Evaluating the Capability of LLMs in Identifying Compilation Errors in Configurable Systems

Lucas Albuquerque
Federal University of Campina Grande
Brazil
lucas.albuquerque@ccc.ufcg.edu.br

Rohit Gheyi
Federal University of Campina Grande
Brazil
rohit@dsc.ufcg.edu.br

Márcio Ribeiro
Federal University of Alagoas
Brazil
marcio@ic.ufal.br

ABSTRACT

Compilation is an important process in developing configurable systems, such as Linux. However, identifying compilation errors in configurable systems is not straightforward because traditional compilers are not variability-aware. Previous approaches that detect some of these compilation errors often rely on advanced techniques that require significant effort from programmers. This study evaluates the efficacy of Large Language Models (LLMs), specifically ChatGPT4, Le Chat Mistral and Gemini Advanced 1.5, in identifying compilation errors in configurable systems. Initially, we evaluate 50 small products in C++, Java, and C languages, followed by 30 small configurable systems in C, covering 17 different types of compilation errors. ChatGPT4 successfully identified most compilation errors in individual products and in configurable systems, while Le Chat Mistral and Gemini Advanced 1.5 detected some of them. LLMs have shown potential in assisting developers in identifying compilation errors in configurable systems.

KEYWORDS

LLMs, Compilation Errors, Configurable Systems.

1 INTRODUCTION

Compilation is an important process for creating functional and efficient programs. This challenge is amplified in configurable systems, as seen with the Linux kernel, where variability and the combination of different modules and features can result in an exponential explosion of possible configurations. In such environments, finding bugs that occur only under specific configurations becomes a particularly costly and labor-intensive task. Developing configurable systems with dozens of macros is not easy [17], especially when annotations are not disciplined [15, 13], potentially affecting code quality [5]. Traditional compilers can only check one configuration at a time. Variability-aware parsers are advanced parsing tools designed to handle software systems with multiple configurations and variability. Using the current variability-aware parsers [12, 9] is time consuming, require some effort to setup and do not detect all errors.

Large Language Models (LLMs) have proven to be valuable tools in software generation and review, assisting with code writing and documentation [10, 24]. Some studies are investigating the extent to which LLMs can assist in testing activities [25] and software engineering [11]. However, to the best of our knowledge, no study has yet explored the extent to which LLMs can aid in detecting variability-aware compilation errors.

In this paper, we evaluate the capability of Large Language Models (LLMs) in identifying compilation errors across diverse programming contexts. Our focus is to analyze the performance of three specific LLMs, ChatGPT4, Le Chat Mistral and Gemini Advanced 1.5, in identifying compilation errors. These LLMs were chosen because they are state-of-the-art models in the field, representing the latest advancements in large language model technology. Initially, we assessed the ability of these LLMs to identify issues in a set of 50 small programs across C++, Java, and C. Later, we expanded our analysis to include applying these models to 30 small configurable systems, ranging from 1 to 5 macros in C, with up to 33 LOC, examining 17 different types of compilation errors. All experimental data are available online [3]. In summary, our main contribution is the following:

- Evaluate to what extent ChatGPT4, Le Chat Mistral and Gemini Advanced 1.5 detect compilation errors in programs and configurable systems (Sections 2 and 3).

2 EVALUATION: PRODUCTS

First we assess LLM's performance in compiling single products.

2.1 Methodology

2.1.1 GQM. We structured our evaluation using the Goal-Question-Metric (GQM) approach [4]. The objective is to assess the effectiveness of LLMs, specifically ChatGPT4 and Le Chat Mistral, in identifying compilation errors from the developers' perspective in the context of individual products. We address the following research questions (RQs) to achieve the goal:

- **RQ$_1$** To what extent can ChatGPT4 detect compilation errors in individual products?
- **RQ$_2$** To what extent can Le Chat Mistral detect compilation errors in individual products?
- **RQ$_3$** To what extent can Gemini Advanced 1.5 detect compilation errors in individual products?

Each LLM's response will be compared to the language compiler to determine the number of correct and incorrect identifications.

2.1.2 Planning. The study's planning involves a structured methodology to assess the capabilities of the selected LLMs. The plan is as follows. The study uses a sample of 50 products selected in April 2024 to ensure the results' relevance and timeliness. These products are divided between self-developed creations and code samples extracted from the Codeforces platform, distributed across C++, Java, and C, ranging from 7 to 70 lines of code (median: 25.06 LOC, mean: 24 LOC). The platform Codeforces was chosen for extracting code samples because it offers a wide variety of coding problems and solutions in multiple programming languages, ensuring diversity and real-world relevance. The code samples were chosen at random to mitigate selection bias and provide a realistic

assessment of the LLMs' capabilities in handling typical programming errors.

The code snippets include (nested) loops, (nested) conditionals, functions, data structures (such as maps, arrays, and vectors), input and output operations, and mathematical calculations. Each product contains exactly one type of compilation error, which may include one to three instances of the same error. This selection aims to provide a comprehensive and representative analysis of LLM capabilities across different programming contexts.

The prompt used is "Does the following `language` code compile? `code`," where `language` specifies the programming language (C++, Java, or C) and `code` is the specific code snippet being analyzed. This prompt formulation was chosen for its simplicity, enabling a direct and focused interaction with the LLMs, specifically assessing their ability to determine whether the provided code compiles. We used default parameters. After receiving the LLMs' responses, each product is compiled using the appropriate compiler (GNU GCC 11 for C, GNU G++ 13 for C++, and Java 21 for Java). This step serves to validate the LLM responses against the compiler's verdict, which acts as a baseline for evaluation.

The responses provided by the LLMs are analyzed based on five main criteria, where each response is classified as "Yes," "No," or "Partially." "Yes" indicates success, "No" denotes failure, and "Partially" (◓) is used for detailed discussion in cases where success is not fully achieved but is considered a failure for final evaluation. The criteria are detailed as follows:

- **Detect.** Determines if the LLM identified the presence of a compilation error.
- **Fix.** Evaluates if the LLM proposed an appropriate fix for the compilation error. The LLM should provide corrected code or directly and clearly describe a solution without changing the code's original purpose.
- **Explanation.** Assesses if the LLM satisfactorily explains the problem. Success is only considered if all sub-criteria are marked "Yes," which includes:
  - **Code Element.** Checks if the LLM pinpointed the specific code element causing the error.
  - **Type of Error.** Determines if the LLM accurately classified the type of error.
  - **Location.** Confirms if the LLM correctly indicated the error's location in the code. The LLM must specify in which function the error occurs, or, in the case of variables, identify the specific variable where the error happens.

In April 2024, we analyzed ChatGPT4 and Le Chat Mistral. In May 2024, we also evaluated Gemini Advanced 1.5.

2.2 Results

The performance results of ChatGPT4, Le Chat Mistral, and Gemini Advanced 1.5 are summarized in Table 1. ChatGPT4 exhibited a good performance in detecting and correcting errors, achieving 41 detections and 44 corrections out of a possible 50. In terms of explanation, this model was effective in 31 out of the 41 errors it detected. Le Chat Mistral, on the other hand, detected 28 errors and corrected 32 out of 50 products. Le Chat Mistral explained 23 of the 28 errors it detected. It has a less consistent level compared to ChatGPT4. Gemini Advanced 1.5 detected 27 errors, corrected 35 out of 50 products, and explained 21 of the 27 errors it detected. Among the three models, Gemini Advanced 1.5 was the least effective in error detection, although it showed a higher number of corrections than Le Chat Mistral.

2.3 Discussion

2.3.1 Compilation Error Detection. In six examples, none of the three LLMs can detect the compilation errors. However, in four of these cases, at least one LLM suggests a code improvement that resolves the compilation error. When analyzing the specific types of compilation errors detected by the LLMs, we observe varying performance across different error categories. Notably, syntax errors like "Missing semicolon" and "Mismatching parentheses" exhibited high detection rates by all three LLMs, with ChatGPT4 identifying all 8 cases of "Missing semicolon" and Le Chat Mistral detecting 7 of them. Detection of "Mismatching brackets" was particularly strong in ChatGPT4, detecting 6 out of 6 cases, while Le Chat Mistral and Gemini Advanced 1.5 detected none. In contrast, semantic errors like "Variable not declared" and "Type mismatch" proved more challenging, with both LLMs showing moderate results. ChatGPT4, Le Chat Mistral, and Gemini Advanced 1.5 each detected 3 out of 4 "Variable not declared" cases.

One specific error analyzed during the evaluation was "variable out of scope," where ChatGPT4 correctly identified 7 out of 10 instances. LLMs can identify variables used outside their permissible scope. However, the two instances where errors were not detected involved a common scenario in C++ programming: the declaration of a variable within a for loop header and attempting to access this variable immediately after the loop ends (Id 14 from Table 1). LLMs have 70% success rate in detecting out-of-scope variables. However, the difficulty in identifying errors involving scopes limited to specific blocks, such as those introduced by loops, suggests an opportunity for improvement.

2.3.2 Compilation Error Fixing. In some cases where LLMs did not explicitly detect an error, they still suggested changes to the code. Interestingly, these proposed changes, although not initially aimed at fixing a specific identified issue, ended up resolving the problem. This led to a number of effective corrections exceeding the detected compilation errors. For instance, ChatGPT4 did not initially identify an error in the for element (the missing closing parenthesis) in Id 3: `for (int i = 0; i < (int)a.size(); i++`. But, it suggests to use a range-based `for` loop to simplify the code: `for (int num : a)`. So, it fixed the compilation error. On the other hand, Le Chat Mistral and Gemini Advanced 1.5 not only detect the compilation error:

> *"... There is a missing right parenthesis ')' in the for loop declaration ..."*

but also provide a fix.

2.3.3 Explanation. While hallucinations, or the generation of incorrect and fictitious information by LLMs, are a known issue [26], the results of this evaluation show that the models often provide coherent and useful explanations, even in cases where the initial detection may seem uncertain. In the results presented, we observed that in some instances, LLMs initially indicate no compilation errors, but as the response develops, they recognize the presence of

Table 1: Evaluation results of identifying compilation errors in products.

					ChatGPT4			Explanation		Le Chat Mistral			Explanation		Gemini Advanced 1.5			Explanation	
id	Language	LOC	Type	Error	Detect	Fix	Code	Type	Location	Detect	Fix	Code	Type	Location	Detect	Fix	Code	Type	Location
1	C++	7	Syntax	Missing semicolon	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2	C++	11	Syntax	Missing semicolon (2x)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	◓
3	C++	8	Syntax	Mismatching parentheses	×	✓	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
4	C++	8	Syntax	Mismatching parentheses	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
5	C++	11	Syntax	Mismatching parentheses	✓	✓	✓	✓	✓	×	×	×	×	×	✓	✓	✓	✓	✓
6	C++	11	Syntax	Mismatching parentheses	✓	✓	✓	✓	✓	×	✓	×	×	×	✓	✓	✓	✓	✓
7	C++	10	Syntax	Mismatching brackets	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
8	C++	11	Syntax	Mismatching brackets	✓	✓	✓	✓	✓	◓	✓	×	×	✓	×	✓	×	×	×
9	C++	35	Syntax	Mismatching parentheses	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
10	C++	10	Syntax	Mismatching quotes	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	✓	✓
11	C++	10	Syntax	Invalid variable name (3x)	✓	✓	✓	✓	◓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
12	C++	7	Semantic	Variable not declared (2x)	◓	✓	✓	✓	◓	◓	✓	✓	✓	◓	✓	✓	✓	✓	✓
13	C++	15	Semantic	Type mismatch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
14	C++	16	Semantic	Variable out of scope	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	×
15	C++	12	Semantic	Variable out of scope	✓	✓	✓	×	✓	×	×	×	×	×	×	✓	×	×	×
16	C++	13	Semantic	Variable out of scope	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	×
17	C++	15	Semantic	Variable out of scope	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
18	C++	41	Semantic	Variable out of scope	✓	✓	✓	✓	◓	✓	✓	✓	✓	✓	×	×	×	×	×
19	C++	31	Semantic	Variable out of scope	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
20	C++	26	Semantic	Variable not declared	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	×	×	×	×	×
21	C++	55	Semantic	Function signature mismatch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
22	C++	23	Semantic	Variable out of scope	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
23	C++	55	Semantic	Variable out of scope (3x)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
24	C++	48	Syntax	Missing semicolon (2x)	✓	✓	✓	◓	✓	×	×	×	×	×	×	×	×	×	×
25	C++	28	Semantic	Variable out of scope	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×	×
26	C++	69	Semantic	Variable redefinition	✓	✓	✓	✓	✓	×	×	✓	×	×	✓	✓	✓	×	✓
27	C++	39	Syntax	Missing semicolon	✓	✓	✓	◓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
28	C++	23	Syntax	Missing semicolon	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓
29	C++	29	Syntax	Missing semicolon	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
30	C++	18	Syntax	Invalid macro usage	✓	✓	✓	✓	✓	×	×	×	×	×	✓	✓	✓	×	✓
31	Java	24	Semantic	Dereference of primitive type	✓	✓	✓	◓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
32	Java	33	Syntax	Missing semicolon	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×
33	Java	35	Syntax	Missing semicolon	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	×	✓	×	×	×
34	Java	27	Syntax	Illegal character	✓	✓	✓	×	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
35	Java	31	Semantic	Variable not declared	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
36	Java	29	Syntax	Mismatching parentheses	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
37	Java	32	Syntax	Mismatching brackets	×	×	×	×	×	×	✓	×	×	×	×	×	×	×	×
38	Java	30	Semantic	Type mismatch	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	×
39	Java	21	Syntax	Mismatching brackets	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
40	Java	55	Semantic	Type mismatch (2x)	◓	✓	✓	✓	◓	◓	✓	×	✓	×	✓	✓	✓	✓	✓
41	C	24	Syntax	Missing operand	✓	✓	✓	×	✓	×	×	✓	×	×	×	✓	×	×	×
42	C	25	Syntax	Mismatching brackets	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
43	C	25	Semantic	Type mismatch	✓	✓	✓	✓	✓	✓	✓	×	×	×	✓	✓	✓	✓	✓
44	C	23	Semantic	Variable out of scope	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
45	C	26	Syntax	Invalid return type	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
46	C	15	Semantic	Function not defined	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
47	C	18	Syntax	Mismatching quotes	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓
48	C	18	Syntax	Mismatching parentheses	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
49	C	38	Semantic	Variable not declared	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓
50	C	29	Semantic	Operator not defined	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×

issues, adjusting their initial conclusions. Although this change in stance might seem inconsistent, it rarely compromises the quality of the explanations provided. The final responses, which include corrections to the model's initial assessment, typically offer detailed explanations of the nature and context of the detected error. According to our "Explanation" evaluation metrics, the results were considered satisfactory, with all three LLMs successfully meeting the explanation criteria more than 75% of the time.

2.4 Threats to Validity

A factor that can compromise result validity is selection bias in code samples, as choosing examples that don't adequately represent the diversity of real-world errors could skew the evaluation of LLMs. We created some examples based on compilation errors found in real systems. The modifications made and the new examples created help to minimize the risk of data leakage when using LLMs [23].

3 EVALUATION: CONFIGURABLE SYSTEMS

Next, we evaluate configurable systems.

3.1 Methodology

3.1.1 GQM. The objective is to assess the effectiveness of LLMs, specifically ChatGPT4 and Le Chat Mistral, in identifying compilation errors from the developers' perspective in the context of configurable systems. We address the following RQs:

- **RQ$_1$** To what extent can ChatGPT4 detect compilation errors in configurable systems?
- **RQ$_2$** To what extent can Le Chat Mistral detect compilation errors in configurable systems?
- **RQ$_3$** To what extent can Gemini Advanced 1.5 detect compilation errors in configurable systems?

Each LLM's response will be compared to the language compiler for each product within the configurable system to accurately determine the number of correct and incorrect identifications.

3.1.2 Planning. The study's planning involves a structured methodology to assess the capabilities of the selected LLMs. We included 30 configurable systems, ranging from 4 to 33 LOC (median: 16.8 LOC, mean: 16 LOC). Each configurable system contains 1 to 5 macros and contains one or two types of compilation errors. The code snippets include loops, conditionals, functions, data structures (such as maps, arrays, and vectors), input and output operations, and mathematical calculations. Additionally, they feature nested ifdefs, ifdefs with simple boolean expressions, both disciplined and undisciplined ifdefs [13], ifdefs inside functions, and ifdefs within function declarations. We created 14 configurable systems. Additionally, there are six systems that are based on Braz et al.'s studies that identified compilation errors in configurable systems [6, 7]. The remaining systems are adapted from Abal et al.'s research on variability bugs in the Linux kernel, providing simplified versions of the original code [1, 2].

We used the prompt "Does the following C code compile? `code`," where `code` represents the code snippet. This prompt was chosen for simplicity, focusing on direct interaction with the LLMs to evaluate their ability to comprehend and process conditional compilation. English was used because LLMs are trained on a significantly larger volume of data in this language. Each configurable system is compiled using the GNU GCC 11 compiler for C. During this process, we manually analyzed how many unique products could be generated by activating different features. Each unique product was manually compiled to verify how many configurations contained compilation errors. The 30 configurable systems generated a total of 103 unique products, of which 40 contained compilation errors.

The analysis of the LLM responses follows the same structure used in the evaluation of individual products (Section 2.1.2), with specific adaptations for the configurable systems context:

- **Detect.** It refers to the number of distinct products with compilation errors that the LLMs successfully identified.
- **Fix.** To classify a correction as "Yes," the LLM's proposed solution must be general and applicable to all products, without relying on specific adjustments like directly defining macros in the code that only guarantee compilation in that particular configuration. This criterion seeks to evaluate the model's ability to propose sustainable and generalizable fixes that maintain product functionality without specific manual interventions.
- **Explanation.** The explanation evaluation follows the same approach as the individual product analysis, considering whether the LLM can satisfactorily clarify the detected problem. This includes correctly identifying the code element causing the compilation error, the nature of the error, and the specific location of the issue within the code of a product.

In April 2024, we analyzed ChatGPT4 and Le Chat Mistral. In May 2024, we evaluated Gemini Advanced 1.5. We utilized the default parameters.

3.2 Results

The results of the configurable systems evaluation using ChatGPT4, Gemini Advanced 1.5 and Le Chat Mistral are presented in Table 2. ChatGPT4 detects all compilation errors (CE) in 28 of the 30 tested configurable systems. This model also identified errors in 38 of the 40 individual configurations derived from these lines, missing only "Type Mismatch" and "Variable not declared." In Id 19, ChatGPT4 detected errors in only one of the two erroneous configurations. In terms of fixes, ChatGPT4 proposed effective fixes for 12 of the 30 configurable systems. Regarding explanations, the model was able to provide adequate explanations for 26 of the 29 configurable systems with compilation errors.

Le Chat Mistral, meanwhile, identified all compilation errors in 24 of the 30 configurable systems, and 31 of the 40 individual configurations. The model managed to propose fixes for 9 of the 30 configurable systems and provided adequate explanations for 18 of the 26 detected lines. In Ids 2 and 21, Le Chat Mistral detected compilation errors in some configurations but not all. These results indicate that although it is effective at detecting some compilation errors, Le Chat Mistral faces more challenges in proposing effective fixes and providing detailed explanations.

Gemini Advanced 1.5, on the other hand, can detect all compilation errors in 16 configurable systems. However, it incorrectly states that 10 configurable systems do not have compilation errors. In four configurable systems, it detects some of the compilation errors. In our study, the Gemini performance is worse than ChatGPT4 and Le Chat Mistral.

3.3 Discussion

3.3.1 Compilation Error Detection. Most undetected errors by LLMs are semantic: both undetected errors by ChatGPT4 and eight out of nine undetected errors by Le Chat Mistral are semantic, pointing to a potential area for enhancement. We present a configurable system (Id 10 from Table 2) in Listing 1. In this example, the `norm` function adapts its calculations depending on whether the macros `A` and `B` are defined. When macro `A` is not defined, the `norm` function is configured to accept only two parameters, conflicting with the call made in the main function, where `norm` is invoked with three arguments.

```
#include <stdio.h>
struct point { int x, y;};
int norm(
            int x,
#ifdef A
            int y,
#endif
            int z
            ) {
      int w = x * z;
#ifdef B
      w += y;
#endif
      return w;
}
int main() {
      int x = norm(1, 2, 3);
      printf("%d\n", x);
      return 0;
}
```

Listing 1: Le Chat Mistral and Gemini Advanced 1.5 do not detect a compilation error in Id 10.

Table 2: Evaluation results of identifying compilation errors in configurable systems.

								ChatGPT4						Le Chat Mistral						Gemini Advanced 1.5					
										Explanation					Explanation					Explanation					
id	Lang.	Macros	#Prod.	LOC	Type	Error	#CE	Detect	Fix	Code	Type	Loc.	Detect	Fix	Code	Type	Loc.	Detect	Fix	Code	Type	Loc.
1	C	2	4	9	Semantic	Type not declared	1	1	✓	✓	✓	✓	0	×	×	×	×	0	×	×	×	×
2	C	2	4	16	Semantic	Func. not def.; Func. redef.	2	2	◓	✓	✓	✓	1	×	✓	◓	✓	2	×	✓	✓	✓
3	C	1	2	10	Syntax	Mismatching quotes	1	1	✓	✓	✓	✓	1	✓	✓	✓	✓	1	✓	✓	✓	✓
4	C	1	2	11	Semantic	Function not defined	1	1	✓	✓	✓	✓	1	✓	✓	✓	✓	1	✓	✓	✓	✓
5	C	1	2	8	Semantic	Variable not declared	1	1	✓	✓	✓	✓	1	×	✓	✓	✓	1	✓	✓	✓	✓
6	C	1	2	13	Syntax	Mismatching brackets	1	1	✓	✓	✓	✓	1	×	×	×	✓	0	×	×	×	×
7	C	1	2	9	Syntax	Missing semicolon	1	1	✓	✓	✓	✓	1	✓	✓	✓	✓	0	×	×	×	×
8	C	2	4	13	Semantic	Type not declared	1	1	×	✓	✓	×	0	×	×	×	×	0	×	×	×	×
9	C	2	4	16	Semantic	Variable not declared	1	1	×	✓	✓	✓	1	×	✓	✓	✓	1	✓	✓	✓	✓
10	C	2	4	22	Semantic	Func. sig. mism. (2x)	2	2	×	✓	✓	✓	0	×	×	×	×	1	✓	◓	◓	◓
11	C	2	4	23	Semantic	Func. sig. mism. (2x); Var. not decl.	3	3	×	✓	✓	✓	0	✓	×	×	×	1	×	◓	◓	◓
12	C	1	2	17	Semantic	Label not declared	1	1	✓	✓	✓	✓	1	×	×	×	×	0	×	×	×	×
13	C	2	4	11	Semantic	Type not declared	1	1	×	✓	✓	✓	1	×	✓	×	✓	1	×	✓	×	✓
14	C	1	2	5	Syntax	Missing struct keyword	1	1	×	✓	×	✓	1	✓	✓	✓	✓	1	✓	✓	×	✓
15	C	1	2	10	Semantic	Label not declared	1	1	×	✓	✓	✓	1	✓	✓	✓	✓	0	×	×	×	×
16	C	1	2	4	Syntax	Invalid return type	1	1	✓	✓	✓	✓	1	◓	✓	✓	✓	1	✓	✓	✓	✓
17	C	1	2	14	Semantic	Cont. statement outs. a loop	1	1	✓	✓	✓	✓	1	×	✓	✓	✓	1	✓	✓	✓	✓
18	C	1	2	12	Semantic	Address of bitfield requested	1	1	×	✓	✓	✓	1	×	×	×	×	1	✓	✓	✓	✓
19	C	3	5	32	Semantic	Fund. not def.; Var. not decl.	2	1	×	◓	◓	◓	2	×	✓	✓	✓	1	×	◓	◓	◓
20	C	3	5	27	Semantic	Variable not declared	1	1	×	✓	✓	✓	1	×	×	×	×	1	✓	✓	×	✓
21	C	2	4	25	Syntax	Invalid return type (3x)	3	3	×	✓	✓	✓	2	×	×	×	×	2	◓	✓	◓	✓
22	C	2	4	20	Semantic	Function not defined	1	1	✓	✓	✓	✓	1	✓	✓	✓	✓	1	×	✓	✓	✓
23	C	2	4	19	Semantic	Function not defined	1	1	✓	✓	✓	✓	1	×	✓	✓	✓	1	✓	✓	✓	✓
24	C	2	4	19	Semantic	Function not defined	1	1	×	✓	✓	✓	1	✓	✓	✓	✓	1	✓	✓	✓	✓
25	C	5	8	33	Semantic	Variable not declared (3x)	3	3	×	✓	✓	✓	3	×	✓	✓	✓	3	✓	✓	✓	✓
26	C	4	8	26	Semantic	Function not defined (2x)	2	2	×	✓	✓	✓	2	×	×	×	×	0	×	×	×	×
27	C	2	3	26	Semantic	Type mismatch	1	0	×	×	×	×	1	×	✓	✓	✓	0	×	×	×	×
28	C	1	2	11	Semantic	Function signature mismatch	1	1	×	✓	✓	✓	1	✓	✓	✓	✓	0	×	×	×	×
29	C	2	4	22	Semantic	Function redefinition	1	1	✓	✓	✓	✓	1	×	✓	✓	✓	1	×	✓	✓	✓
30	C	3	2	21	Semantic	Macro not defined	1	1	×	✓	✓	✓	1	×	✓	✓	✓	0	×	×	×	×

This inconsistency should lead to a compilation error, but both Le Chat Mistral and Gemini Advanced 1.5, while analyzing possible products, identifies this only as a warning rather than a compilation error, failing to recognize the problem:

> *"... Neither `A` nor `B` defined: The code compiles without errors. The `norm` function has arguments `x` and `z`, and it only calculates `w = x * z`. The `main` function calls `norm(1, 2, 3)`, passing the extra argument `2`, but it's ignored by the function ..."*

Gemini Advanced 1.5 does point out another problem (Listing 1):

> *"... Only `B` defined: The code won't compile. There will be an error because `y` is used in the `norm` function but not declared as an argument ..."*

This seems to indicate a lack of understanding of the C language by both Le Chat Mistral and Gemini Advanced 1.5. On the other hand, ChatGPT4 detects both problems.

Furthermore, Listing 2 shows an example containing a missing semicolon (Id 7). ChatGPT4 and Le Chat Mistral can detect and explain how to fix it. Gemini does not detect the compilation error:

> *"... if you were to define `A` then the subtraction would be performed ..."*

```
#include <stdio.h>
int main() {
    int a = 3, b = 7;
    int x = a * b;
#ifdef A
    x -= 1
#endif
    printf("%d\n", x);
}
```

Listing 2: Gemini Advanced 1.5 does not detect a compilation error in Id 7.

Le Chat Mistral can detect the compilation errors not detected by ChatGPT4 in two configurable systems containing nested ifdefs (Ids 19 and 27). There are other examples with nested ifdefs (Ids 20 and 21). ChatGPT4 cannot detect a type mismatch in Id 27 when one macro is enabled and another one is disabled. However, the latest ChatGPT4-o detects and explains in details the compilation error. Le Chat Mistral correctly explains why the compilation error happens. In Id 19, we have 3 macros, and ChatGPT4 yields the following message:

> *"... If `B` is defined but `C` is not, the variable `x` is defined and updated in the loop, but `y` remains undeclared and unused ..."*

It correctly indicates that `y` remains undeclared but ChatGPT4 cannot identify that `y` is used in `x += sum + y;`. ChatGPT4-o and Gemini Advanced 1.5 also do not detect it. On the other hand, Le Chat Mistral correctly identifies the compilation error:

> *"... the variable `y` will not be declared, but there is an attempt to use `y` in the expression `x += sum + y;`, which will result in a compilation error ..."*

3.3.2 Compilation Error Fixing. During the evaluation of configurable systems, LLMs suggested fewer fixes compared to individual products (Section 2). Despite this decline, the results are still promising, indicating that even in more complex scenarios, LLMs can identify and suggest valid interventions, albeit less frequently. This suggests that although the models have a reasonable understanding of errors, they could benefit from more specific guidance when proposing solutions. This leads us to consider improvements

for future work, where the prompt used to interact with the LLMs could be adjusted to explicitly request a correction. Changing the prompt to explicitly request a solution could help guide the models not only to identify the issue but also to focus more directly on generating an applicable fix.

3.3.3 Explanation. ChatGPT4 demonstrated a good performance in explaining errors, providing consistent and clear details that aid in understanding the issues detected. This model was effective in clarifying the contexts of compilation errors and their implications, especially for complex cases like "Function not defined" and "Type not declared," where the explanations were detailed and informative. Additionally, ChatGPT4 responses were longer and more comprehensive than those of Le Chat Mistral.

Both Le Chat Mistral and Gemini Advanced 1.5 exhibited limitations in crafting detailed explanations. The model were less consistent, especially in cases requiring a deeper understanding of the interactions between multiple macros and their impact on the code's logic. The discrepancy between compilation error detection and the quality of explanations was more pronounced, indicating significant room for improvement in the accuracy and depth of responses. All three LLMs struggled to explain errors involving configurable systems that generate 4^+ products, often failing to provide explanations that fully captured the nature and cause of the issues. This challenge suggests that, although useful, the models still require refinement to effectively handle the complexity of configurable systems.

3.4 Threats to Validity

Selection bias in the code samples is a significant concern since examples that don't adequately capture the diversity of errors found in real-world development environments can lead to a skewed evaluation of the LLMs' capabilities. We created some examples based on compilation errors found in real configurable systems. The modifications made and the new examples created help to minimize the risk of data leakage when using LLMs [23]. Additionally, while the compilers used as a baseline are generally reliable, the possibility of them containing bugs cannot be completely ruled out. We manually analyze the compiler results.

Beyond these aspects, a specific limitation of this study was the relatively small size of configurable systems assessed, with the largest containing only 33 LOC. Many configurable systems are simplified versions of more complex codebases, potentially making it easier for the LLMs to detect and correct errors. This simplification might not fully reflect the challenges encountered in more extensive and intricate software scenarios, potentially inflating the models' perceived effectiveness.

4 RELATED WORK

Some variability-aware tools have been previously proposed, such as TypeChef [12] and SuperC [9], for detecting certain syntax and type errors in configurable systems written in C. These tools use advanced techniques to implement non-trivial static analyses to identify compilation errors in real-world configurable systems. Users must configure these tools before use. Our work assesses how well LLMs can perform variation-aware analysis, requiring minimal effort from the user.

Abal et al. [2, 1] identified a number of bugs in configurable C systems and studied their characteristics. Some of these bugs are related to compilation errors and were included in our work. For future work, we aim to explore how well LLMs can identify other issues, such as vulnerabilities in addition to compilation errors, using this set of cataloged examples [2, 21]. For instance, we could evaluate a set of vulnerabilities in configurable systems identified by previous approaches [22, 19].

Medeiros et al. [18] proposed a technique to identify a set of syntax errors in configurable C systems. Later, Medeiros et al. [20] proposed a method to detect undeclared variable usage. Both techniques could identify real-world compilation errors in configurable C systems. In our evaluation, some examples are associated with bugs found in these earlier studies.

Braz et al. [6, 7] proposed a technique to detect compilation errors in configurable C systems by analyzing the impact of changes. They suggested a non-trivial static analysis to identify new compilation errors introduced by changes. This technique successfully identified multiple compilation errors in real systems. Our work adopts a simpler approach by using LLMs to detect compilation errors in configurable systems. For future work, we plan to evaluate not only real systems but also LLMs with larger context windows like Gemini to handle larger examples.

5 CONCLUSION

In this paper, we evaluate the extent to which LLMs such as ChatGPT4 and Le Chat Mistral are capable of identifying compilation errors in configurable systems. ChatGPT4 successfully identified 41 out of 50 possible errors in products and 28 out of errors in 30 configurable systems, demonstrating high effectiveness in detecting compilation errors. On the other hand, Le Chat Mistral identified 28 out of 50 errors in products and 24 out of 30 errors in configurable systems in small examples. Gemini Advanced 1.5 identified errors in 16 out of 30 configurable systems. LLMs have shown potential in assisting developers in identifying compilation errors in configurable systems. Some of them are not detected by variability-aware parsers [12]. The ChatGPT4's explanations help developers to understand and fix them.

Future Work. We plan to evaluate real systems. Additionally, we intend to consider other LLMs such as Claude 3.5 Sonnet, GitHub Copilot, Llama 3, among others. We also aim to evaluate other prompts [14, 8], as well as assess how well LLMs can detect and correct compilation errors more deeply, especially in real configurable systems. We aim to investigate the time required for processing by LLMs and the extent to which they propose fixes. We may face challenges similar to those encountered previously by other techniques analyzing highly configurable systems [16]. We will consider the use of sampling algorithms in these scenarios in the context of LLMs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions. This work was partially supported by CNPq and FAPEAL grants.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering*. ACM, 421–432.

[2] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. 2018. Variability bugs in highly configurable systems: a qualitative analysis. *Transactions on Software Engineering and Methodology*, 26, 3, 10:1–10:34.

[3] Lucas Albuquerque, Rohit Gheyi, and Márcio Ribeiro. 2024. Evaluating the capability of llms in identifying compilation errors in configurable systems (artifacts). https://zenodo.org/records/12773324. (2024).

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. *The Goal Question Metric Approach*, 528–532.

[5] Ira D. Baxter and Michael Mehlich. 2001. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the Eighth Working Conference on Reverse Engineering*. IEEE Computer Society, 281–290.

[6] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, and Leopoldo Teixeira. 2016. A change-centric approach to compile configurable systems with #ifdefs. In *Proceedings of the 15th International Conference on Generative Programming: Concepts & Experiences*, 109–119.

[7] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, Leopoldo Teixeira, and Sabrina Souto. 2018. A change-aware per-file analysis to compile configurable systems with #ifdefs. *Computer Languages, Systems & Structures*, 54, 427–450.

[8] DAIR.AI. 2024. Prompt Engineering Guide. https://www.promptingguide.ai/techniques. (2024).

[9] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 323–334.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

[11] Xinyi Hou et al. 2023. Large Language Models for software engineering: A systematic literature review. DOI: 10.48550/ARXIV.2308.10620.

[12] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 805–824.

[13] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 105–114.

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

[15] Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. 2017. The discipline of preprocessor-based annotations does #ifdef tag n't #endif matter. In *Proceedings of the 25th International Conference on Program Comprehension*, 297–307.

[16] Flávio Medeiros, Christian Kastner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the International Conference on Software Engineering*, 643–654.

[17] Flávio Medeiros, Christian Kastner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The love/hate relationship with the C preprocessor: an interview study. In *Proceedings of the European Conference on Object-Oriented Programming*, 999–1022.

[18] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating preprocessor-based syntax errors. In *Generative Programming: Concepts and Experiences*. ACM, 75–84.

[19] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Larissa Braz, Christian Kästner, Sven Apel, and Kleber Santos. 2020. An empirical study on configuration-related code weaknesses. In *34th Brazilian Symposium on Software Engineering*. ACM, 193–202.

[20] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. 2015. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In *Proceedings of the Generative Programming: Concepts and Experiences* (GPCE), 35–44.

[21] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 50–61.

[22] Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade, Baldoino Fonseca, and Márcio Ribeiro. 2018. A qualitative analysis of variability weaknesses in configurable systems with #ifdefs. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, 51–58.

[23] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the silence: the threats of using llms in software engineering. In *ACM/IEEE 46th International Conference on Software Engineering - New Ideas and Emerging Results*. ACM/IEEE.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, 5998–6008.

[25] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50, 911–936.

[26] Yue Zhang et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. (2023). arXiv: 2309.01219 [cs.CL].