# Investigating LLM Capabilities in the Identification of Compilation Errors in Configurable Systems

**Lucas Albuquerque**[1], **Rohit Gheyi**[1]

[1]Federal University of Campina Grande (UFCG)

`lucas.albuquerque@ccc.ufcg.edu.br, rohit@dsc.ufcg.edu.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, GEMINI ADVANCED 1.5, LE CHAT MISTRAL, *and* LLAMA 3, *in identifying compilation errors in configurable systems. We evaluate them in 30 small configurable systems in C, covering 17 different types of compilation errors.* CHATGPT4 *successfully identified 28 out of 30 compilation errors.* LE CHAT MISTRAL, LLAMA 3 *and* GEMINI ADVANCED 1.5 *detected 24, 20, and 16 errors, respectively. LLMs have shown potential in assisting developers in identifying compilation errors in 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 [Medeiros et al. 2015a], especially when annotations are not disciplined [Malaquias et al. 2017, Liebig et al. 2010], potentially affecting code quality [Baxter and Mehlich 2001]. 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 [Kästner et al. 2011, Gazzillo and Grimm 2012] 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 [Goodfellow et al. 2016, Vaswani et al. 2017]. Some studies are investigating the extent to which LLMs can assist in testing activities [Wang et al. 2024] and software engineering [Hou et al. 2023]. 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 four specific LLMs, CHATGPT4, GEMINI ADVANCED 1.5, LE CHAT MISTRAL, and LLAMA 3, in identifying compilation errors. We assessed the ability of these LLMs to identify issues in a set of 30 small configurable systems, ranging from 1 to 5 macros in C, with up to 33 lines of code (LOC), examining 17 different types of compilation errors. We extend our previous work [Albuquerque et al. 2024] by investigating LLAMA 3 and discussing about TYPECHEF and transformations applied to real systems. CHATGPT4 successfully identified 28 out of 30 compilation errors. LE CHAT MISTRAL, LLAMA 3 and GEMINI ADVANCED 1.5 detected 24, 20, and 16 errors, respectively. Some of the errors identified by CHATGPT4 are not identified by the current state of the art variability-aware parser [Kästner et al. 2011]. LLMs have shown potential in assisting developers in identifying compilation errors in configurable systems. All experimental data are available online [Albuquerque and Gheyi 2024].

## 2. Methodology

### 2.1. GQM

The goal is to assess the effectiveness of LLMs, specifically CHATGPT4, GEMINI ADVANCED 1.5, LE CHAT MISTRAL, and LLAMA 3, 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 GEMINI ADVANCED 1.5 detect compilation errors in configurable systems?

**RQ**$_3$ To what extent can LE CHAT MISTRAL detect compilation errors in configurable systems?

**RQ**$_4$ To what extent can LLAMA 3 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.

### 2.2. 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, input and output operations, and mathematical calculations. Additionally, they feature nested ifdefs, ifdefs with simple boolean expressions, both disciplined and undisciplined ifdefs [Liebig et al. 2010], 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 [Braz et al. 2016, Braz et al. 2018]. 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 [Abal et al. 2014, Abal et al. 2018].

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 LLM responses are analyzed based on the following criteria:

- **Detect.** It refers to the number of distinct products with compilation errors that the LLMs successfully identified.
- **Fix.** Evaluates if the LLM successfully proposed a solution for the compilation error. To classify a fix as correct, 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.

We analyzed CHATGPT4 and LE CHAT MISTRAL in April 2024, GEMINI ADVANCED 1.5 in May 2024, and LLAMA 3 in June 2024.

## 3. Results and Discussion

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 at least one compilation error detected. The results of the configurable systems evaluation are presented in Table 1.

LE CHAT MISTRAL 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. LE CHAT MISTRAL cannot detect the following errors: Type not declared (2), Function redefinition (1), Function signature mismatch (4), Variable not declared (1), and Invalid return type (1).

LLAMA 3 detected errors in 20 of the 30 configurable systems, and 27 of the 40 individual configurations. The model managed to propose fixes for 7 of the 30 configurable systems and provided adequate explanations for 20 of the 24 detected lines. In Ids 2, 19, 21 and 26, LLAMA 3 detected compilation errors in some configurations but not all. LLAMA 3 proposed the fewest number of fixed among the LLMs evaluated in our study. In some answers (Id 16), it asks the user if she would like a fix. The output for Ids 12 and 13 indicate that the configurable system do not compile but the explanation are incorrect.

GEMINI ADVANCED 1.5 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 ADVANCED 1.5 performance is worst among the LLMs considered

**Table 1. Evaluation results of identifying compilation errors in config. systems.**

| | | | | | | GPT4 | | Mistral | | Gemini 1.5 | | LLama 3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Id** | **Mac.** | **#Prod.** | **LOC** | **Error** | **#CE** | **Det.** | **Fix** | **Det.** | **Fix** | **Det.** | **Fix** | **Det.** | **Fix** |
| 1 | 2 | 4 | 9 | Type not declared | 1 | 1 | ✓ | 0 | × | 0 | × | 0 | × |
| 2 | 2 | 4 | 16 | Fun. not def.; Fun. redef. | 2 | 2 | ◓ | 1 | × | 2 | × | 1 | × |
| 3 | 1 | 2 | 10 | Mismatching quotes | 1 | 1 | ✓ | 1 | ✓ | 1 | ✓ | 1 | ✓ |
| 4 | 1 | 2 | 11 | Function not defined | 1 | 1 | ✓ | 1 | ✓ | 1 | ✓ | 1 | × |
| 5 | 1 | 2 | 8 | Variable not declared | 1 | 1 | ✓ | 1 | × | 1 | ✓ | 1 | ✓ |
| 6 | 1 | 2 | 13 | Mismatching brackets | 1 | 1 | ✓ | 1 | × | 0 | × | 0 | × |
| 7 | 1 | 2 | 9 | Missing semicolon | 1 | 1 | ✓ | 1 | ✓ | 0 | × | 1 | ✓ |
| 8 | 2 | 4 | 13 | Type not declared | 1 | 1 | × | 0 | × | 0 | × | 1 | × |
| 9 | 2 | 4 | 16 | Variable not declared | 1 | 1 | × | 1 | × | 1 | ✓ | 1 | × |
| 10 | 2 | 4 | 22 | Fun. sig. mis. (2x) | 2 | 2 | × | 0 | × | 1 | ✓ | 2 | × |
| 11 | 2 | 4 | 23 | Fun. sig. mis. (2x); Var. not dec. | 3 | 3 | × | 0 | ✓ | 1 | × | 3 | × |
| 12 | 1 | 2 | 17 | Label not declared | 1 | 1 | ✓ | 1 | × | 0 | × | 1 | × |
| 13 | 2 | 4 | 11 | Type not declared | 1 | 1 | × | 1 | × | 1 | × | 1 | × |
| 14 | 1 | 2 | 5 | Missing struct keyword | 1 | 1 | × | 1 | ✓ | 1 | ✓ | 1 | × |
| 15 | 1 | 2 | 10 | Label not declared | 1 | 1 | × | 1 | ✓ | 0 | × | 1 | × |
| 16 | 1 | 2 | 4 | Invalid return type | 1 | 1 | ✓ | 1 | ◓ | 1 | ✓ | 1 | × |
| 17 | 1 | 2 | 14 | Cont. stmt out. a loop | 1 | 1 | ✓ | 1 | × | 1 | ✓ | 1 | × |
| 18 | 1 | 2 | 12 | Address of bitfield requested | 1 | 1 | × | 1 | × | 1 | ✓ | 1 | × |
| 19 | 3 | 5 | 32 | Fun. not def.; Var. not dec. | 2 | 1 | × | 2 | × | 1 | × | 1 | × |
| 20 | 3 | 5 | 27 | Variable not declared | 1 | 1 | × | 1 | × | 1 | ✓ | 0 | × |
| 21 | 2 | 4 | 25 | Invalid return type (3x) | 3 | 3 | × | 2 | × | 2 | ◓ | 1 | ◓ |
| 22 | 2 | 4 | 20 | Function not defined | 1 | 1 | ✓ | 1 | ✓ | 1 | × | 1 | ✓ |
| 23 | 2 | 4 | 19 | Function not defined | 1 | 1 | ✓ | 1 | × | 1 | ✓ | 1 | × |
| 24 | 2 | 4 | 19 | Function not defined | 1 | 1 | × | 1 | ✓ | 1 | ✓ | 1 | ✓ |
| 25 | 5 | 8 | 33 | Variable not declared (3x) | 3 | 3 | × | 3 | × | 3 | ✓ | 0 | × |
| 26 | 4 | 8 | 26 | Function not defined (2x) | 2 | 2 | × | 2 | × | 0 | × | 1 | × |
| 27 | 2 | 3 | 26 | Type mismatch | 1 | 0 | × | 1 | × | 0 | × | 0 | × |
| 28 | 1 | 2 | 11 | Function signature mismatch | 1 | 1 | × | 1 | ✓ | 0 | × | 1 | ✓ |
| 29 | 2 | 4 | 22 | Function redefinition | 1 | 1 | ✓ | 1 | × | 1 | × | 1 | ✓ |
| 30 | 3 | 2 | 21 | Macro not defined | 1 | 1 | × | 1 | × | 0 | × | 0 | × |

in our study. Listing 1 shows an example containing a missing semicolon (Id 7). ChatGPT4, Le Chat Mistral and LLama 3 can detect and explain how to fix it. Gemini Advanced 1.5 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 1. Gemini Advanced 1.5 does not detect a compilation error in Id 7.**

## 3.1. Compilation Error Detection

Most undetected errors by LLMs are semantic: both undetected errors by ChatGPT4 and five out of six undetected errors by Le Chat Mistral are semantic. The same scenario happens for Gemini Advanced 1.5 and LLama 3, pointing to a potential area for enhancement. We present a configurable system (Id 10 from Table 1) in Listing 2. 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. This inconsistency should lead to a compilation error, but LE CHAT MISTRAL, while analyzing possible products, identifies this only as a warning rather than a compilation error, failing to recognize the problem. On the other hand, CHATGPT4 and LLAMA 3 detect the compilation error. GEMINI ADVANCED 1.5 correctly identifies one compilation error:

> *"... 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 ..."*

However, it does not recognize the other compilation error assuming that the extra argument will be ignored by the compiler:

> *"... 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 ..."*

```
#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 2. LE CHAT MISTRAL does not detect a compilation error in Id 10.**

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 disable. 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 also does 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 ..."*

## 3.2. Compilation Error Fixing

During the evaluation of configurable systems, LLMs suggested some fixes even when these were not requested in the prompt. We observed that while the LLMs could detect and explain errors, their ability to suggest effective corrections was less consistent. Despite this decline, the results are still promising. They indicate 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. 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.

LLAMA 3 exhibited a performance similar to ChatGPT-4, effectively explaining errors and providing clear details that helped in understanding the issues detected. Its explanations were also detailed and informative, particularly for complex cases. Additionally, CHATGPT4 and LLAMA 3 responses were longer and more comprehensive than the other two models.

LE CHAT MISTRAL and GEMINI ADVANCED 1.5 exhibited limitations in crafting detailed explanations. These two models 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 four 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 software product lines.

## 3.4. TYPECHEF

TYPECHEF [Kästner et al. 2011] is a variability-aware parser. It performs a complex static analysis to identify compilation erros in configurable systems. In our study, we considered some subjects (Ids: 13-18) proposed before [Braz et al. 2018] that have compilation errors, but they cannot be detected by TYPECHEF. All LLMs can detect all compilation errors, but GEMINI ADVANCED 1.5 that cannot the Label not Declared error in Id 15. GEMINI ADVANCED 1.5, CHATGPT4 and LE CHAT MISTRAL provide fixes for 4, 2 and 2 out of 6 examples. LLAMA 3 does not show a fix for any of the six examples. However, GEMINI ADVANCED 1.5 does not provide a good explanation for the six examples. LLAMA 3 provide the best answers. As future work, we intend to perform a more comprehensive study comparing LLMs to TYPECHEF [Kästner et al. 2011] and SUPERC [Gazzillo and Grimm 2012], another variability-aware parser.

## 3.5. Real Configurable Systems

Identifying compilation errors in real configurable systems is a challenging task [Kästner et al. 2011, Gazzillo and Grimm 2012, Braz et al. 2018]. For example, consider the BusyBox commit[1] containing 147 insertions and 28 deletions. The `httpd.c` file has 2,422 LOC. Next we present a code snippet showing the error in Listing 3.

```
... #ifdef ENABLE_AUTH_MD5 && ENABLE_PAM
  struct pam_userinfo {
    const char *name;
    const char *pw;
  };
#endif ...
#ifdef ENABLE_PAM
  struct pam_userinfo userinfo; 9
#endif ...
```

**Listing 3. Some insertions in the `httpd.c` file.**

Since `httpd.c` has 18 macros, generating and compiling all configurations may be costly. Previous approaches [Medeiros et al. 2013, Medeiros et al. 2015b] cannot detect it. TYPECHEF also cannot identify the compilation error. The complex static change-centric per-file analysis proposed by Braz et al. [Braz et al. 2016, Braz et al. 2018] can detect it when `ENABLE_PAM` is enabled and `ENABLE_AUTH_MD5` is disabled.

The current LLM has a context window [Brown et al. 2020] referring to the maximum amount of text that the model can consider at one time to generate or comprehend responses. So, we cannot include the entire source code and analyze them in the current LLMs. Since we cannot analyze the entire code, we passed to CHATGPT4 a file containing the commit insertions and deletions, following a similar approach used by Braz et al. [Braz et al. 2016, Braz et al. 2018]. First, we used the following prompt:

- Does this commit introduce any compilation errors related to variability?

However, it does not detect the compilation error. We tried to modify by adding asking to do a per file analysis and assume that the other files are correct, but it did not detect. Finally, we asked:

- Does this commit introduce any compilation errors related to variability when `ENABLE_AUTH_MD5` is disabled and `ENABLE_PAM` is enabled?

In this case, CHATGPT4 correctly identifies, and provides a good explanation about the compilation error and a fix. CLAUDE 3.5 SONNET also could only detect in the last prompt. As future work, we intend to investigate more other prompts and LLMs to analyze to what extent LLMs can identify compilation errors in real configurable systems. We intend to investigate whether using part of previous techniques combining with LLMs can yield better results.

[1]https://git.busybox.net/busybox/diff/networking/httpd.c?id=7291755

### 3.6. 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 [Sallou et al. 2024]. 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 [Kästner et al. 2011] and SuperC [Gazzillo and Grimm 2012], 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. [Abal et al. 2018, Abal et al. 2014] 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 [Abal et al. 2018, Mordahl et al. 2019]. For instance, we could evaluate a set of vulnerabilities in configurable systems identified by previous approaches [Muniz et al. 2018, Medeiros et al. 2020].

Medeiros et al. [Medeiros et al. 2013] proposed a technique to identify a set of syntax errors in configurable C systems. Later, Medeiros et al. [Medeiros et al. 2015b] 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. [Braz et al. 2016, Braz et al. 2018] 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, GEMINI ADVANCED 1.5, LE CHAT MISTRAL, and LLAMA 3 are capable of identifying compilation errors in products and configurable systems. ChatGPT4 successfully identified 28 out of errors in 30 configurable systems, demonstrating high effectiveness in detecting compilation errors. On the other hand, Le Chat Mistral, Llama 3 and Gemini Advanced 1.5 identified 24, 20 and 16 out of 30 errors in small configurable systems, respectively. LLMs have shown potential in assisting developers in identifying compilation errors in configurable systems. For future work, we plan to evaluate real systems. Additionally, we intend to consider other LLMs such as Claude 3.5 Sonnet, among others. We also aim to evaluate other prompts [Liu et al. 2023, DAIR.AI 2024], as well as assess how well LLMs can detect and correct compilation errors more deeply, especially in real configurable systems. We may face challenges similar to those encountered previously by other techniques analyzing highly configurable systems [Medeiros et al. 2016]. We will consider the use of sampling algorithms in these scenarios in the context of LLMs.

## References

Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 variability bugs in the linux kernel: a qualitative analysis. In *Automated Software Engineering*, pages 421–432. ACM.

Abal, I., Melo, J., Stănciulescu, S., Brabrand, C., Ribeiro, M., and Wasowski, A. (2018). Variability bugs in highly configurable systems: A qualitative analysis. *Transactions on Software Engineering and Methodology*, 26(3):10:1–10:34.

Albuquerque, L. and Gheyi, R. (2024). Investigating LLM capabilities in the identification of compilation errors in configurable systems (artifacts). `https://zenodo.org/records/13230363`.

Albuquerque, L., Gheyi, R., and Ribeiro, M. (2024). Evaluating the capability of LLMs in identifying compilation errors in configurable systems. In *Proceedings of the Brazilian Symposium on Software Engineering (NIER track)*, SBES-NIER.

Baxter, I. D. and Mehlich, M. (2001). Preprocessor conditional removal by simple partial evaluation. In *Working Conference on Reverse Engineering*, pages 281–290. IEEE.

Braz, L., Gheyi, R., Mongiovi, M., Ribeiro, M., Medeiros, F., and Teixeira, L. (2016). A change-centric approach to compile configurable systems with #ifdefs. In *Generative Programming: Concepts & Experiences*, pages 109–119.

Braz, L., Gheyi, R., Mongiovi, M., Ribeiro, M., Medeiros, F., Teixeira, L., and Souto, S. (2018). A change-aware per-file analysis to compile configurable systems with #ifdefs. *Computer Languages, Systems & Structures*, 54:427–450.

Brown, T. B. et al. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*.

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

Gazzillo, P. and Grimm, R. (2012). SuperC: parsing all of C by taming the preprocessor. In *Programming Language Design and Implementation*, pages 323–334. ACM.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Hou, X. et al. (2023). Large Language Models for software engineering: A systematic literature review.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, pages 805–824. ACM.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *International Conference on Software Engineering*, pages 105–114. ACM.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys (CSUR)*, 55(9):1–35.

Malaquias, R., Ribeiro, M., Bonifácio, R., Monteiro, E., Medeiros, F., Garcia, A., and Gheyi, R. (2017). The discipline of preprocessor-based annotations does #ifdef tag n't #endif matter. In *Int. Conference on Program Comprehension*, pages 297–307.

Medeiros, F., Kastner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In *ICSE*, pages 643–654.

Medeiros, F., Kastner, C., Ribeiro, M., Nadi, S., and Gheyi, R. (2015a). The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 999–1022.

Medeiros, F., Ribeiro, M., and Gheyi, R. (2013). Investigating preprocessor-based syntax errors. In *Generative Programming: Concepts and Experiences*, pages 75–84.

Medeiros, F., Ribeiro, M., Gheyi, R., Braz, L., Kästner, C., Apel, S., and Santos, K. (2020). An empirical study on configuration-related code weaknesses. In *34th Brazilian Symposium on Software Engineering*, pages 193–202. ACM.

Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L., and Gheyi, R. (2015b). An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In *Generative Programming: Concepts and Experiences*, pages 35–44.

Mordahl, A., Oh, J., Koc, U., Wei, S., and Gazzillo, P. (2019). An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Foundations of Software Engineering*, pages 50–61. ACM.

Muniz, R., Braz, L., Gheyi, R., Andrade, W., Fonseca, B., and Ribeiro, M. (2018). A qualitative analysis of variability weaknesses in configurable systems with #ifdefs. In *Variability Modelling of Software-Intensive Systems*, pages 51–58.

Sallou, J., Durieux, T., and Panichella, A. (2024). Breaking the silence: the threats of using llms in software engineering. In *ICSE-NIER*. ACM/IEEE.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *NeurIPS*, pages 5998–6008.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *TSE*, 50:911–936.