made manageable through a semi-automated process. For initial framework API collection, manual work is needed to adapt the API-collecting scripts to new framework. Then, the Full-API DB can be automatically built, while the Meta-API DB needs further manual validation and supplementation as introduced in Section 3.2.

Another threat is that the selected rules in the benchmark ruleset may not be representative. To mitigate this, we choose rules from PMD's built-in set, which are widely recognized as references. After classifying these rules by difficulty and targets, we randomly select rules via the stratified sampling strategy to ensure balanced representation across both difficulty levels and categories, as introduced in Section 4.1.2.

6 Future Directions

The overall checker development workflow consists of both the test preparation and checker development. Our work focuses on automating checker development, as it represents the primary bottleneck in complete workflow, demanding specialized expertise. In contrast, test case creation is a more manageable task. However, this preparatory step could also be automated. One direction for future work could be exploring using LLMs or historical patterns to automatically generate test suites from rule descriptions, creating a fully automated, end-to-end checker production pipeline.

Another natural next step is to extend AutoChecker's support to other popular static analysis frameworks like CodeQL or Sonarcube, as well as other programming languages such as Python and C++. As established in our validity analysis, this extension requires a manageable, one-time investment to construct the necessary API-context database for each new framework. By broadening its compatibility, AutoChecker could bring the benefits of automated, test-driven checker development to a much wider audience of developers.

7 Related Work

7.1 Code Checker Development

In static analysis studies, researchers develop code checkers for various discovered bug patterns [17, 21, 70]. For instance, Chen et al. [21] summarized anti-patterns in logging code, and Zhang et al. [70] designed bug patterns for exception handling. These patterns are then manually encoded as a static checker for real-world issue detection. While effective, manual checker implementation is time-consuming and requires significant expertise.

Code checker development requires highly specialized, flexible implementations, which traditional code generation approaches cannot handle. Traditional pattern/template-based code generation approaches synthesize code by expanding templates with static schema (design-time) and dynamic data inputs (run-time) [59], commonly used for structured tasks with reusable patterns (e.g., class synthesis [19, 38], algorithmic transformation [36, 64]). Though checkers also need to follow basic structural design (e.g., the PMD template we summarized in Sec.), they demand variable and flexible checking logic (implementations) for different rules—unpredictable at design-time and impossible to predefine in templates.

Recently, the advent of Machine Learning (ML) and LLMs has inspired researchers to analyze and scan code with models instead of implementing code checkers [15, 33, 53, 72]. Most studies directly apply ML models to detect various vulnerabilities, such as GNN-based Devign [73], Transformer-based LineVul [32], LLM-based Llm4Vuln [58], etc. However, these approaches mostly focus on function-level detection and only identify limited types of vulnerabilities, which are not effective at detecting vulnerabilities in real-world projects [27, 57].

In order to scan real-world projects, researchers recently explore combining static analysis tools (code checkers) with models like LLMs. Specifically, Wang et al. [62] and Li et al. [44] leverage LLMs to infer source-sink specifications to augment taint checkers for a given project and CWE, while some

[3]Algorithms/Java [13], elastic/elasticsearch [3], macrozheng/mall [8], google/guava [4], and spring-projects/spring-boot [12].
[4]The number of successful checkers across three runs: 6, 6, 5; Deduplicated total: 8.

studies [22, 41, 42] directly use LLMs to reduce the false positive alarms of static analysis tools. However, these efforts focus on improving existing checkers rather than creating new ones. In contrast, AutoChecker generates custom checkers through an automated end-to-end way based on LLMs.

7.2 LLM-based Repo-level Code Generation

Recently, code generation tasks have been revolutionized by LLMs [31, 34, 39]. LLMs have shown incredible capability in generating individual programs [23, 47]. Repo-level code generation aims at generating code using the APIs defined in the repository [69]. Compared to function-level generation, this task is more challenging and downstream, requiring repo-specific API knowledge. A recent survey [26] categorized methods for repo-level generation into two types: fusion-based and ranking-based.

Fusion-based approaches [14, 28, 54] jointly model repo-context into the LLM. Among these studies, MGD [14] queried static analysis tools in the background, and the answers participated in the model's decoding stage to influence code generation. These approaches usually need to modify the model decoding process, while AutoChecker augments related contexts directly into the prompt.

Ranking-based methods [49, 55, 68, 69, 71] retrieve the most similar code context from the repository into the prompt, which are primarily used in most studies. For example, Liu et al. [49] find relevant import statements and similar code snippets into the prompt for repo-level code generation, while Zhang et al. [69] apply two-stage retrieval for fine-grained API retrieval. In AutoChecker, the logic-guided API-context retrieval method is also ranking-based, with optimizing settings (the decomposed logic-guided retrieval and Meta-Op DB) specifically designed for checker generation.

8 Conclusions

We propose AutoChecker, an LLM-powered approach to automatically write static code checkers with the rule description and the corresponding test suite. To the best of our knowledge, this is the first attempt to explore test-guided static checker generation using LLMs. AutoChecker employs a novel test-driven checker development process to incrementally generate and refine the checker case by case. During each round, it retrieves related API-contexts as additional knowledge for the LLM through the logic-guided API-context retrieval method. Experimental results show that AutoChecker's effectiveness outperforms baseline approaches across all the metrics, including the average test pass rate. Furthermore, with adequate test cases, AutoChecker is able to generate checkers that perform nearly as well as official ground truth checkers in real-world projects.

Acknowledgments

Thanks to all the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 62132020, and Major Project of ISCAS (ISCAS-ZD-202302).

References

[1] 2024. BAAI. https://www.baai.ac.cn/.

[2] 2024. CodeQL. https://codeql.github.com.

[3] 2024. elastic/elasticsearch. https://github.com/elastic/elasticsearch/commit/9eab11c.

[4] 2024. google/guava. https://github.com/google/guava/commit/b84a41d.

[5] 2024. GPT 4|OpenAI. https://openai.com/index/gpt-4/.

[6] 2024. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/.

[7] 2024. LangChain. https://www.langchain.com/.

[8] 2024. macrozheng/mall. https://github.com/macrozheng/mall/commit/370eb4b.

[9] 2024. PMD Source Code Analyzer. https://docs.pmd-code.org/pmd-doc-7.0.0-rc4/.

[10] 2024. pmd/pmd. https://github.com/pmd/pmd/tree/pmd_releases/7.0.0-rc4.

[11] 2024. Sonarqube. https://www.sonarqube.org.

[12] 2024. spring-projects/spring-boot. https://github.com/spring-projects/spring-boot/commit/fadd054.

[13] 2024. TheAlgorithms/Java. https://github.com/TheAlgorithms/Java/commit/bcf4034.

[14] Lakshya A. Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023*.

[15] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[16] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[17] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. Nar-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 411–422.

[18] David Binkley. 2007. Source code analysis: A road map. *Future of Software Engineering (FOSE'07)* (2007), 104–119.

[19] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. 2018. F Prime: an open-source framework for small-scale flight software systems. (2018).

[20] Fraser Brown, Andres Nötzli, and Dawson R. Engler. 2016. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016*. ACM, 143–157.

[21] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 71–81.

[22] Jinbao Chen, Hongjing Xiang, Luhao Li, Yu Zhang, Boyao Ding, and Qingwei Li. 2024. Utilizing Precise and Complete Code Context to Guide LLM in Automatic False Positive Mitigation. *arXiv preprint arXiv:2411.03079* (2024).

[23] 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).

[24] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE security & privacy* 2, 6 (2004), 76–79.

[25] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343.

[26] Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, et al. 2024. R2C2-Coder: Enhancing and Benchmarking Real-world Repository-level Code Completion Abilities of Code Large Language Models. *arXiv preprint arXiv:2406.01359* (2024).

[27] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).

[28] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code Completion by Jointly Modeling In-file and Cross-file Context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024*. ELRA and ICCL, 3433–3445.

[29] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[30] Yao-Yang LIU Zhen ZHENG Feng ZHANG Jin-Cheng FENG Yi-Yang FU Ji-Dong ZHAI Bing-Sheng HE Xiao ZHANG Xiao-Yong DU. 2026. A comprehensive taxonomy of prompt engineering techniques for large language models. *Frontiers of Computer Science* 20 (2026), 2003601–. doi:10.1007/s11704-025-50058-z

[31] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[32] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[33] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[34] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[35] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[36] Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2024. Decomposition-based synthesis for applying divide-and-conquer-like algorithmic paradigms. *ACM Transactions on Programming Languages and Systems* 46, 2 (2024), 1–59.

[37] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[38] Sven Jörges. 2013. *Construction and evolution of code generators: A model-driven and service-oriented approach*. Vol. 7747. Springer.

[39] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering* FSE (2024).

[40] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[41] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2107–2111.

[42] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[43] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* (2023).

[44] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=9LdJDU7E91

[45] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[46] Han Liu, Sen Chen, Ruitao Feng, Chengwei Liu, Kaixuan Li, Zhengzi Xu, Liming Nie, Yang Liu, and Yixiang Chen. 2023. A comprehensive study on quality assurance tools for Java. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 285–297.

[47] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[48] Jun Liu, Jiwei Yan, Yuanyuan Xie, Jun Yan, and Jian Zhang. 2024. Fix the Tests: Augmenting LLMs to Repair Test Cases with Static Collector and Neural Reranker. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE.

[49] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. Codegen4libs: A two-stage approach for library-oriented code generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 434–445.

[50] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022*. Association for Computational Linguistics, 6227–6240.

[51] Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional API Recommendation for Library-Oriented Code Generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 87–98.

[52] Diogo S Mendonça and Marcos Kalinowski. 2022. An empirical investigation on the challenges of creating custom static analysis rules for defect localization.

Software Quality Journal 30, 3 (2022), 781–808.

[53] Yu CHEN Yi SHEN Taiyan WANG Shiwen OU Ruipeng WANG Yuwei LI Zulie PAN. 2026. Software defect detection using large language models: a literature review. *Frontiers of Computer Science* 20 (2026), 2006202–. doi:10.1007/s11704-025-40672-2

[54] Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998* (2023).

[55] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[56] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. 2002. Code quality analysis in open source software development. *Information systems journal* 12, 1 (2002), 43–60.

[57] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. 2024. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv e-prints* (2024), arXiv–2403.

[58] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[59] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. 2018. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures* 52 (2018), 43–62.

[60] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT feedback: What experienced developers like about static analysis. In *Proceedings of the 26th Conference on Program Comprehension*. 64–73.

[61] Sebastian Uchitel, Marsha Chechik, Massimiliano Di Penta, Bram Adams, Nazareno Aguirre, Gabriele Bavota, Domenico Bianculli, Kelly Blincoe, Ana Cavalcanti, Yvonne Dittrich, et al. 2024. Scoping software engineering for AI: the TSE perspective. Institute of Electrical and Electronics Engineers.

[62] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2023. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. *arXiv preprint arXiv:2311.04448* (2023).

[63] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 284–289.

[64] Ziteng Wang, Shankara Pailoor, Aaryan Prakash, Yuepeng Wang, and Işıl Dillig. 2024. From Batch to Stream: Automatic Generation of Online Algorithms. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1014–1039.

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[66] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. arXiv:2309.07597 [cs.CL]

[67] Xiaoheng Xie, Gang Fan, Xiaojun Lin, Ang Zhou, Shijie Li, Xunjin Zheng, Yinan Liang, Yu Zhang, Na Yu, Haokun Li, Xinyu Chen, Yingzhuang Chen, Yi Zhen, Dejun Dong, Xianjin Fu, Jinzhou Su, Fuxiong Pan, Pengshuai Luo, Youzheng Feng, Ruoxiang Hu, Jing Fan, Jinguo Zhou, Xiao Xiao, and Peng Di. 2024. CodeFuse-Query: A Data-Centric Static Code Analysis System for Large-Scale Organizations. *CoRR* abs/2401.01571 (2024).

[68] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. In *EMNLP (Findings)*. Association for Computational Linguistics, 277–288.

[69] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*. Association for Computational Linguistics, 2471–2484.

[70] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. 2023. Detecting exception handling bugs in C++ programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1084–1095.

[71] Peitian Zhang, Shitao Xiao, Zheng Liu, Zhicheng Dou, and Jian-Yun Nie. 2023. Retrieve anything to augment large language models. *arXiv preprint arXiv:2310.07554* (2023).

[72] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 47–51.

[73] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).