# Still Manual? Automated Linter Configuration via DSL-Based LLM Compilation of Coding Standards

ZEJUN ZHANG, Nanjing University, China
YIXIN GAN, Nanjing University, China
ZHENCHANG XING*, CSIRO's Data61, Australia
TIAN ZHANG, Nanjing University, China
YI LI, Nanyang Technological University, Singapore
QINGHUA LU, CSIRO's Data61, Australia
SHERRY (XIWEI) XU, CSIRO's Data61, Australia
LIMING ZHU, CSIRO's Data61, Australia

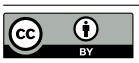
Coding standards are essential for maintaining consistent and high-quality code across teams and projects. Linters help developers enforce these standards by detecting code violations. However, manual linter configuration is complex and expertise-intensive, and the diversity and evolution of programming languages, coding standards, and linters lead to repetitive and maintenance-intensive configuration work. To reduce manual effort, we propose LintCFG, a domain-specific language (DSL)-driven, LLM-based compilation approach to automate linter configuration generation for coding standards, independent of programming languages, coding standards, and linters. Inspired by compiler design, we first design a DSL to express coding rules in a tool-agnostic, structured, readable, and precise manner. Then, we build linter configurations into DSL configuration instructions. For a given natural language coding standard, the compilation process parses it into DSL coding standards, matches them with the DSL configuration instructions to set configuration names, option names and values, verifies consistency between the standards and configurations, and finally generates linter-specific configurations. Experiments with Checkstyle for Java coding standard show that our approach achieves over 90% precision and recall in DSL representation, with accuracy, precision, recall, and F1-scores close to 70% (with some exceeding 70%) in fine-grained linter configuration generation. Notably, our approach outperforms baselines by over 100% in precision. An ablation study confirms the effectiveness of the main components of our approach. A user study further shows that our approach improves developers' efficiency in configuring linters for coding standards. Finally, we demonstrate the generality of the approach by generating ESLint configurations for JavaScript coding standards, showcasing its broad applicability across other programming languages, coding standards, and linters. We developed a lightweight, general-purpose AI skill, which is publicly available on GitHub.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Coding Standard, Linter Configuration, Domain-Specific Language

*Corresponding author.

Authors' Contact Information: Zejun Zhang, Nanjing University, Nanjing, China, zejun.zhang@nju.edu.cn; Yixin Gan, Nanjing University, Nanjing, China, ganyixin@smail.nju.edu.cn; Zhenchang Xing, CSIRO's Data61, Canberra, Australia, zhenchang.xing@data61.csiro.au; Tian Zhang, Nanjing University, Nanjing, China, ztluck@nju.edu.cn; Yi Li, Nanyang Technological University, Singapore, Singapore, yi_li@ntu.edu.sg; Qinghua Lu, CSIRO's Data61, Sydney, Australia, qinghua.lu@data61.csiro.au; Sherry (Xiwei) Xu, CSIRO's Data61, Sydney, Australia, xiwei.xu@data61.csiro.au; Liming Zhu, CSIRO's Data61, Sydney, Australia, liming.zhu@data61.csiro.au.




ACM 2994-970X/2026/7-ARTFSE002
https://doi.org/10.1145/3797064

**ACM Reference Format:**
Zejun Zhang, Yixin Gan, Zhenchang Xing, Tian Zhang, Yi Li, Qinghua Lu, Sherry (Xiwei) Xu, and Liming Zhu. 2026. Still Manual? Automated Linter Configuration via DSL-Based LLM Compilation of Coding Standards. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE002 (July 2026), 23 pages. https://doi.org/10.1145/3797064

## 1 Introduction

Coding standards, also known as coding conventions or coding styles, are guidelines that govern how code should be written and organized within programming languages [21, 24, 39, 54, 61, 63]. They cover a wide range of aspects, including file basics, file structure, formatting, documentation, naming conventions, language features, and basic programming practices. To maintain software quality and enable long-term maintenance, many organizations, software projects, and open-source communities establish their own coding standards [1–3, 12, 76]. To enforce coding standards, numerous linting tools have been developed to automatically detect standard violations in code [7, 9, 11, 16, 29, 31, 45, 46, 56, 68, 77]. Previous research has primarily focused on individual studies of coding standards and linters, such as the differences and evolution of coding standards for a programming language [20, 21, 27, 45, 50, 63], and improving the capabilities of linting tools [7, 9, 11, 16, 22, 28, 65, 66, 79].

Automatically generating linter configurations for coding standards has long been overlooked, despite its significance. First, different programming communities, organizations, and software projects choose their own coding standards and linters. This creates substantial mental overhead for developers, as there is no unified tooling to streamline configuration generation. Secondly, the configuration process is complex: developers must first master the coding standard and linters, then sift through hundreds of configurations to find the corresponding linter configuration names, determine the matched options and their values, and then generate machine-readable formats supported by the linter. Moreover, developers must continuously track changes in coding standards and linter tools, which is both cumbersome and error-prone. As new coding standards emerge or projects transition to new linters, manually updating configurations becomes even more challenging.

Our observation on over 1,000 GitHub repositories highlights the widespread and active evolution of both coding standards and linters (see APPENDIX A). From 2009 to 2024, 1,066 coding standard projects and 1,692 linter projects were created. Of these, 20% (218 out of 1,066) of coding standards and 47% (787 out of 1,692) of linter tools were under active development in 2024. Moreover, our analysis of Stack Overflow questions (Section 2) reveals that users often do not know whether linters support a coding standard and struggle to configure linters correctly. This highlights the need for automated linter configuration support to reduce manual efforts.

To automatically generate linter configurations to comply with a coding standard, we propose LintCFG, a domain-specific language (DSL)-driven, LLM-based compilation approach that is independent of programming languages, coding standards, or linters. Our approach is inspired by compiler design [41, 48, 53], which focuses on translating high-level representations into machine-readable code.

We first design a DSL to represent coding rules in a tool-agnostic, structured, precise, and readable manner. This decision is motivated by the observed limitations of existing representations: natural-language coding standards are tool-agnostic and readable but lack structure and precision, whereas machine-readable linter configuration formats are precise and structured but tied to specific linters and often difficult to understand. We use a card sorting approach [64] to design the DSL by analyzing coding rules from the Google Java coding standard and Checkstyle documentation [12, 15]. We select these sources not to serve specific standards or linters, but because of their broad adoption and comprehensive coverage, making them ideal foundations for DSL design. Leveraging this generic DSL and the generation capabilities of LLMs, coding rules from coding standards and linter

configurations can be parsed into a unified intermediate form, agnostic to programming languages, coding standards, or linters.

We then build linter configurations as a DSL configuration instruction set. A typical linter configuration consists of a configuration name, option names, and their corresponding values. Inspired by the structure of configuration schemas, we define each configuration as a DSL instruction comprising the general instruction and the option instruction. The general instruction is to represent the overall behavior for configuration names in DSL. The option instruction is to represent the specific behavior for different option values of a option name in DSL. After building the linter instruction framework, we prompt LLMs to generate the DSL configuration instruction for each linter configuration based on the linter documentation.

Finally, given a natural language (NL) coding standard, we design a compilation process to generate the linter-specific configuration building on the DSL and DSL configuration instruction set. Similarly to compiler steps such as syntax parsing, intermediate code generation, semantic analysis, and machine code generation, we break the compilation into five steps, with each step focusing on a specific task to ensure correctness and scalability. First, we parse the NL coding standard into the DSL coding standard. Then, we select configuration names by matching the DSL coding standard with general DSL instructions. Next, we determine option names and values by matching the option DSL instructions with the DSL coding standard. To ensure alignment, we verify consistency between the DSL coding standard and the matched DSL instructions in terms of rule types, checked objects, and semantics. Finally, we convert the aligned configurations into the linter-specific configuration (e.g., in XML format).

We comprehensively evaluate our approach across five dimensions. We first assess the effectiveness of DSL representations for the Google Java coding standard and Checkstyle configurations. The generated DSL representations achieve over 80% accuracy and exceed 90% in precision, recall, and F1-score. Next, we evaluate the effectiveness of our approach in generating Checkstyle configurations for Google Java coding standard at three levels of granularity: configuration name, option name, and option value. We design six baselines with varying levels of linter information and Retrieval-Augmented Generation (RAG) capabilities. Our approach significantly outperforms all baselines, approaching or exceeding 70% in all metrics across all levels, with minimum improvements of 51.5%, 106.3%, 5.1%, and 77.7% in accuracy, precision, recall, and F1-score, respectively. Notably, precision improves by more than 100% over the best-performing baseline. We further conduct an ablation study to validate the contributions of the DSL design and the compilation process. Furthermore, a user study shows that our generated configurations for coding standards can help linter users complete the task more quickly and correctly. Finally, we evaluate the generality of our approach by generating ESLint configurations for JavaScript coding standards. Our approach consistently achieves high accuracy, precision, recall, and F1-score, demonstrating its effectiveness across different programming languages, coding standards, and linter tools.

The contributions of this paper are as follows:

- To the best of our knowledge, we are the first to automatically generate linter configurations for coding standards.
- We propose a DSL-driven, LLM-based compilation approach that is independent of specific programming languages, coding standards, and linter tools. Experimental results demonstrate the effectiveness, usefulness, and generality of the approach.
- We highlight key implications for future research and construct benchmarks for Checkstyle configurations for Google Java coding standard, as well as ESLint configurations for the Google JavaScript coding standard, to support future studies.

## 2 Motivation Study

**Motivation.** We are interested in exploring whether users struggle with configuring linter tools to motivate the task of automating linter configuration generation for coding standards.
**Method.** To understand the challenges of configuring linters for coding standards, we search Stack Overflow questions using linter tool names (e.g., Checkstyle, ESLint, Pylint, and Prettier) and coding standards (e.g., coding standard, coding convention, style guide and rule) as keywords. From the top 500 questions returned, two authors independently review and classify each question as related or unrelated to challenges in configuring linters for coding standards. Cohen's kappa [69] is 0.89, indicating substantial agreement. Disagreements are resolved through discussion, resulting in 90 questions being classified as related to challenges in configuring linters for further analysis. Finally, the authors categorize these questions into different types of challenge through discussion.
**Result.** We summarize two types of challenges in configuring linters for coding standards.
**(1) Users feel uncertain whether a linter supports a coding standard** because they must fully understand each configuration's semantics and continuously track tool updates. It accounts for 44.4% (40 of 90 questions). For example, users want Checkstyle to prohibit all class methods except specific ones [8]. They found a possible configuration, but it cannot exclude certain methods, making them uncertain about Checkstyle's support for the standard. Although this question [8] was asked 14 years ago, it was still active within 1 year and was viewed more than 291,000 times, indicating that users find it challenging and time-consuming to determine whether linting tools support the intended coding standards. For another example, users asked whether Pylint supported comment spell checking 11 years ago [6]. This feature was initially unavailable but was later introduced in version 1.4, showing that users must track tool updates to determine if a linter supports a coding standard.
**(2) Users struggle to configure the linter correctly** due to the challenges of setting the correct tool configuration names, option names, and option values for the coding standards, accounting for 55.6% (50 of 90 questions). For example, users asked how to remove semicolons in Prettier. Although they knew this feature was supported in Prettier, they didn't know how to enable it in the settings [13]. The corresponding configuration name for Prettier is "semi". For another example, users mistakenly thought the "exception" option of spaced-comment rule in ESLint addressed exceptions for spaced comments on special comments [10]. In reality, it is the "markers" option that handles this coding standard.

**Summary:** Our analysis of Stack Overflow questions shows that users face challenges when configuring linter tools for coding standards.

## 3 Approach

Figure 1 illustrates our approach for generating linter configurations for coding standards, which consists of three steps. The first two steps are preparation phases that are generally performed once, while the third step is executed each time a coding standard needs to be configured in the linter. First, we design a generic domain-specific language (DSL) to clearly express the coding rules of both coding standards and linter configurations. Next, we build linter configurations as a DSL configuration instruction set by parsing linter documentation. Finally, for a given natural language (NL) coding standard, we employ a compilation pipeline to produce the corresponding linter configurations. This pipeline parses the NL coding standards into DSL coding standards, matches them with the DSL instruction set to determine configuration names, option names, and values, and verifies their alignment. Once aligned, the linter configurations are generated in tool-specific formats such as XML. Figure 2 shows an example of NL coding standards and the corresponding Checkstyle configurations.

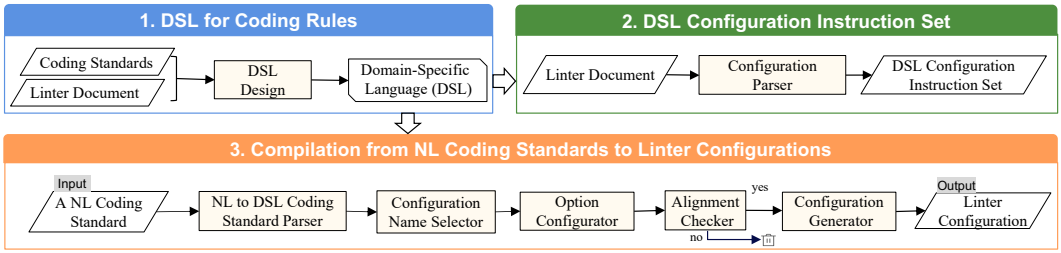



Fig. 1. Approach overview of generating linter configurations for coding standards

**Natural Language (NL)**

**Google Java Style Guide:**

- **Sentence i:** Sometimes there is more than one reasonable way to convert an English phrase into camel case.
- **Sentence j:** The standard block tags @param, @return, @throws, @deprecated should always appear in this order, and these tags must not have an empty description.

**Explanation:**

- **Sentence i:** it does not contain rules.
- **Sentence j:** it contains rules but **mixes different levels of information** such as the number of rules, rule types, objects being checked by rules, and corresponding constraint types of rules.
  - the sentence contains **two rules**;
  - "should always" and "must" indicate that the two rules are **mandatory rules**;
  - "block tag" and "these tags" refer to the **four types of block tags being checked**;
  - "appear in this order" and "not have" imply **constraint relationships**, such as order and negation.

**NL Summary:**

**Good:** general; readable **Bad:** too free → lacking structure and precision

**Linter Configuration Format (e.g., XML)**

**Checkstyle Configuration:**

```
<module name='AtclauseOrder'>
  <property name="tagOrder" value="@param, @return, @throws,
  @deprecated"/> </module>
<module name='NonEmptyAtclauseDescription'>
  <property name="tokens" value="PARAM_LITERAL,
  RETURN_LITERAL, THROWS_LITERAL,
  DEPRECATED_LITERAL"/> </module>
```

**Explanation:**

- The module names, property names and values are **too vague and abstract**, making it difficult to understand their specific meanings.
- The **format** and **configuration settings**, such as module names, are **specific to** Checkstyle, while other linters use distinct formats and settings.

**Linter Configruation Format Summary:**

**Good:** structured; precise **Bad:** tool-dependent; hard to read

**Domain-Specific Language (DSL)**

**DSL Representation:**

Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated] ;
Mandatory: No [EmptyDescription] for [@param,@return,@throws,@deprecated]

**Explanation:**

- **Each rule** is represented **independently**.
- The prefix "Mandatory:" denotes the **rule type**, indicating it must be satisfied.
- "Order " and "No" denote different **constraint relationships**: order and negation.
- **Terms denoting objects to be checked** are enclosed in square brackets ([ ]), which distinguishes them from terms that denote constraint relationships.

**DSL Summary:**

**Good:** general; readable; structured; precise

***The DSL We Designed:***

```
RuleSet     ::= Rule [';' Rule]*
Rule        ::= ['Optional:' | 'Mandatory:'] Constraint [ExceptRule]*
Constraint  ::= TermList [Operator TermList]*
              | 'No' Constraint
              | 'Order of' TermList ['is' | 'is not'] TermList
              | 'Number of' TermList Operator TermList
              | 'if' Constraint 'then' Constraint
ExceptRule  ::= 'Except' Constraint
Operator    ::= [.]+  (e.g., 'is' | 'is not' | 'have' | 'after' |  '=' | 'Add ' | '...' )
TermList    ::= Term [', ' Term]*
Modifier    ::= Word (e.g., 'some' | 'each' | 'all' | 'first' | 'last' | '...')
Term        ::= '[' PLterm ']'
              | Modifier* PLterm
              | PLterm 'of' PLterm
Word        ::= [a-zA-Z]+
PLterm      ::= [.]+
```

**Note:** PLterm *denotes terminology in programming languages like Java;*
*'. ' means to match any single character except for newline.*
*'...' indicates that more words can be included if needed;*
*'*' means zero or more repetitions; '+' means one or more repetitions.*

Fig. 2. Comparison of coding rule representations (NL, linter configuration, and DSL)

## 3.1 DSL for Coding Rules from Coding Standards and Linter Configurations

*3.1.1 Motivating Examples for DSL Design.* Similar to compiler design [41, 48, 53], which translates high-level code into machine-readable form through intermediate representations such as abstract syntax trees or bytecodes, our approach aims to compile NL coding standards into linter configurations. To support this, we design a DSL that expresses coding rules in a tool-agnostic, structured, precise, and readable form. While both NL coding standards and machine-readable linter configurations can represent coding rules, they do not meet the requirements.

NL is tool-agnostic and readable but lacks structure and precision. A NL sentence of coding standards may or may not contain rules and can embed multiple layers of information, such as

rule types, objects being checked, and constraint types. For instance, in Figure 2, the sentences in the *NL box* illustrate this issue. In the Google Java coding standard, sentence *i* does not have rules, while sentence *j* has two rules. The two rules have mixed information: (1) "should always" and "must" indicate mandatory rules; (2) "block tag" and "these tags" refer to the four types of block tags being checked; (3) "appear in this order" and "not have" imply order and negation constraint relationships.

In contrast, machine-readable formats, such as XML, are structured and precise, yet tool-dependent and often too abstract to fully convey the intended semantics. In the *Linter Configuration Format box* of Figure 2, the Checkstyle configuration for sentence *j* consists of configuration names, option names and values. However, the configuration names, option names, and values are abstract and incomplete, not enough to convey the exact meaning. Moreover, linter configurations such as configuration names and formats are specific to Checkstyle and cannot be reused by other linters, which often use different settings and formats.

To provide a tool-agnostic, structured, precise and readable representation for coding rules, we design the DSL shown in the *domain-specific language* box from Figure 2. The corresponding DSL representations of sentence *j* are "*Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated]*" and "*Mandatory: No [EmptyDescription] for [@param, @return, @throws, @deprecated]*". The DSL independently expresses each rule, clearly separating rule types (e.g., *"Mandatory:"*), objects (enclosed in *"[]"*), and constraint relationships (e.g., *"Order of"* and *"No"*).

*3.1.2 Information Source for Designing DSL.* To design the DSL, we first select research objects of coding standards and linters. Java, with its mature ecosystem and rigorous coding practices [20], represents a language with long-standing conventions, making it an ideal choice. Given the Google coding standards' broad acceptance, comprehensive coverage, and frequent reference across software projects and the industry [12], we select the Google Java style guide as a representative coding standard. To enforce these standards, we select Checkstyle [7] as the linter for Java, as it is highly configurable, widely used, and has been extensively studied [34, 47, 67, 77]. To understand linter configurations, we refer to Checkstyle documentation [15], which provides detailed information about linter configurations. We emphasize that our choice of the Google Java style guide and Checkstyle is not aimed at targeting these specific coding standards or linters. Rather, we select them due to their broad adoption, comprehensive coverage, and representativeness, making them ideal entry points for designing the DSL.

*3.1.3 Process of Designing DSL.* We use a card sorting approach [64] to define DSL by analyzing a sample of 320 sentences with a confidence level of 95% and a confidence interval of 5 drawn from all sentences from both Google Java style guide and Checkstyle documentation. The process of designing DSL has two iterations and we used two evaluators. In the first iteration, we randomly sample 175 sentences with a confidence level of 95% and a confidence interval of 5 from 320 sentences. Two authors first independently represent each sentence using their designed DSLs, and then they discuss to construct a DSL. In the second iteration, two of the authors independently represent remaining sentences with the DSL. If the DSL is not enough to represent the rule of a sentence, they annotate the sentence with a brief description. They find that there are no sentences that cannot be represented using the DSL. We use Cohen's Kappa measure [52] to examine the agreement between the two authors. The Kappa value is 0.7, which indicates a high agreement between two authors. Finally, two authors discuss the disagreements to reach an agreement.

*3.1.4 DSL for Coding Standards and Linter Configurations.* Figure 2 shows the designed DSL. We only provide necessary explanation of the DSL grammar below. A complete explanation of the DSL grammar can be found in APPENDIX B.

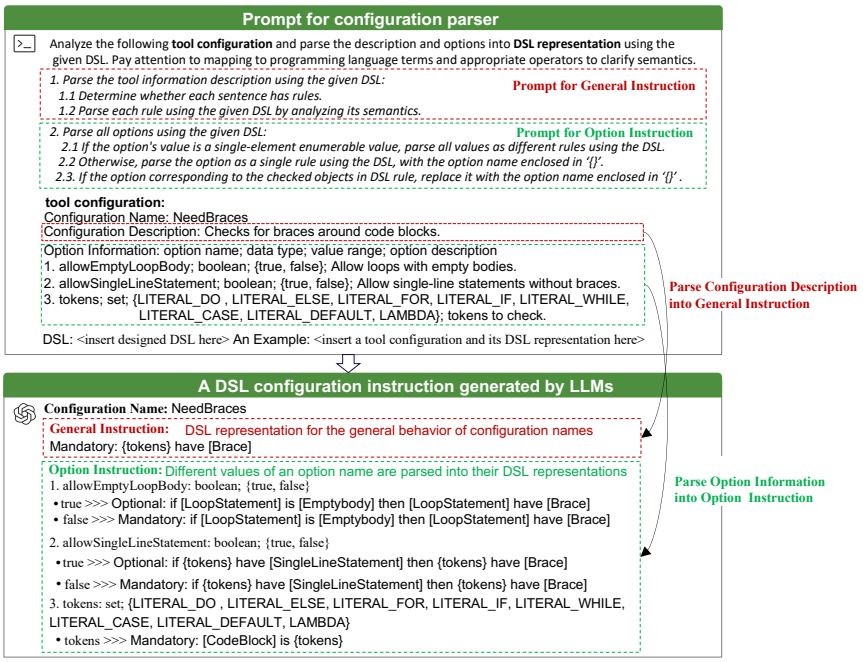



Fig. 3. Building a linter configuration into the DSL configuration instruction from the linter document

• *RuleSet* represents sets of *Rules*, where individual rules are separated by semicolons.
• A *Rule* comprises a *RuleType*, a *Constraint*, and optionally any number of *ExceptRule*. The *RuleType* can be 'Mandatory' or 'Optional', indicating whether the rule must be followed or is optional. The *ExceptRule* specifies exceptions to a *Constraint*. *Constraint* has five types. We explain the first two constraint relationships below, and the others are in APPENDIX B.
• *TermList [Operator TermList]**: Denotes the relationship that TermList must satisfy, expressed via operators. For example, "Checks for braces around code blocks" from the Checkstyle documentation is represented in the DSL as "*Mandatory: [CodeBlock] have [Brace]*".
• *'No' Constraint*: The *Constraint* is prohibited. For example, in the Figure 2, the NL description of the Google Java coding standard, "these tags must not have an empty description" is represented in the DSL as "*Mandatory: No [EmptyDescription] for [@param, @return, @throws, @deprecated]*".
• *Operator* denotes the relationship *TermLists* should satisfy.
• *TermList* is a list of *Terms* separated by commas.
• *PLterm* represents terminology specific to programming languages, such as "BlockTag" and "@param" in the DSL representation of the Figure 2. It can be any sequence of characters, excluding newlines.

### 3.2 Building Linter Configurations into a DSL Configuration Instruction Set

A linter configuration consists of two key components: a configuration name and its associated options. The name specifies the general purpose of the configuration, and the options control its specific behaviors. Therefore, we parse each linter configuration into a DSL instruction comprising the general instruction and the option instruction.

Building on the generative capabilities of LLMs, we prompt the LLM to parse a linter configuration into its general instructions and option-specific instructions in DSL. We provide an example to

clarify the task and the expected response format. Figure 3 presents the prompt and LLM response for *NeedBrace* configuration.

The general instruction and option instruction are detailed below.

• **General Instruction**: It denotes the general behavior of the configuration name. The configuration description summarizes the general functionality of the configuration name. Thus, we parse the configuration description into the general DSL instruction. Since the description often contains multiple sentences, some of which are code examples or explanations rather than rules, we first prompt the LLM to determine each sentence express coding rules, as shown in the step 1.1 of the prompt in Figure 3. Once identified, we then prompt the LLM to parse each coding rule using the DSL, as shown in the step 1.2 of the prompt in Figure 3.

• **Option Instruction**: It denotes different behaviors for different option values of a option name. For each option, different option values correspond to different behaviors. Thus, we generate a separate DSL representation for each value to accurately capture its corresponding behavior. Specifically, for finite enumerated values (e.g., boolean values true and false), we parse different values into different DSL rules, as shown in the prompt of step 2.1 in Figure 3. Otherwise, we parse the option into a single DSL rule by enclosing the option name in brackets, as shown in the prompt of step 2.2 in Figure 3. This is because parsing each option value for infinite or numerous values is impractical and redundant, enclosed the option name in brackets allows for later option value assignment. We also observe that some options specify the objects to be checked. Similar to the infinite-value case, we replace these objects with the option name in brackets, as shown in the prompt of step 2.3 in Figure 3. For example, the "allowSingleLineStatement" option only has two values: true and false. Therefore, we parse each value into a separate DSL rule. Since "tokens" option corresponds to the checked object "CodeBlock", we replace it with "{tokens}" for later option value assignment.

### 3.3 Compilation from NL Coding Standards to Linter Configurations

Directly using a single prompt to generate linter configurations for coding standards is inefficient, as this approach can reduce scalability, complicate debugging, and can easily exceed LLM token limits due to the large number of linter configurations included in the prompt. Inspired by compiler design, which translates high-level representations into machine code through stages like parsing, intermediate code generation, semantic analysis, and machine code generation, we split the compilation into five steps, where each step only focus on a task to ensure the correctness and scalability. The DSL and the DSL configuration instruction set provide a foundation for compiling a NL coding standard into the linter configurations. For each step, we provide an example for LLMs to better understand the task.

*3.3.1 NL to DSL Coding Standard Parser.* A natural language (NL) coding standard often contains non-rule sentences (e.g., examples or explanations) and may express multiple rules in a single sentence. To enable precise and structured interpretation, we parse the NL standard into the DSL rule representation as an intermediate representation for linter configuration generation.

We prompt the LLM with a template comprising the task prompt, the NL coding standard, the DSL grammar from Section 3.1.4, and an example pairing an NL coding standard with its DSL representation. Step 1 (green box) in Figure 4 illustrates the task prompt and the resulting DSL coding standard. For example, in Figure 4, the NL coding standard is parsed into the DSL coding standard comprising four DSL rules.

*3.3.2 Configuration Name Selector.* To generate configurations, it is natural to first identify the configuration name, followed by the specific options to be set. Directly inputting the entire DSL instruction set for name selection is unnecessary and may exceed the LLM's token limit. Since the

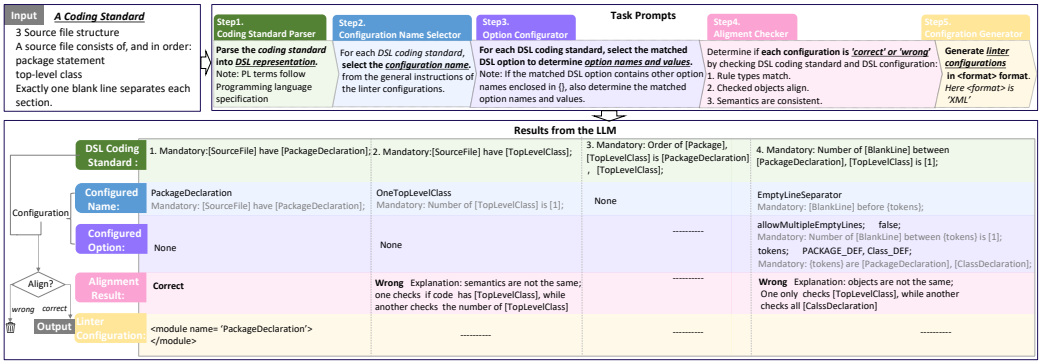



Fig. 4. Compilation from a NL coding standard to linter configuration

general instruction captures the overall behavior of the configuration, matching it with the DSL coding standard is sufficient to identify candidate configuration names.

We prompt the LLM with a template consisting of the task prompt, the DSL coding standard, general instructions of all linter configurations, and an example pairing the DSL coding standard with its corresponding configuration names. Step 2 in Figure 4 shows the task prompt and the configuration names selected by the LLM. For example, for the first DSL coding standard in Figure 4, the LLM responds with "PackageDeclaration" configuration name from Checkstyle.

*3.3.3 Option Configurator.* After selecting candidate configuration names for the DSL coding standard, we provide the LLM with the option instructions of configured names to determine the appropriate options. The LLM matches the DSL option with the DSL coding standard to identify relevant option names and values.

The prompt template consists of the task prompt, the DSL coding standard, option instructions of the selected configuration names, and an example pairing the DSL coding standard with its configured options. The step 3 in Figure 4 shows the task prompt and configured options from the LLM. For example, for the last DSL coding standard, the matching DSL option is "*Mandatory: Number of [BlankLine] between {tokens} is [1]*". Since the DSL option contains "tokens" option that enclosed in brackets, LLMs further extract the option name and set the value to "PACKAGE_DEF, CLASS_DEF".

*3.3.4 Alignment Checker.* Correct configurations are essential, as incorrect ones can lead linters to report misleading issues, wasting developer time and potentially leading to wrong code changes. Meanwhile, we observe that LLMs tend to prioritize finding as many configurations as possible. To ensure correctness, we further verify the consistency between the generated DSL configuration and the DSL coding standard across three aspects: rule type consistency, object consistency, and semantic alignment.

The prompt template consists of the task prompt, the DSL coding standard, the corresponding DSL configurations from general and option instructions, and an example pairing the DSL coding standard with its DSL configurations and alignment results. Step 4 of Figure 4 shows the task prompt and the alignment results generated by the LLM. For example, for the last DSL coding standard, the checked objects is not the same as the checked objects of the corresponding DSL option, so the configuration is wrong.

*3.3.5 Linter-Specific Configuration Generator.* After obtaining aligned configurations, we convert them into linter-specific formats. We use LLMs for their flexibility and ability to adapt across linters with varying schemas, avoiding the need to maintain separate generators.

Table 1. Results of DSL generation for Google Java coding standards and Checkstyle configurations.

| Name | #Num | #Samp | #DSL | Acc(%) | P(%) | R(%) | F1(%) |
|---|---|---|---|---|---|---|---|
| Java Style Guide | 68 | 68 | 182 | 89.7 | 96.2 | 100 | 98.0 |
| Checkstyle | 184 | 125 | 620 | 83.2 | 92.3 | 100 | 96.0 |

Note: #Num is the total number of coding standards or linter configurations. #Samp is the number of sampled coding standards or linter configurations. #DSL is the number of corresponding DSL representations generated by our approach.

The prompt template consists of task prompt, successfully verified configurations in Section 3.3.4, an example pairing with verified configurations, and linter-specific configurations. Step 5 of Figure 4 shows the task prompt and the linter-specific configurations. For example, only the first configuration is correct, so the final configuration uses the module "PackageDeclaration".

## 4 Evaluation

To evaluate our approach, we study five research questions:

**RQ1:** How effective are the DSL and the LLM in generating DSL representations for the Java coding standards and Checkstyle configurations?
**RQ2:** How effective is our approach in generating Checkstyle configurations for the Java coding standards?
**RQ3:** How do the different modules in our approach impact its effectiveness?
**RQ4:** How useful is our approach for developers in generating Checkstyle configurations for Java coding standards?
**RQ5:** Can our approach be effectively extended to other programming languages, coding standards, and linters?

### 4.1 RQ1: Effectiveness of DSL Representation Generation

*4.1.1 Motivation.* After we design DSL in Section 3.1, it is important to verify if the DSL and LLMs can effectively express coding standards and linter configurations.

*4.1.2 Approach.* We manually review the correctness of DSL representations for Google Java coding standards and Checkstyle configurations. If the total number of coding standards or linter configurations is fewer than 100, we examine all instances; otherwise, we apply random sampling with a 95% confidence level and a 5% margin of error [62]. For coding standards or linter configurations, two authors and two external experts, each with over six years of Java experience, independently classify each DSL representation as correct (a true positive), incorrect (a false positive), or missing (a false negative). We denote the number of true positives, false positives and false negatives as $TP$, $FP$ and $FN$. Since a coding standard or linter configuration typically involves multiple DSL representations, they further evaluate whether all DSL representations for a given coding standard or linter configuration are correct. Any inconsistencies are resolved through discussion. We employ four metrics: precision, recall, F1-score and accuracy. Precision is calculated as $P = \frac{TP}{TP+FP}$, recall as $R = \frac{TP}{TP+FN}$, and F1-score as $F1 = \frac{2\times P\times R}{P+R}$. Accuracy evaluates the proportion of coding standards or linter configurations for which all DSL rules are correctly generated. It is calculated as $Acc = \frac{\text{number of coding standards or linter configurations with correct DSL representation}}{\text{total number of coding standards or linter configurations}}$.
We employ the widely adopted state-of-the-art GPT-4o [4], whose performance and wide use in recent research make it an ideal prototype engine for our approach [37, 70, 71, 73, 78]. To minimize output variability of the LLM, the temperature is 0.

*4.1.3 Result.* Table 1 presents the accuracy, precision, recall, and F1-score for DSL representations of the Google Java coding standard and Checkstyle configurations. In total, we review 182 DSL representations across 68 coding standards and 620 DSL representations for 125 linter configurations.

For the Google Java style guide and Checkstyle, the precision, recall, and F1 scores exceed 90%, with accuracy above 80%. The results demonstrate that the designed DSL and the LLM can effectively represent a wide range of coding rules in both Java coding standards and Checkstyle configurations.
**Failure analysis of our approach**. By manually analyzing wrong and missing DSL representations for coding standards and linter configurations, we summarize two reasons as follows:
(1) Incomplete information from a sentence in coding standards or linter documentation can lead to incorrect DSL representations. For instance, in the Checkstyle configuration for "FallThrough", the description states, "The check honors special comments to suppress the warning, by default, the texts fallthru, fall thru, ..."; however, this description lacks complete information about which type of statements the comments apply to. In reality, the rule of Checkstyle pertains only to case blocks, but the DSL instruction set builder incorrectly parses the sentence as *"Optional: [Comment] matches [fallthru, fall thru, ...] suppress [Warning]"*, omitting the specification that the "[Comment] of [CaseBlock]".
(2) DSL parsers for coding standards or linter configurations may misinterpret whether a sentence defines rules, leading to missed or incorrect DSL representations. For example, the NL description in the Google Java coding standard, "A character means any Unicode code point" is an explanatory sentence instead of a rule. However, the DSL parser mistakenly interprets and parses it as a rule.

**Summary:** The high accuracy, F1-score, precison and recall shows the DSL and LLMs can effectively express Java coding standards and Checkstyle configurations into DSL representations.

### 4.2 RQ2: Effectiveness of Linter Configuration Generation

*4.2.1 Motivation.* The success of ChatGPT demonstrates the ability of LLMs to comprehend prompts and complete tasks [5, 23, 74]. We aim to compare our approach with existing LLM-based methods, highlighting how our approach effectively generating linter configurations for coding standards.

*4.2.2 Approach.* **Benchmark.** We create a benchmark of linter configurations for coding standards in the form of <coding standard, tool configuration>. Since no existing dataset is available, we manually construct Checkstyle configurations for the Google Java coding standard using 120 hours. Two authors and two external experts, each with over six years of Java programming experience, independently create these configurations and resolve any inconsistencies through discussion. Table 2 presents the statistics for the benchmark. For the Google Java coding standard, 28% of the 68 coding standards lacked configurations, while 72% had configurations. Among those with configurations, 10% contained only configuration names, 62% included both names and options, and 19% involved multiple configuration names. The result highlights the challenge in generating linter configurations, as many coding standards either lack corresponding configurations or require complex configurations with options or with multiple configuration names.
**Baselines.** Since linter documentation contains information at different levels, and Retrieval-Augmented Generation (RAG) [38] is an effective method for enhancing LLM capabilities, we design six baselines. The prompt is as shown in Figure 5. To determine the prompt, two authors and two external experts independently review the coding standards and linter configurations to draft the prompt. They then discuss their results to finalize the best prompt. Here we provide an example which is aligned with the example in our approach to maintain fairness. The LLM settings are consistent with those described in Section 4.1.2.

1. **Closed Book** (i.e., <tool information> is empty): The baseline directly uses the LLM to generate linter configurations for coding standards without providing additional context or knowledge. The

Table 2. Benchmark statistics for Checkstyle configurations on Google Java coding standard.

| Configuration Category for Coding Standard | Number |
|---|---|
| No Configuration | 19 |
| Has Configuration | 49 |
| Config with Only Configuration Name | 7 |
| Config with Configuration Name and Options | 42 |
| Config with Multiple Configurations | 13 |
| Total Coding Standards | 68 |

<tool information> of the prompt template is empty. It serves as a fundamental comparison point to assess the impact of incorporating additional information on effectiveness of our approach.

2. **Name** (i.e., <tool information> consists only of linter configuration names): To augment LLM's capabilities, leveraging additional information or knowledge with LLMs is a well-established practice [32, 51, 58, 80]. Linters for coding standards generally provide official documentation [15] that includes configuration names, descriptions, and option information. Given that varying levels of granularity in the information might affect the performance of LLMs, we explore different baselines by utilizing LLMs with varying levels of tool information granularity. In this baseline, the LLM generates configurations for coding standards using only the tool configuration names.
3. **Name+Desc** (i.e., <tool information> consists of linter configuration names and configuration descriptions): It uses the LLM with both the linter configuration names and their descriptions to generate configurations for coding standards.
4. **Name+Desc+Opts** (i.e., <tool information> consists of linter configuration names, descriptions and option information): It uses the LLM with the linter configuration names, their descriptions, and information of options to generate configurations for coding standards.
5. **RAG (Name+Desc)** (i.e., Retrieval-Augmented Generation with tool configuration names and descriptions): RAG is a method that enhances generation models by first retrieving relevant knowledge from knowledge bases and then integrating it into the generation process [38]. RAG has been widely used in several software engineering tasks [35, 38, 49, 81].
   For this baseline, we use official linter documentation, including configuration names and descriptions, as external knowledge. In the indexing phase, we encode this external knowledge into vector representations using the OpenAI embedding model ada-002 [14], and store these vectors in ChromaDB. During retrieval, we query the top-k semantically similar linter configuration names for a given coding standard. According to Jiang et al. [44], we set k to 10, considering that a higher k value can overwhelm models and hinder information extraction in long contexts. In the generation phase, we integrate the retrieved top-k linter configuration into the LLM to generate configurations for coding standards. The <tool information> of the prompt template refers to the configuration names and corresponding descriptions of the retrieved top-k linter configurations.
6. **RAG (Name+Desc+Opts)** (i.e., Retrieval-Augmented Generation with linter configuration names, descriptions, and option information): It follows the same process as the fifth baseline. The one difference is the inclusion of RAG, which leverages linter configuration names, descriptions, and option details as external knowledge, in conjunction with LLMs, to generate configurations for coding standards.

**Metrics.** For a coding standard, linter configuration typically includes the configuration name, option name and the corresponding option values. Consequently, for one metric, we evaluate effectiveness at three levels of granularity: configuration name, option name, and option value, as shown in Table 3.

We first use the accuracy metric, which measures the percentage of coding standards for which the approach correctly generates linter configurations. The formula of the accuracy of an approach

For the following **coding standard**, please generate **tool configurations** based on the following tool information.
1. Analyze each sentence of the coding standard to determine if exists corresponding configurations.
2. Ensure the generated configurations align with the specified coding standard.
**Coding Standard:** *<insert a coding standard here>*
**Tool Information:** *<insert tool documentation here>*
An Example: *<insert an example consisting of a coding standard and the corresponding tool configurations here>*

Fig. 5. Prompt template of baselines

Table 3. Results of generating Checkstyle configurations for Google Java coding standards.

| Approach | Config Name Level Metrics (%) | | | | Option Name Level Metrics(%) | | | | Option Value Level Metrics(%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc | P | R | F1 | Acc | P | R | F1 | Acc | P | R | F1 |
| Closed-book | 32.4 | 39.4 | 55.4 | 46.1 | 19.1 | 26.0 | 36.5 | 30.3 | 32.4 | 29.8 | 41.9 | 34.8 |
| Name | 39.7 | 16.8 | 74.3 | 27.4 | 23.5 | 11.6 | 51.4 | 19.0 | 32.4 | 12.8 | 56.8 | 20.9 |
| Name+Desc | 39.7 | 26.2 | 75.7 | 38.9 | 29.4 | 21.0 | 60.8 | 31.2 | 29.4 | 18.7 | 54.1 | 27.8 |
| Name+Desc+Opts | 48.5 | 23.4 | 78.4 | 36.0 | 48.5 | 23.0 | 77.0 | 35.4 | 38.2 | 19.0 | 63.5 | 29.2 |
| RAG (Name+Desc) | 14.7 | 24.4 | 73.0 | 36.6 | 13.2 | 21.7 | 64.9 | 32.5 | 13.2 | 19.0 | 56.8 | 28.5 |
| RAG (Name+Desc+Opts) | 25.0 | 31.8 | 73.0 | 44.3 | 25.0 | 31.8 | 73.0 | 44.3 | 13.2 | 18.8 | 43.2 | 26.2 |
| **Our Method** | **73.5** | **81.3** | **82.4** | **81.9** | **73.5** | **81.3** | **82.4** | **81.9** | **72.1** | **69.3** | **70.3** | **69.8** |
| Change (%) | ↑51.5 | ↑106.3 | ↑5.1 | ↑77.7 | ↑51.5 | ↑155.7 | ↑7.0 | ↑84.9 | ↑88.7 | ↑132.6 | ↑10.7 | ↑100.6 |

Note: Bold values represent the best performance across all approaches. Underlined values represent the best performance among the baselines. "Change" shows the minimum improvement over the baselines.

is $Acc = \frac{\text{number of coding standards with correct configurations}}{\text{number of coding standards}}$. Since a coding standard may have multiple linter configurations, we further refine our evaluation metrics. For a coding standard, a true positive occurs when a configuration given by an approach exists in the benchmark. A false positive occurs when a configuration given by an approach does not exist in the benchmark. A false negative occurs when a configuration not given by an approach exists in the benchmark. We denote the number of true positives, false positives and false negatives as $TP$, $FP$ and $FN$. We calculate the precision as $P = \frac{TP}{TP+FP}$, the recall as $R = \frac{TP}{TP+FN}$, and the F1-score as $F1 = \frac{2\times P\times R}{P+R}$ [75]. The $Acc$ is measured at the level of all configurations for a coding standard, and $P$, $R$, and $F1$ are measured at the level of each configuration for a coding standard.

*4.2.3 Result.* Table 3 presents the accuracy, precision, recall, and F1-score of our approach and the baselines. Our approach achieves 72.1%~73.5% for accuracy, 69.3%~81.3% for precision, 70.3%~82.4% for recall, and 69.8%~81.9% for F1-score across three levels of granularity. Our approach consistently outperforms the baselines in accuracy, precision, recall, and F1-score across the three levels of granularity, with minimum improvements of 51.5%, 106.3%, 5.1%, and 77.7%, respectively. Among the baselines, precision is notably low, indicating their tendency to generate incorrect configurations. Even with traditional augmentation methods, such as incorporating tool documentation or the RAG strategy, precision remains low and may even degrade. In contrast, our approach significantly improves precision (by over 100%) while maintaining strong recall, showing its effectiveness in generating accurate linter configurations.

**Failure analysis of our approach.** To explore why our approach cannot generate configurations for coding standards, we examine configurations in the benchmark but our approach does not generate. The primary reason is that accurate linter configuration generation may require strong semantic inference and domain knowledge, which our approach lacks. For example, the Java coding standard for a summary fragment, "...It does not begin with 'A@code Foo...' ", is represented in DSL as *"Mandatory: No [SummaryFragment] begin with [A@code Foo...]"*. It should map to the "forbiddenSummaryFragments" option in the SummaryJavadoc configuration, represented in DSL as *"Mandatory: [JavadocSummary] not have {forbiddenSummaryFragments}"*. The option is a regular expression that can be set to match strings starting with "[A@code Foo...]". However, our DSL instruction configurator fails to recognize it due to limited reasoning capability.

Table 4. Results of ablation study.

| Approach | Config Name Level Metrics (%) | | | | Option Name Level Metrics(%) | | | | Option Value Level Metrics(%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc | P | R | F1 | Acc | P | R | F1 | Acc | P | R | F1 |
| **Our Method** | **73.5** | **81.3** | 82.4 | **81.9** | **73.5** | **81.3** | 82.4 | **81.9** | **72.1** | **69.3** | 70.3 | **69.8** |
| w/o DSL (*Change*) | 61.8 (↓18.9) | 67.0 (↓21.3) | 76.7 (↓7.4) | 71.5 (↓14.5) | 60.3 (↓21.9) | 65.3 (↓24.5) | 75.6 (↓9.0) | 70.1 (↓16.8) | 51.5 (↓40.0) | 52.2 (↓32.8) | 60.2 (↓16.8) | 55.9 (↓24.9) |
| w/o Selector (*Change*) | 60.3 (↓21.9) | 72.9 (↓11.5) | 47.3 (↓74.2) | 57.4 (↓42.7) | 58.9 (↓24.8) | 70.8 (↓14.8) | 45.9 (↓79.5) | 55.7 (↓47.0) | 57.3 (↓25.8) | 66.7 (↓3.9) | 43.2 (↓62.7) | 52.5 (↓33.0) |
| w/o Checker (*Change*) | 73.5 (0) | 75.0 (↓8.4) | **82.9** (↑0.6) | 78.8 (↓3.9) | 72.1 (↓1.9) | 73.8 (↓10.1) | **82.9** (↑0.6) | 78.1 (↓4.9) | 69.1 (↓4.3) | 65.5 (↓5.8) | **72.1** (↑2.5) | 68.6 (↓1.7) |

**Summary:** Our approach achieves superior accuracy, F1-score, precision, and recall compared to other baselines in generating Checkstyle configurations for Java coding standards, particularly at the fine-grained level and in terms of precision.

### 4.3 RQ3: Ablation Study

*4.3.1 Motivation.* The effectiveness of our approach motivates us to assess the contribution of each module involved in the DSL-driven and AI chain compilation approach.

*4.3.2 Approach.* We conduct an ablation study with three variants based on our approach. The LLM settings are the same as those described in Section 4.2.2.
**w/o DSL:** Disables the generation of DSL representations (parsers) for coding standards and linter configurations.
**w/o Selector:** Skips the configuration name selection step and directly generates configuration names and options.
**w/o Checker:** Omits the alignment checker that verifies whether the generated configurations conform to the coding standards.

*4.3.3 Result.* Table 4 shows the result of the ablation study. Removing the DSL module(**w/o DSL**) results in a substantial decrease in accuracy (↓18.9–40.0%) and F1-score (↓14.5–24.9%), highlighting the crucial role of the structured, readable, and tool-agnostic DSL in guiding accurate configuration generation. The configuration name selector module also proves essential: its removal(**w/o Selector**) leads to severe drops in recall (↓74.2% at the configuration name level and ↓62.7% at the option value level), suggesting its importance in narrowing the generation space and improving coverage. While the alignment checker (**w/o Checker**) has a smaller overall impact, it still improves precision, with its absence causing up to a 10.1% drop across all levels. The checker enhances the reliability of the configurations. The slight increase or stability in recall is expected, as the checker filters incorrect configurations rather than expanding coverage. Given that incorrect configurations can lead to misleading enforcement, precision is more critical than recall in this context.

**Summary:** The ablation results for Checkstyle configuration generation in Java coding standards highlight the critical role of DSL-driven representations and AI-chain compilation.

### 4.4 RQ4: Usefulness of Our Approach

*4.4.1 Motivation.* Given the challenges of manual linter configuration for coding standards in Section 2, we evaluate whether our approach enables developers to generate these configurations more effectively.

*4.4.2 Approach.* We randomly select six various coding standards from the Google Java coding standard: two without corresponding Checkstyle configurations, two with a single configuration, and two with multiple configurations. Fourteen participants (seven PhD students and seven industry

Table 5. Results of user study.

| Groups | Correctness (%) | | | Time(s) |
|---|---|---|---|---|
| | Config Name Level | Option Name Level | Option Value Level | |
| G1 | 45.2 | 33.3 | 23.8 | 337.7 |
| G2 | 95.2 | 95.2 | 95.2 | 226.5 |
| Change(%) | ↑110.6 | ↑185.9 | ↑300.0 | ↓49.1 |

professionals, each with 1–5 years of Checkstyle experience) are recruited. Each participant is asked to write linter configurations for each coding standard in ten minutes. The participants are paired with similar backgrounds, one assigned to the control group (G1) and the other to the experimental group (G2). G1 is only given the coding standards, while G2 is also provided with configurations automatically generated by our approach as references. Participants are free to use the Internet or LLMs during the task, mirroring real-world conditions. This setup allows us to evaluate whether our generated configurations can help developers effectively generate configurations in the wild. After the study, we conduct a short survey to collect feedback on participant behavior. We develop two separate online user study platforms for G1 and G2 to complete the task [17, 18].

To evaluate the performance difference between two groups, we compute their completion time and answer correctness. The completion time is automatically recorded during the study. Then we calculate the answer correctness with the percentage of questions answered correctly for each group (G1 and G2) at three levels: module name, option name, and option value. We use Wilcoxon signed-rank test [72] to determine if the performance difference between two groups is statistically significant at the confidence level of 95%.

*4.4.3 Result.* Table 5 presents the results of the user study. *Correctness* refers to the average percentage of correctly answered questions for each group across different configuration levels. *Time* denotes the average completion time of a question for each group. The Wilcoxon signed-rank test shows statistically significant differences in correctness and completion time between G1 and G2 (p-value < 0.05), indicating that our approach helps developers configure linters more accurately and quickly.

Participants in G1, who lacked reference configurations, exhibit declining correctness as the configuration granularity increased—from 45.2% at the configuration name level to 23.8% at the option value level. Interviews reveal that novice Checkstyle users struggled to identify which modules corresponded to which coding standards, citing the overwhelming number of available modules. Experienced users reported that identifying configuration names was relatively manageable, but accurately setting options was more challenging due to subtle details in the coding standards, especially under the 10-minute time constraint.

G1 participants complete each task on average 337.7 seconds (five minutes and 37 seconds), which is significantly shorter than ten minutes. However, interviews reveal that this was not due to confidence or efficiency. Participants often ignore or misunderstand details in coding standards or linter configurations, such as checked objects of coding rules. Moreover, when they struggle to find relevant modules or precise option settings, they give up or submit answers to avoid running out of time.

In contrast, group G2, equipped with our generated configuration references, achieves consistently high correctness across all levels (95.2% at all granularities). Compared to G1, the minimum improvement in correctness exceeds 100%. Moreover, G2 complete each task in an average of 226.5 seconds, 49.1% faster than G1. All participants praise the references for reducing lookup time and improving configuration accuracy, highlighting their value in both correctness and efficiency. Across all 42 tasks completed by seven participants, only two are answered incorrectly, both by novices. These errors arose when unsupported coding standards were erroneously assumed to be

Table 6. Benchmark statistics for ESLint configurations on Google JavaScript coding standard.

| Configuration Category for Coding Standard | Number |
|---|---|
| No Configuration | 89 |
| Has Configuration | 60 |
| Config with Only Configuration Name | 15 |
| Config with Configuration Name and Options | 45 |
| Config with Multiple Configurations | 16 |
| Total Number of Coding Standards | 149 |

covered by Checkstyle. For example, for an "empty block" coding standard about brace formatting of empty blocks, a participant mistakenly generated the *EmptyBlock* configuration. Although this configuration does not exist in our reference, the participant is misled by the keyword match in the module name and overlooks that *EmptyBlock* actually prohibits empty statements in blocks, thus incorrectly considering it a valid configuration.

**Summary:** The Checkstyle configurations generated by our approach effectively help developers configure linters more accurately and efficiently.

### 4.5 RQ5: Generality of Our Approach

*4.5.1 Motivation.* We are interested in whether our approach can be effectively extended to other programming languages, coding standards and linters.

*4.5.2 Approach.* To robustly assess generality while keeping benchmarking effort manageable, we prioritize selecting different programming languages, linters, and coding standards based on the complexity of the task. We choose JavaScript, a dynamically-typed, interpreted language primarily used for web development, contrasting with Java, a strongly-typed, compiled language used in enterprise and mobile applications. For linters, we select ESLint, which uses JSON, in contrast to Checkstyle's XML-based configurations. We use the Google JavaScript style guide as the Javascript coding standard for our study. The Google JavaScript style guide covers a broader set of rules, with twice as many coding standards as the Java coding standard, providing a richer dataset to evaluate the generality of our approach. Additionally, although the JavaScript style guide is also from Google, their natural language descriptions differ due to the distinct programming languages. Even though for similar coding rules, their description is different. For instance, the Java standard states, "The column limit (Section 4.4, Column limit: 100) does not apply to import statements", while JavaScript's standard says, "Import statements must not be line-wrapped and are therefore an exception to the 80-column limit". When applying the approach to other programming languages, coding standards, and linter configurations, we only need to provide the coding standards, linter documentation, and the linter configuration format (e.g., JSON) to directly utilize our approach. The evaluation approach, dataset, baselines, and metrics are consistent with those described in Sections 4.1.2 and 4.2.2.

Table 6 shows the benchmark of ESLint configurations for JavaScript coding standards, which took about 182 hours to create. For the Google JavaScript style guide, 59.7% of the 149 coding standards lacked configurations, 40.3% had configurations, of which 30.2% required options or multiple configuration names. The benchmark also highlight the challenge in generating linter configurations, as many coding standards either lack corresponding configurations or require complex configurations with options or with multiple configuration names.

*4.5.3 Result.* Table 7 presents the accuracy, precision, recall, and F1-score for generating DSL representations of JavaScript coding standards and ESLint configurations. We review 393 DSL representations for 108 coding standards and 540 for 166 tool configurations. Precision, recall, and

Table 7. Results of DSL generation for JavaScript coding standards and ESLint configurations.

| Name | #N | #S | #DSL | Acc(%) | P(%) | R(%) | F1(%) |
|---|---|---|---|---|---|---|---|
| JS Style Guide | 149 | 108 | 393 | 84.3 | 94.6 | 99.5 | 97.0 |
| ESLint | 291 | 166 | 540 | 84.9 | 92.7 | 99.4 | 96.0 |

Table 8. Results of generating ESLint configurations for Google JavaScript coding standards.

| Approach | Config Name Level Metrics(%) | | | | Option Name Level Metrics(%) | | | | Option Value Level Metrics(%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc | P | R | F1 | Acc | P | R | F1 | Acc | P | R | F1 |
| Closed-book | 28.9 | 21.3 | 68.5 | 32.5 | 24.8 | 14.1 | 45.0 | 21.4 | 24.8 | 12.4 | 39.8 | 18.9 |
| Name | 20.8 | 12.8 | 77.8 | 22.0 | 16.8 | 8.5 | 51.9 | 14.6 | 16.1 | 7.8 | 47.2 | 13.3 |
| Name+Desc | 36.2 | 24.3 | 64.8 | 35.4 | 28.9 | 16.7 | 44.4 | 24.2 | 28.2 | 15.6 | 41.7 | 22.7 |
| Name+Desc+Opts | - | - | - | - | - | - | - | - | - | - | - | - |
| RAG (Name+Desc) | 14.8 | 11.7 | 77.8 | 20.4 | 10.7 | 7.4 | 49.1 | 12.8 | 10.7 | 6.8 | 45.4 | 11.9 |
| RAG (Name+Desc+Opts) | 18.1 | 9.6 | **80.6** | 17.2 | 14.8 | 6.6 | 55.6 | 11.8 | 14.1 | 6.2 | 51.9 | 11.1 |
| **Our Method** | **75.2** | **84.4** | 70.4 | **76.8** | **71.1** | **75.6** | **63.0** | **68.7** | **71.1** | **75.6** | **63.0** | **68.7** |
| Change (%) | ↑107.7 | ↑247.3 | ↓12.7 | ↑116.9 | ↑146.0 | ↑352.7 | ↑13.3 | ↑183.9 | ↑152.1 | ↑384.6 | ↑21.4 | ↑202.6 |

Note: '-' means inapplicable results due to LLMs' token limits.

F1 scores consistently exceed 90%, with accuracy above 80%. The results highlight the generality of our approach in generating DSL representations.

For ESLint configurations of JavaScript coding standards, our approach achieves 71.1%~75.2% for accuracy, 75.6%~84.4% for precision, 63.0%~70.4% for recall, and 68.7%~76.8% for F1-score across three levels of granularity. Similar to the results for Checkstyle configurations of Java coding standards, our approach consistently outperforms the baselines in accuracy, precision, and F1-score across all granularity levels, with minimum improvements of 107.7%, 247.3% and 116.9%. While recall decreases at the configuration name level (-12.7%), it improves at the option name and option value levels (+13.3% and +21.4%, respectively). Notably, precision is more critical than recall in tool configurations, ensuring correctness and minimizing deployment errors. As with Checkstyle configurations, our approach significantly improves precision (by more than 100%) while maintaining strong recall.

**Summary:** The high accuracy, F1-score, precision and recall of ESLint configurations for JavaScript coding standards illustrate that our approach can be effectively extended to other programming languages, coding standards and linters.

## 5 Discussion

### 5.1 Implications

Our approach, detailed in Section 3 and evaluated in Section 4, demonstrates promising performance and scalability potential in linter configuration generation for coding standards. **It can be readily adopted by software companies, large-scale projects, and individual developers to generate and maintain configurations for coding standards**. To apply it to other programming languages, coding standards, or linters, developers only need to provide the coding standards, linter documentation, and configuration format (e.g., XML).

The design core is the DSL-driven compilation pipeline with LLMs. Inspired its effectiveness, a **new insight for solving domain-related tasks** is that, instead of leaving everything to the LLM or fine-tuning LLMs with datasets, it is crucial to **design a lightweight DSL that models knowledge and information** in a structured, precise, and readable way. Since natural language is free-form, unstructured, and often ambiguous, designing the DSL can enhance the clarity and precision of communication with LLMs. By building on LLMs' strengths in generation, we can have LLMs function more as a subroutine instead of the central driver, ultimately improving the ability to address domain-specific tasks.

Meanwhile, our compilation pipeline leverages core compiler concepts—such as syntax parsing, intermediate code generation, semantic analysis, and machine code generation—to ensure modularity, reusability, and extensibility. For example, if coding standards have incomplete information, we could add a module for information completion before the NL coding standard parser. If we want to enhance semantic reasoning capabilities, we could introduce a reasoning module before or during the DSL instruction configurator. Besides, unlike traditional compilers, we leverage LLMs to enable fine-grained linter configuration. **Future research can explore the potential of LLMs as compilers** independent of programming languages, operating systems, and hardwares.

Finally, the approach and benchmark can assist **future research in conducting comparative studies** on the capabilities of various linters and their alignment with coding standards. They can also **facilitate the collaborative development of linters and coding standards**. For example, our benchmarks uncover limitations in existing linters: 27.9% of coding standards from the Google Java coding standard and 59.7% of coding standards from the Google JavaScript coding standard are unsupported by Checkstyle and ESLint in Section 4.2.2, Section 4.5.2, Table 2 and Table 6. We further analyze the unsupported coding standards in linters and identify two reasons. First, coding standards may require deep code analysis beyond current linters' capabilities. For example, a Java coding standard that static imports must not reference static nested classes requires cross-file resolution, which Checkstyle cannot perform. Second, a single conceptual coding standard may have multiple syntactic variants, but linters typically support only some of them. For instance, the JavaScript coding standard specifies indentation for continuation lines, yet ESLint provides no configuration for it. These unsupported standards can be reasonably supported, but doing so would demand significant implementation effort and extensions to linter configuration designs. Researchers and linter developers can use these findings to prioritize improvements and extend linter capabilities for better alignment with coding standards.

### 5.2 Threats to Validity

**Internal Validity:** A possible threat to internal validity is the presence of human errors in the manual evaluation in Section 4. To reduce the impact of the problem, each instance was evaluated by at least two authors and two external experts. In case of disagreement, all the evaluators revisited the instances to reach a consensus.

**External Validity:** One potential threat to external validity lies in our selection of programming languages, coding standards, and linter configurations. However, our goal is to evaluate the feasibility and generalizability of the proposed method, not to achieve exhaustive coverage. Our experiments validate its effectiveness across Java and JavaScript, the Google Java and JavaScript coding standards with different coding standards and description, and Checkstyle and ESLint with different linter formats. This design enables us to demonstrate that our approach is not tied to a specific programming language, coding standards, or linters. Benchmark construction is another challenge, requiring substantial effort—we spent 302 hours manually creating these benchmarks. In the future, our approach can help construct additional benchmarks, enabling broader coverage of programming languages, coding standards, and linters.

## 6 Related Work

**Studies on Coding Standards.** Coding standards evolve as organizations refine best practices, adopt new paradigms, and phase out outdated rules [20, 24, 26, 33, 33, 55, 57, 61, 63]. Abdallah et al. [20] analyzed 20 widely-used Java coding standards from 1996 to 2016. Their study revealed that organizations often develop their own standards, which evolve over time through the addition of new coding standards, the removal of outdated ones, and the updating to existing coding standards. Among these, the SUN and Google standards were particularly effective. Butler et al. [26]

investigated Java naming conventions and found that while developers generally adhere to naming conventions, compliance with specific guidelines varies significantly. Abdallah et al. [20] analyzed 20 most popular Java coding standards from 1996 to 2016, highlighting the continuous addition, removal, and revision of coding standards. Beyond empirical studies, researchers have explored mining coding standards [21, 27, 43, 45, 50, 63]. Allamanis et al. [21] introduced NATURALIZE, leveraging statistical NLP to infer coding style consistency. Unlike prior work, we propose a DSL to express coding rules independently of programming languages, coding standards and linters.

**Studies on Linters for Coding Standards.** Numerous linter tools have been developed to improve code quality and reduce developer effort [16, 21, 29, 31, 42, 45, 46, 54, 56, 68, 77]. Well-known researched static analysis tools include FindBugs [11], Checkstyle [7] and ESLint [9]. Beyond tool development, studies have examined developer usage and challenges in adopting these tools [22, 28, 65, 66, 79]. Beller et al. [22] found that developers frequently modify default configurations and prioritize minimizing false positives. Tómasdóttir et al. [66] analyzed over 9,500 ESLint configurations and surveyed developers, revealing that evolving coding standards complicate tool adoption due to inconsistencies in rule interpretation and integration challenges. Unlike prior work, which focuses on tool usage and static analysis, we generate tool configurations to relieve developers from the burden of manual tool configuration.

**Studies on DSL.** Domain-specific languages (DSLs) offer significant benefits by providing a high-level abstraction for specifying problems within particular domains [25, 30, 36, 40, 59, 60]. Historically, translating natural language into DSLs has been complex. Desai et al. [30] developed a synthesis algorithm to translate English sentences into DSL programs, incorporating a training phase to learn a dictionary and weights for the algorithm. With the advent of LLMs, Gandhi et al. [36] demonstrated how LLMs can translate natural language inputs into DSL programs that interact with application APIs. Unlike these approaches, we start by designing a tool-independent DSL structurally and precisely represent coding rules from coding standards and linter configurations, followed by utilizing an AI-based compilation pipeline to generate linter configurations for coding standards.

## 7 Conclusion

We design a domain-specific language (DSL) and employ an AI chain compilation to automate the generation of linter configurations for coding standards. The DSL express coding rules of coding standards and linter configurations in a tool-independent, structured, precise and readable way. The AI chain transpilation ensures the generation of accurate and fine-grained linter configurations. Experiments show the effectiveness, usefulness and generality of the designed DSL and our approach. In the future, we will employ our approach to assist constructing more benchmarks with more programming languages, coding standards and linters, followed by empirical studies to align coding standards with linters, fostering collaborative development of coding standards and linters.

## 8 Data Availability

The source code and data can be found here [19].

## Acknowledgment

This research was supported by the National Natural Science Foundation of China under Grant 62232014.

## References

[1] 2023. *Alibaba Coding Guidelines*. https://github.com/alibaba
[2] 2023. *Awesome Guidelines*. https://github.com/Kristories/awesome-guidelines

[3] 2023. *Gitlab Coding Standards.* https://docs.gitlab.com/development/contributing/style_guides/
[4] 2023. *GPT.* https://platform.openai.com/docs/guides/gpt
[5] 2023. *Introducing ChatGPT.* https://chat.openai.com/
[6] 2024. *Automated docstring and comments spell check.* https://stackoverflow.com/questions/27162315
[7] 2024. *CheckStyle.* https://github.com/checkstyle/checkstyle
[8] 2024. *Disable a particular Checkstyle rule for a particular line of code?* https://stackoverflow.com/questions/4023185
[9] 2024. *ESLint.* https://github.com/eslint/eslint
[10] 2024. *ESLint: how to spaced-comment exceptions on VSCode folding regions comments?* https://stackoverflow.com/questions/62469552
[11] 2024. *FindBugs.* https://findbugs.sourceforge.net/
[12] 2024. *Google Style Guide.* https://github.com/google/styleguide
[13] 2024. *How to remove semicolons in prettier?* https://stackoverflow.com/questions/45404823
[14] 2024. *New and Improved Embedding Model.* https://openai.com/index/new-and-improved-embedding-model/
[15] 2024. *Offical Documentation of Checkstyle.* https://checkstyle.sourceforge.io/checks.html
[16] 2024. *PMD.* https://pmd.github.io
[17] 2025. *online user study for control group(G1).* https://anynomousaccount.github.io/linterconfiguserstudy2
[18] 2025. *online user study for experimental group(G2).* https://idiomaticrefactoring.github.io/userstudylinterconfig-1
[19] 2025. *Replication Package.* https://doi.org/10.5281/zenodo.16732288
[20] Mohammad MA Abdallah and Mustafa M Al-Rifaee. 2017. Java standards: A comparative study. *International Journal of Computer Science and Software Engineering* 6, 6 (2017), 146.
[21] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. 281–293. https://doi.org/10.1145/2635868.2635883
[22] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481. https://doi.org/10.1109/SANER.2016.105
[23] Abhiram Bellur, Fraol Batole, Mohammed Raihan Ullah, Malinda Dilhara, Yaroslav Zharov, Timofey Bryksin, Kai Ishikawa, Haifeng Chen, Masaharu Morimoto, Takeo Hosomi, et al. 2025. Together We Are Better: LLM, IDE and Semantic Embedding to Assist Move Method Refactoring. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–13. https://doi.org/10.1109/icsme64153.2025.00046
[24] Cathal Boogerd and Leon Moonen. 2008. Assessing the value of coding standards: An empirical study. In *2008 IEEE International conference on software maintenance*. IEEE, 277–286. https://doi.org/10.1109/icsm.2008.4658076
[25] Alexandre Bragança, Isabel Azevedo, Nuno Bettencourt, Carlos Morais, Diogo Teixeira, and David Caetano. 2021. Towards supporting SPL engineering in low-code platforms using a DSL approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 16–28. https://doi.org/10.1145/3486609.3487196
[26] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. Investigating naming convention adherence in Java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 41–50. https://doi.org/10.1109/icsm.2015.7332450
[27] Uriel Ferreira Campos, Guilherme Smethurst, João Pedro Moraes, Rodrigo Bonifácio, and Gustavo Pinto. 2019. Mining rule violations in javascript code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 195–199. https://doi.org/10.1109/msr.2019.00039
[28] 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 (ASE'16)*. ACM, 332–343. https://doi.org/10.1145/2970276.2970347
[29] Mihai Christodorescu and Somesh Jha. 2003. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*. https://doi.org/10.21236/ada449067
[30] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 345–356. https://doi.org/10.1145/2884781.2884786
[31] BV Tishantha Dilruk. 2019. Coding standard violation detection by pattern analysis. (2019).
[32] Yan Ding, Xiaohan Zhang, Saeid Amiri, Nieqing Cao, Hao Yang, Andy Kaminski, Chad Esselink, and Shiqi Zhang. 2023. Integrating action knowledge and LLMs for task planning and situation handling in open worlds. *Autonomous Robots* 47, 8 (Aug. 2023), 981–997. https://doi.org/10.1007/s10514-023-10133-5
[33] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension (ICSE '18)*. ACM, 277–285. https://doi.org/10.1145/3196321.3196342

[34] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, 65–73. https://doi.org/10.1145/3105726.3106182

[35] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24)*. ACM, 6491–6501. https://doi.org/10.1145/3637528.3671470

[36] Apurva Gandhi, Thong Q Nguyen, Huitian Jiao, Robert Steen, and Ameya Bhatawdekar. 2023. Natural Language Commanding via Program Synthesis. *arXiv preprint arXiv:2306.03460* (2023).

[37] Yi Gao, Xing Hu, Xiaohu Yang, and Xin Xia. 2025. Automated Unit Test Refactoring. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE033 (June 2025), 21 pages. https://doi.org/10.1145/3715750

[38] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).

[39] Boryana Goncharenko and Vadim Zaytsev. 2016. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE '16)*. ACM, 90–104. https://doi.org/10.1145/2997364.2997386

[40] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. 2008. DSLs: the good, the bad, and the ugly. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. 791–794. https://doi.org/10.1145/1449814.1449863

[41] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. 2012. *Modern Compiler Design*. Springer New York. https://doi.org/10.1007/978-1-4614-4699-6

[42] Rowan Hart, Brian Hays, Connor McMillin, El Kindi Rezig, Gustavo Rodriguez-Rivera, and Jeffrey A. Turkstra. 2023. Eastwood-Tidy: C Linting for Automated Code Style Assessment in Programming Courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. ACM, 799–805. https://doi.org/10.1145/3545945.3569817

[43] Andre Hora, Nicolas Anquetil, Stephane Ducasse, and Marco Tulio Valente. 2013. Mining system specific rules from change patterns. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 331–340. https://doi.org/10.1109/wcre.2013.6671308

[44] Ziyan Jiang, Xueguang Ma, and Wenhu Chen. 2024. LongRAG: Enhancing Retrieval-Augmented Generation with Long-context LLMs. *arXiv preprint arXiv:2406.15319* (2024).

[45] Corentin Latappy, Quentin Perez, Thomas Degueule, Jean-Rémy Falleri, Christelle Urtado, Sylvain Vauttier, Xavier Blanc, and Cédric Teyton. 2023. MLinter: Learning Coding Practices from Examples—Dream or Reality?. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 795–804. https://doi.org/10.1109/saner56733.2023.00092

[46] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX security symposium*, Vol. 14. 18–18.

[47] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. 2022. Styler: learning formatting conventions to repair Checkstyle violations. *Empirical Software Engineering* 27, 6 (Aug. 2022). https://doi.org/10.1007/s10664-021-10107-0

[48] Kenneth C Louden. 1997. *Compiler construction: principles and practice*. PWS Publishing Co. https://doi.org/10.5555/523017

[49] Adam Mackay. 2024. Test Suite Augmentation using Language Models-Applying RAG to Improve Robustness Verification. In *ERTS2024*.

[50] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. Style-Analyzer: Fixing Code Style Inconsistencies with Interpretable Unsupervised Algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 468–478. https://doi.org/10.1109/msr.2019.00073

[51] Ariana Martino, Michael Iannelli, and Coleen Truong. 2023. *Knowledge Injection to Counter Large Language Model (LLM) Hallucination*. Springer Nature Switzerland, 182–185. https://doi.org/10.1007/978-3-031-43458-7_34

[52] Marry L. McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia Medica* (2012), 276–282. https://doi.org/10.11613/bm.2012.031

[53] Steven Muchnick. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

[54] Naoto Ogura, Shinsuke Matsumoto, Hideaki Hata, and Shinji Kusumoto. 2018. Bring your own coding style. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 527–531. https://doi.org/10.1109/saner.2018.8330253

[55] Naelson Oliveira, Marcio Ribeiro, Rodrigo Bonifacio, Rohit Gheyi, Igor Wiese, and Baldoino Fonseca. 2022. Lint-Based Warnings in Python Code: Frequency, Awareness and Refactoring. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 208–218. https://doi.org/10.1109/scam55253.2022.00030

[56] Hayatou Oumarou, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Kolyang Dina Taiwe. 2015. Identifying the exact fixing actions of static rule violation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 371–379. https://doi.org/10.1109/saner.2015.7081847

[57] Randall Pruim, Maria-Cristiana Gîrjău, and Nicholas J. Horton. 2023. Fostering Better Coding Practices for Data Scientists. *Harvard Data Science Review* 5, 3 (2023). https://doi.org/10.1162/99608f92.97c9f60f

[58] Xiaoxue Ren, Xinyuan Ye, Yun Lin, Zhenchang Xing, Shuqing Li, and Michael R. Lyu. 2023. API-Knowledge Aware Search-Based Software Testing: Where, What, and How. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. ACM, 1320–1332. https://doi.org/10.1145/3611643.3616269

[59] Benoît Ries, Alfredo Capozucca, and Nicolas Guelfi. 2018. Messir: a text-first DSL-based approach for UML requirements engineering (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18)*. ACM, 103–107. https://doi.org/10.1145/3276604.3276614

[60] Christian Schäfer, Thomas Kuhn, and Mario Trapp. 2011. A pattern-based approach to DSL development. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11 (SPLASH '11)*. ACM, 39–46. https://doi.org/10.1145/2095050.2095058

[61] Andrew J. Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '20)*. ACM, 1–11. https://doi.org/10.1145/3382494.3410680

[62] Ravindra Singh and Naurang Singh Mangat. 1996. *Systematic Sampling*. Springer Netherlands, 145–164. https://doi.org/10.1007/978-94-017-1404-4_6

[63] Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia. 2011. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 504–507. https://doi.org/10.1109/icsm.2011.6080819

[64] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[65] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie van Deursen. 2017. Why and how JavaScript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 578–589. https://doi.org/10.1109/ase.2017.8115668

[66] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie van Deursen. 2020. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering* 46, 8 (Aug. 2020), 863–891. https://doi.org/10.1109/tse.2018.2871058

[67] Eric Torunski, M. Omair Shafiq, and Anthony Whitehead. 2017. Code style analytics for the automatic setting of formatting rules in IDEs: A solution to the Tabs vs. Spaces Debate. In *2017 Twelfth International Conference on Digital Information Management (ICDIM)*. IEEE, 6–14. https://doi.org/10.1109/icdim.2017.8244675

[68] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 329–331. https://doi.org/10.1109/csmr.2008.4493342

[69] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[70] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael Lyu. 2025. Divide-and-Conquer: Generating UI Code from Screenshots. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2099–2122. https://doi.org/10.1145/3729364

[71] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 373–384. https://doi.org/10.1109/icse55347.2025.00211

[72] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[73] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824. https://doi.org/10.1145/3715754

[74] Dapeng Yan, Zhipeng Gao, and Zhiming Liu. 2023. A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1887–1898. https://doi.org/10.1109/ase56229.2023.00096

[75] Dapeng Yan, Kui Liu, Yuqing Niu, Li Li, Zhe Liu, Zhiming Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Crex: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics. *Information and Software Technology* 152 (Dec. 2022), 107043. https://doi.org/10.1016/j.infsof.2022.107043

[76] Dapeng Yan, Wenjie Yang, Zhipeng Gao, Kui Liu, Zhikuang Cai, Xiaoyuan Xie, and Zhiming Liu. 2026. Evolving Trends in Cleanliness of Open Source Projects. *ACM Trans. Softw. Eng. Methodol.* (March 2026). https://doi.org/10.1145/3800680

Just Accepted.
[77] Chau Chin Yiu. 2023. Checkstyle for Legacy Applications [J]. *Itestra De* (2023).
[78] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2618–2640. https://doi.org/10.1145/3729386
[79] 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)*. IEEE, 334–344. https://doi.org/10.1109/msr.2017.2
[80] Zejun Zhang, Zhenchang Xing, Xiaoxue Ren, Qinghua Lu, and Xiwei Xu. 2024. Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1107–1128. https://doi.org/10.1145/3643776
[81] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. 2026. Retrieval-Augmented Generation for AI-Generated Content: A Survey. *Data Science and Engineering* 11, 1 (Jan. 2026), 1–29. https://doi.org/10.1007/s41019-025-00335-5

Received 2025-09-12; accepted 2025-12-22