



**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

LUCAS BRENNER HERCULANO E ALBUQUERQUE

**AVALIANDO A CAPACIDADE DE LLMS NA IDENTIFICAÇÃO
DE ERROS DE COMPILAÇÃO EM LINHAS DE PRODUTO DE
SOFTWARE**

CAMPINA GRANDE - PB

2024

LUCAS BRENNER HERCULANO E ALBUQUERQUE

**AVALIANDO A CAPACIDADE DE LLMS NA IDENTIFICAÇÃO
DE ERROS DE COMPILAÇÃO EM LINHAS DE PRODUTO DE
SOFTWARE**

**Trabalho de Conclusão Curso
apresentado ao Curso Bacharelado em
Ciência da Computação do Centro de
Engenharia Elétrica e Informática da
Universidade Federal de Campina
Grande, como requisito parcial para
obtenção do título de Bacharel em
Ciência da Computação.**

Orientador: Professor Dr. Rohit Gheyi

CAMPINA GRANDE - PB

2024

LUCAS BRENNER HERCULANO E ALBUQUERQUE

**AVALIANDO A CAPACIDADE DE LLMS NA IDENTIFICAÇÃO
DE ERROS DE COMPILAÇÃO EM LINHAS DE PRODUTO DE
SOFTWARE**

**Trabalho de Conclusão Curso
apresentado ao Curso Bacharelado em
Ciência da Computação do Centro de
Engenharia Elétrica e Informática da
Universidade Federal de Campina
Grande, como requisito parcial para
obtenção do título de Bacharel em
Ciência da Computação.**

BANCA EXAMINADORA:

Professor Dr. Rohit Gheyi

Orientador – UASC/CEEI/UFCG

Professor Dr. Franklin de Souza Ramalho

Examinador – UASC/CEEI/UFCG

Professor Dr. Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 15 de MAIO de 2024.

CAMPINA GRANDE - PB

RESUMO

A compilação é um processo essencial no desenvolvimento de linhas de produto de software, como o Linux. Entretanto, identificar erros de compilação em Linhas de Produto de Software (LPS) não é trivial, já que os compiladores tradicionais não são conscientes de variação. Abordagens anteriores foram propostas que identificam alguns desses erros de compilação usando técnicas avançadas que requerem um esforço dos programadores em usarem. Este estudo avalia a eficácia de Modelos de Linguagem de Grande Porte (LLMs), especificamente o ChatGPT 4 e Le Chat Mistral, na identificação de erros de compilação em LPS. Inicialmente foram testados 50 produtos nas linguagens C++, Java e C, e posteriormente 30 LPS em C, abrangendo 17 tipos diferentes de erros de compilação. Os dois LLMs foram avaliados com base na sua capacidade de reconhecer e diagnosticar corretamente os erros. O ChatGPT conseguiu identificar 82% e 95% dos erros de compilação em produtos e LPS, enquanto que o Le Chat Mistral obteve 56% e 78%, respectivamente. A análise revelou que, embora os LLMs possam identificar uma gama de erros de compilação, desafios específicos permanecem, especialmente em ambientes de LPS com alta variabilidade. O estudo sugere a necessidade de refinamentos contínuos nos modelos de LLM para melhorar sua precisão e utilidade em cenários de desenvolvimento de software complexos.

EVALUATING THE CAPABILITY OF LLMs IN IDENTIFYING COMPILATION ERRORS IN SOFTWARE PRODUCT LINES

ABSTRACT

Compilation is an essential process in the development of software product lines, such as Linux. However, identifying compilation errors in Software Product Lines (SPL) is not trivial, as traditional compilers are not variation-aware. Previous approaches have been proposed that identify some of these compilation errors using advanced techniques that require effort from programmers to use. This study evaluates the efficacy of Large Language Models (LLMs), specifically ChatGPT 4 and Le Chat Mistral, in identifying compilation errors in SPLs. Initially, 50 products in C++, Java, and C were tested, followed by 30 SPLs in C, covering 17 different types of compilation errors. The two LLMs were evaluated based on their ability to recognize and correctly diagnose errors. ChatGPT managed to identify 82% and 95% of the compilation errors in products and SPLs, while Le Chat Mistral achieved 56% and 78%, respectively. The analysis revealed that, although LLMs can identify a range of compilation errors, specific challenges remain, especially in SPL environments with high variability. The study suggests the need for continuous refinements in LLM models to improve their accuracy and utility in complex software development scenarios.

Avaliando a Capacidade de LLMs na Identificação de Erros de Compilação em Linhas de Produto de Software

Lucas Albuquerque
Universidade Federal de Campina Grande
Campina Grande, Paraíba
lucas.albuquerque@ccc.ufcg.edu.br

Rohit Gheyi
Universidade Federal de Campina Grande
Campina Grande, Paraíba
rohit@dsc.ufcg.edu.br

RESUMO

A compilação é um processo essencial no desenvolvimento de linhas de produto de software, como o Linux. Entretanto, identificar erros de compilação em Linhas de Produto de Software (LPS) não é trivial, já que os compiladores tradicionais não são conscentes de variação. Abordagens anteriores foram propostas que identificam alguns desses erros de compilação usando técnicas avançadas que requerem um esforço dos programadores em usarem. Este estudo avalia a eficácia de Modelos de Linguagem de Grande Porte (LLMs), especificamente o ChatGPT 4 e Le Chat Mistral, na identificação de erros de compilação em LPS. Inicialmente foram testados 50 produtos nas linguagens C++, Java e C, e posteriormente 30 LPS em C, abrangendo 17 tipos diferentes de erros de compilação. Os dois LLMs foram avaliados com base na sua capacidade de reconhecer e diagnosticar corretamente os erros. O ChatGPT conseguiu identificar 82% e 95% dos erros de compilação em produtos e LPS, enquanto que o Le Chat Mistral obteve 56% e 78%, respectivamente. A análise revelou que, embora os LLMs possam identificar uma gama de erros de compilação, desafios específicos permanecem, especialmente em ambientes de LPS com alta variabilidade. O estudo sugere a necessidade de refinamentos contínuos nos modelos de LLM para melhorar sua precisão e utilidade em cenários de desenvolvimento de software complexos.

Keywords

LLM, Erros de Compilação, Sistemas Configuráveis, LPS.

1. INTRODUÇÃO

A compilação é um processo crítico no desenvolvimento de software, transformando código-fonte escrito em linguagens de programação de alto nível em código de máquina que pode ser executado por um computador. Este processo é essencial para a criação de programas funcionais e eficientes. No entanto, durante a compilação, os erros são comuns e as mensagens de erro fornecidas pelos compiladores podem ser criptográficas e pouco informativas, dificultando a tarefa de depuração para os desenvolvedores.

Modelos de Linguagem de Grande Porte (LLMs) têm se mostrado ferramentas valiosas na geração e revisão de software, apoiando a escrita de código e a documentação [3, 7]. Existem vários trabalhos que estão investigando até que ponto LLMs podem ajudar em atividades de teste [4] e engenharia de software [8]. Contudo, os códigos gerados por esses modelos nem sempre são perfeitos; muitas vezes, eles introduzem erros de compilação que precisam ser identificados e corrigidos. Este desafio é amplificado

em projetos que adotam a abordagem de Linha de Produto de Software (LPS), como é o caso do kernel do Linux, onde a variabilidade e a combinação de diferentes módulos e funcionalidades podem resultar em uma explosão exponencial de configurações possíveis. Em tais ambientes, encontrar bugs que ocorrem apenas sob certas configurações torna-se uma tarefa especialmente custosa e trabalhosa. Não é trivial desenvolver sistemas configuráveis contendo dezenas de macros [20], especialmente quando as anotações não são disciplinadas [23, 24], podendo influenciar negativamente na qualidade do código [25].

O objetivo deste estudo é avaliar a capacidade de Modelos de Linguagem de Grande Porte (LLMs) na identificação de erros de compilação em contextos diversificados de programação. Nosso foco é analisar o desempenho de dois LLMs específicos, ChatGPT 4 e Le Chat Mistral, no reconhecimento de erros de compilação. Inicialmente, avaliamos a capacidade desses LLMs em identificar problemas em um conjunto de 50 programas, distribuídos entre as linguagens C++, Java e C. Posteriormente, expandimos nossa análise para incluir a aplicação desses modelos em 30 LPS, que variam de 1 a 5 macros em C, com até 33 linhas de código, examinando 17 tipos diferentes de erros de compilação. Todos os dados estão disponíveis online [28].

Este artigo está estruturado da seguinte maneira: A Seção 2 apresenta uma fundamentação teórica sobre os conceitos envolvidos; a Seção 3 descreve a metodologia e os resultados obtidos na avaliação dos produtos individuais; a Seção 4 aborda a avaliação das linhas de produto de software, explorando os resultados e discussões pertinentes; e as seções subsequentes discutem trabalhos relacionados e as conclusões do estudo.

2. FUNDAMENTAÇÃO TEÓRICA

Nessa seção apresentamos uma visão geral sobre erros de compilação, LLM, e LPS.

2.1 Erros de Compilação

Erros de compilação são falhas identificadas pelo compilador durante a tradução do código fonte de uma linguagem de programação para código de máquina. Esses erros são geralmente divididos em duas categorias principais: sintáticos e semânticos.

Erros Sintáticos: Resultam da violação das regras gramaticais da linguagem. São erros facilmente detectáveis pelos compiladores, que frequentemente fornecem mensagens específicas sobre a localização e a natureza do erro. Exemplos comuns incluem o uso incorreto de parênteses ou a omissão de um ponto e vírgula.

Erros Semânticos: Ocorrem quando o código, apesar de sintaticamente correto, não faz sentido devido a operações

logicamente inválidas ou inadequadas. Estes são mais complexos para identificar e podem incluir problemas como a utilização de tipos incompatíveis em uma operação ou a chamada incorreta de funções. Erros semânticos podem compilar, mas resultam em comportamentos inesperados ou falhas durante a execução.

Ambos os tipos de erro precisam ser corrigidos para que o software funcione corretamente, sendo críticos para o processo de desenvolvimento e manutenção de software.

2.2 LLMs

Modelos de Linguagem de Grande Porte (LLMs) são sistemas avançados de inteligência artificial que aprendem a partir da análise de vastos conjuntos de texto [27]. Essa imensa base de dados permite que eles compreendam e manipulem a linguagem natural de forma eficaz. LLMs podem executar uma série de tarefas, incluindo a tradução de textos, a geração de respostas contextualizadas e o resumo de documentos extensos.

Esses modelos representam um progresso notável no campo da inteligência artificial, com implicações profundas para a interação humano-computador. Construídos com redes neurais profundas que aprendem a partir de milhões de frases, os LLMs não apenas dominam tarefas de processamento de texto, mas também são aplicáveis em engenharia de software. Eles também auxiliam programadores na revisão de código, detecção de erros de compilação e na sugestão de correções, melhorando a eficiência e reduzindo o tempo de desenvolvimento.

2.3 Linhas de Produto de Software

Uma Linha de Produto de Software (LPS) é uma abordagem de desenvolvimento que permite a criação de uma família de produtos de software com características comuns, mas adaptadas a diferentes necessidades do mercado [26]. As LPS são valorizadas por sua capacidade de aumentar a eficiência e flexibilidade no desenvolvimento de software. Por simplicidade, neste artigo iremos considerar sistemas configuráveis como LPS.

```
1 #include <stdio.h>
2
3 int val;
4
5 #ifdef A
6 int f(int x) {
7     return x * 3 + val++;
8 }
9 #else
10 int f() {
11     return val++;
12 }
13 #endif
14
15 int main() {
16     int foo = 3;
17 #ifdef B
18     foo += f();
19 #endif
20     printf("%d\n", foo);
21 }
```

Listagem 1. Exemplo de uma linha de produto com duas *macros*.

Um componente técnico crucial nas LPS são as diretivas de pré-processador, como *#ifdef*, e as *macros*. Diretivas como *#ifdef*

permitem a inclusão ou exclusão condicional de partes do código, promovendo a modularidade e adaptando o software a diferentes configurações ou plataformas. Enquanto isso, as *macros*, que são instruções para substituição de texto ou comando processadas pelo pré-processador, simplificam o desenvolvimento ao reduzir repetições e facilitar mudanças globais (ver Listagem 1). No entanto, a combinação de diretivas e *macros* pode complicar a manutenção e os testes devido à complexidade das interações entre configurações, ocultando potencialmente erros que só emergem sob condições específicas.

3. AVALIAÇÃO 1: PRODUTO

Antes de avaliarmos a capacidade dos LLMs em detectar erros de compilação em LPS, consideramos inicialmente apenas produtos, conforme descrito nas seções a seguir.

3.1 Metodologia

A seguir apresentamos a metodologia usada para avaliar a detecção de erros de compilação em produtos.

3.1.1 GQM

A seguir estruturamos a nossa avaliação usando GQM [9]. O objetivo é avaliar a eficácia dos LLMs, especificamente ChatGPT 4 e Le Chat Mistral, com o propósito de identificar erros de compilação do ponto de vista dos desenvolvedores no contexto de produtos.

Nós avaliamos as seguintes questões de pesquisa:

QP₁. Até que ponto o ChatGPT 4 consegue detectar erros de compilação em produtos?

QP₂. Até que ponto o Le Chat Mistral consegue detectar erros de compilação em produtos?

Cada resposta dos LLMs será comparada ao compilador da linguagem para contarmos os números de acertos e erros.

3.1.2 Planejamento

O planejamento do estudo envolve uma metodologia estruturada para avaliar as capacidades dos LLMs selecionados. Segue-se o seguinte plano:

Seleção de Produtos: A pesquisa utiliza uma amostra de 50 produtos, que foram testados no mês de abril de 2024, para assegurar a atualidade e relevância dos resultados. Esses produtos são divididos entre criações próprias e códigos extraídos da plataforma Codeforces, com uma distribuição entre as linguagens C++, Java e C, variando entre 7 e 70 linhas de código (mediana: 25,06 LOC e média: 24 LOC). Cada produto possui exatamente um tipo de erro de compilação, podendo conter de 1 a 3 instâncias do mesmo erro. Essa seleção visa garantir uma análise abrangente e representativa das capacidades dos LLMs em diferentes contextos de programação.

Formulação do Prompt: Utilizamos o prompt "Does the following *{language}* code compile? *{code}*", onde *{language}* indica a linguagem do programa (C++, Java, ou C) e *{code}* é o trecho específico de código a ser analisado. Esta formulação do prompt foi escolhida para priorizar a simplicidade, permitindo uma interação direta e focada com os LLMs, concentrando-se especificamente na capacidade dos modelos de avaliarem se o código apresentado compila ou não.

Compilação e Validação: Após obter as respostas dos LLMs, cada produto é compilado usando o compilador apropriado (GNU

GCC 11 para C, GNU G++ 13 para C++, e Java 21 para Java). Essa etapa serve para validar as respostas dos LLMs contra o veredito do compilador, que funciona como um baseline para avaliação.

Análise Qualitativa das Respostas: As respostas fornecidas pelos LLMs são analisadas com base em cinco critérios principais, onde cada resposta é classificada como "Yes", "No", ou "Partially". Consideramos "Yes" como um indicativo de sucesso e "No" como um fracasso. "Partially" é utilizado para fins de discussão detalhada nos casos onde o sucesso não é completamente alcançado, mas, para efeitos de avaliação final, é considerado um fracasso. A seguir, detalhamos cada critério:

- **Detect:** Verifica se o LLM identificou a presença de um erro de compilação.
- **Fix:** Avalia se o LLM propôs uma correção adequada para o erro de compilação.
 - O LLM deve apresentar o código corrigido ou descrever de forma direta e clara uma solução para o problema, sem modificar o propósito original do código.
- **Explanation:** Avalia se o LLM explica o problema de forma satisfatória. Para este critério, o sucesso é considerado apenas se todos os subcritérios são avaliados como "Yes". Conta com os seguintes critérios:
 - 1) **Code Element:** Examina se o LLM conseguiu apontar o elemento específico do código que causou o erro.
 - 2) **Type of Error:** Determina se o LLM classificou corretamente o tipo de erro encontrado.
 - 3) **Location:** Confirma se o LLM indicou corretamente a localização do erro no código.
 - O LLM deve especificar em qual função o erro ocorre ou, em casos de variáveis, em qual variável específica o erro ocorre.

Modelos Avaliados: A análise inclui dois modelos de LLMs, ChatGPT 4 e Le Chat Mistral, permitindo uma comparação direta das capacidades de cada um em contextos similares de teste.

3.2 Resultados

Os resultados obtidos na análise de desempenho dos modelos de LLM, ChatGPT 4 e Le Chat Mistral, são sumarizados nas Tabelas 1 e 2, destacando diferenças significativas em sua capacidade de detectar, corrigir e explicar erros de compilação em uma variedade de cenários de teste.

ChatGPT 4 exibiu uma boa performance na detecção e correção de erros, alcançando 41 detecções e 44 correções em 50 possíveis, conforme apresentado na Tabela 1. Em termos de explicação, este modelo foi eficaz em 31 dos 41 erros que detectou, indicando uma habilidade sólida de fornecer contexto e detalhes sobre as correções.

Le Chat Mistral, por sua vez, detectou 28 erros e corrigiu 32 dos 50 produtos, com os detalhes também apresentados na Tabela 1. O Mistral explicou 23 dos 28 erros detectados, mostrando competência, embora em um nível menos consistente em comparação ao ChatGPT 4.

Em erros sintáticos como "*Missing semicolon*" e "*Mismatching quotes*", ambos os modelos mostraram altas taxas de sucesso, detalhadas na Tabela 2. No entanto, em desafios mais complexos, como "*Mismatching brackets*" e "*Type mismatch*", o ChatGPT 4 superou o Le Chat Mistral, destacando-se especialmente onde o Le Chat Mistral falhou completamente, como na detecção de "*Mismatching brackets*", conforme registrado na Tabela 2.

LLM	Detect	Fix	Explanation
GPT4	41/50	44/50	31/41
Mistral	28/50	32/50	23/28

Tabela 1. Desempenho dos LLMs na detecção, correção e explicação de erros de compilação.

id	Language	LOC	Type	Error	GPT 4 text					Mistral				
					Detect?	Fix?	Explanation			Detect?	Fix?	Explanation		
							Code element	Type of Error	Location			Code element	Type of Error	Location
1	C++	7	Syntax	Missing semicolon	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2	C++	11	Syntax	Missing semicolon (2x)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
3	C++	8	Syntax	Mismatching parentheses	No	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes
4	C++	8	Syntax	Mismatching parentheses	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
5	C++	11	Syntax	Mismatching parentheses	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
6	C++	11	Syntax	Mismatching parentheses	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No
7	C++	10	Syntax	Mismatching brackets	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
8	C++	11	Syntax	Mismatching brackets	Yes	Yes	Yes	Yes	Yes	Partially	Yes	No	No	Yes
9	C++	35	Syntax	Mismatching brackets	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
10	C++	10	Syntax	Mismatching quotes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
11	C++	10	Syntax	Invalid variable name (3x)	Yes	Yes	Yes	Yes	Partially	Yes	Yes	Yes	Yes	Yes
12	C++	7	Semantic	Variable not declared (2x)	Partially	Yes	Yes	Yes	Partially	Partially	Yes	Yes	Yes	Partially
13	C++	15	Semantic	Type mismatch	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
14	C++	16	Semantic	Variable out of scope	No	No	No	No	No	No	No	No	No	No
15	C++	12	Semantic	Variable out of scope	Yes	Yes	Yes	No	Yes	No	No	No	No	No
16	C++	13	Semantic	Variable out of scope	No	No	No	No	No	No	No	No	No	No
17	C++	15	Semantic	Variable out of scope	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
18	C++	41	Semantic	Variable out of scope	Yes	Yes	Yes	Yes	Partially	Yes	Yes	Yes	Yes	Yes
19	C++	31	Semantic	Variable out of scope	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
20	C++	26	Semantic	Variable not declared	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
21	C++	55	Semantic	Function signature mismatch	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
22	C++	23	Semantic	Variable out of scope	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
23	C++	55	Semantic	Variable out of scope (3x)	No	No	No	No	No	No	No	No	No	No
24	C++	48	Syntax	Missing semicolon (2x)	Yes	Yes	Yes	Partially	Yes	No	No	No	No	No
25	C++	28	Semantic	Variable out of scope	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
26	C++	69	Semantic	Variable redefinition	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	No
27	C++	39	Syntax	Missing semicolon	Yes	Yes	Yes	Partially	Yes	Yes	Yes	Yes	Yes	Yes
28	C++	23	Syntax	Missing semicolon	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
29	C++	29	Syntax	Missing semicolon	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
30	C++	18	Syntax	Invalid macro usage	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
31	Java	24	Semantic	Dereference of primitive type	Yes	Yes	Yes	Partially	Yes	Yes	Yes	Yes	Yes	Yes
32	Java	33	Syntax	Missing semicolon	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
33	Java	35	Syntax	Missing semicolon	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
34	Java	27	Syntax	Illegal character	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes
35	Java	31	Semantic	Variable not declared	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
36	Java	29	Syntax	Mismatching parentheses	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
37	Java	32	Syntax	Mismatching brackets	No	No	No	No	No	No	Yes	No	No	No
38	Java	30	Semantic	Type mismatch	No	No	No	No	No	No	No	No	No	No
39	Java	21	Syntax	Mismatching brackets	No	No	No	No	No	No	No	No	No	No
40	Java	55	Semantic	Type mismatch (2x)	Partially	Yes	Yes	Yes	Partially	Partially	Yes	No	Yes	No
41	C	24	Syntax	Missing operand	Yes	Yes	Yes	No	Yes	No	No	Yes	No	No
42	C	25	Syntax	Mismatching brackets	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
43	C	25	Semantic	Type mismatch	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
44	C	23	Semantic	Variable out of scope	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
45	C	26	Syntax	Invalid return type	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
46	C	15	Semantic	Function not defined	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
47	C	18	Syntax	Mismatching quotes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
48	C	18	Syntax	Mismatching parentheses	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
49	C	38	Semantic	Variable not declared	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
50	C	29	Semantic	Operator not defined	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Tabela 2. Resultados da avaliação de produtos.

3.3 Discussão

Nessa seção, analisaremos os resultados alcançados.

3.3.1 Detecção de Erros de Compilação

Na análise dos tipos específicos de erros de compilação detectados pelos LLMs, observa-se uma performance variável em diferentes categorias de erros. Notavelmente, erros de sintaxe

como a falta de ponto e vírgula ("*Missing semicolon*") e parênteses desemparelhados ("*Mismatching parentheses*") apresentaram altas taxas de detecção por ambas os LLMs, com o ChatGPT 4 identificando todos os 8 casos de "*Missing semicolon*" e o Mistral identificando 7 desses casos. A detecção de "*Mismatching brackets*" foi particularmente forte no ChatGPT 4, detectando 6 de 6 casos, enquanto o Le Chat Mistral não detectou

nenhum, destacando uma discrepância significativa no desempenho entre os dois LLMs para esse tipo de erro.

Por outro lado, os erros semânticos como "*Variable not declared*" e "*Type mismatch*" mostraram uma detecção mais desafiadora, com ambas os LLMs apresentando resultados moderados. ChatGPT 4 e Le Chat Mistral detectaram 3 e 3 casos de "Variable not declared", respectivamente, de um total de 4 casos. Em erros como "*Function not defined*" e "*Operator not defined*", o ChatGPT 4 teve sucesso em todos os casos testados, enquanto o Mistral falhou em detectar alguns, o que indica uma potencial área de melhoria para o modelo Mistral.

Essa variabilidade no desempenho dos LLMs sugere que, embora ambas sejam ferramentas úteis para a identificação de erros de compilação, existem diferenças significativas na maneira como cada modelo processa e responde a diferentes tipos de erros. Isso pode ser devido às diferenças nas arquiteturas subjacentes dos LLMs, seus conjuntos de treinamento, ou como as características específicas de cada linguagem de programação são interpretadas e analisadas por cada modelo.

Um dos erros específicos analisados durante a avaliação foi o de "*variable out of scope*", onde o ChatGPT 4 conseguiu identificar corretamente 7 de 10 ocorrências deste tipo de erro. A análise desses casos revelou que os modelos de LLM possuem uma capacidade sólida de identificar variáveis que são utilizadas fora do seu escopo permitido. No entanto, duas das situações em que os erros não foram detectados envolvem um cenário comum em programação C++: a declaração de uma variável no cabeçalho de um loop `for` e a tentativa de acesso a essa variável imediatamente após o término do loop. Este tipo de erro é ilustrado na Figura 1, demonstrando uma área específica onde a detecção de erros pelos LLMs ainda necessita de melhorias.

A taxa de acerto de 70% na detecção de variáveis fora de escopo indica uma competência considerável dos LLMs neste tipo de análise de erro. No entanto, a dificuldade em identificar erros envolvendo escopos limitados a blocos específicos, como aqueles introduzidos por loops, sugere uma oportunidade clara para aprimoramento.

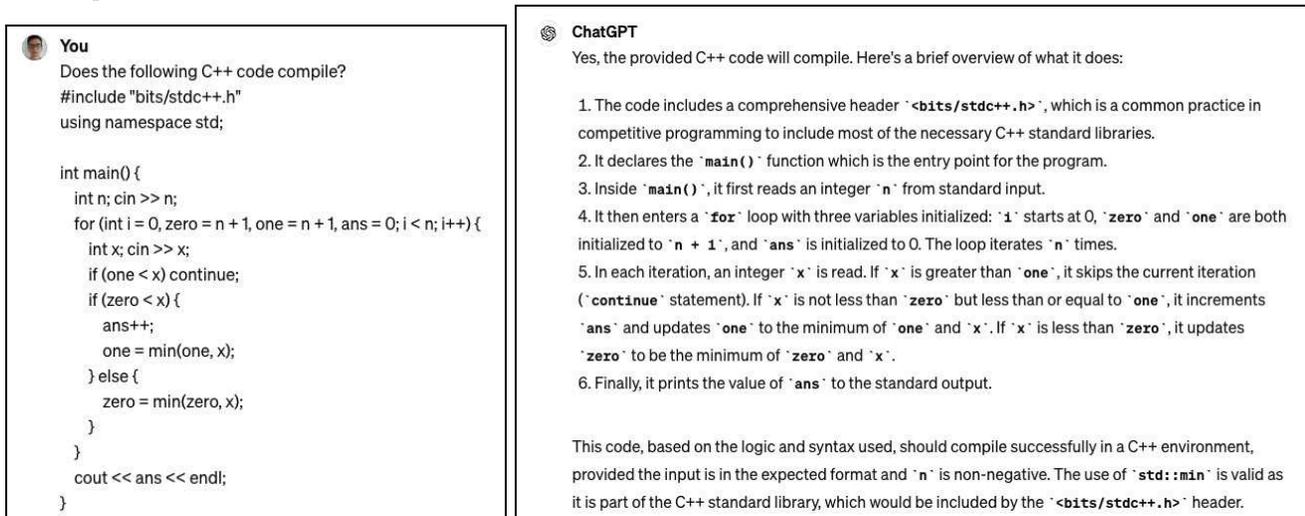


Figura 1. ChatGPT 4 não detecta a utilização de uma variável no cabeçalho de um loop após o término do loop.

3.3.2 Correção de Erros de Compilação

A análise dos resultados da avaliação de produtos revela um aspecto notável da performance dos LLMs: sua proatividade em sugerir mudanças sempre que um erro é detectado. Esta tendência é evidente em todos os casos analisados, indicando que os modelos não apenas identificam os erros mas também se empenham em fornecer soluções viáveis para corrigi-los.

Observou-se que em alguns casos onde os LLMs não detectaram explicitamente um erro, eles ainda assim sugeriram alterações no código. Curiosamente, essas mudanças propostas, embora inicialmente não direcionadas a corrigir um erro específico identificado, acabaram por resolver o problema. Isso resultou em

uma quantidade de correções efetivas que excedeu o número de detecções de erros de compilação.

Um exemplo pode ser visto na Figura 2. O LLM não identificou inicialmente um erro no elemento `for` (a falta de um parêntesis fechando), mas sugeriu uma mudança para utilizar um loop baseado em range, que não apenas simplifica o código, mas também elimina a necessidade do cast, corrigindo implicitamente o problema de compilação (ver Figura 2). Esta sugestão não só aprimora a clareza e a manutenção do código, como também evita potenciais erros que poderiam surgir do uso inadequado de tipos de dados.

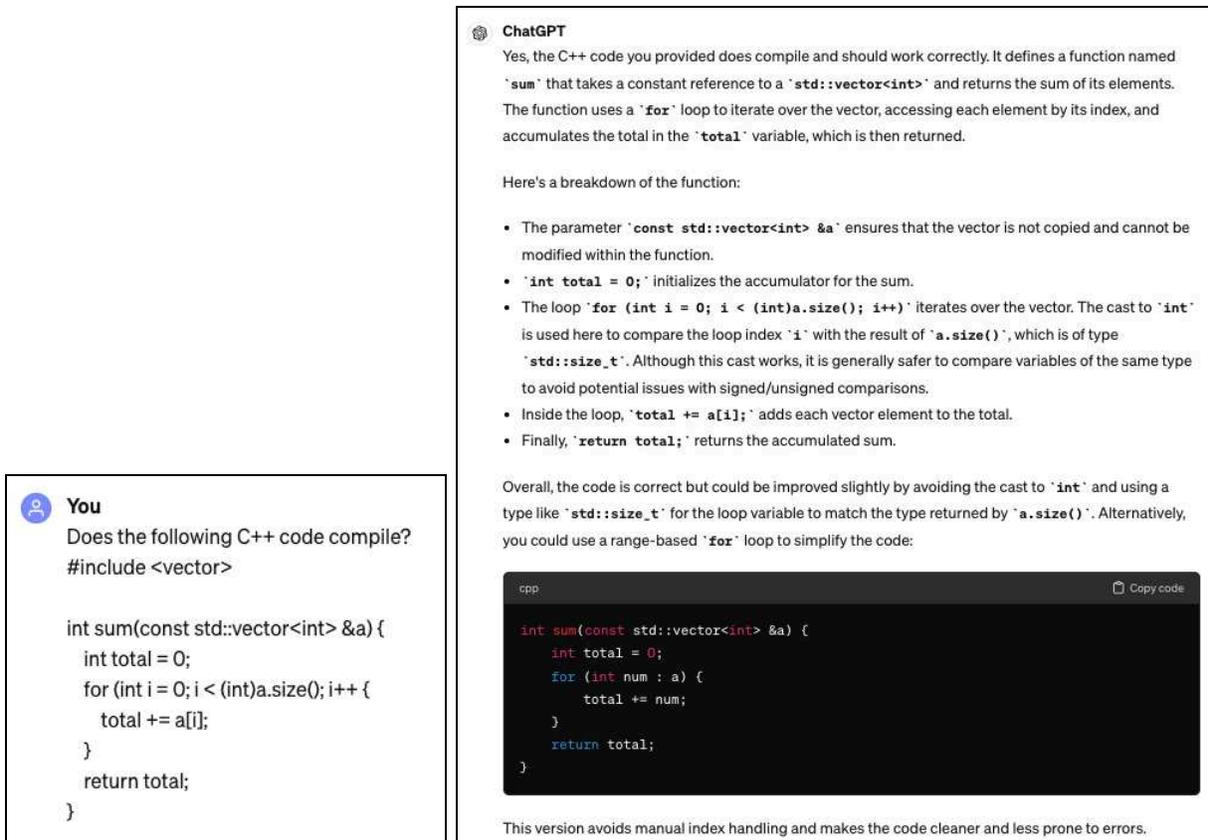


Figura 2. ChatGPT 4 não detecta o erro de compilação, mas acaba corrigindo-o.

3.3.3 Explicação

A explicação de erros de compilação por parte dos LLMs, como o ChatGPT 4 e o Le Chat Mistral, revela aspectos significativos da interação entre a compreensão do modelo e a apresentação de informações precisas. Enquanto alucinações, ou a geração de informações incorretas e fictícias pelos LLMs, são um problema conhecido [1], os resultados desta avaliação demonstram que os modelos frequentemente fornecem explicações coerentes e úteis, mesmo em situações onde a detecção inicial pode parecer incerta.

Nos resultados apresentados, observamos que, em alguns casos, os LLMs inicialmente indicam que não há erros de compilação, mas ao desenvolver a resposta, reconhecem a presença de problemas, ajustando suas conclusões iniciais. Esta mudança de posição, embora possa parecer uma inconsistência, raramente compromete a qualidade das explicações fornecidas. As respostas finais, que incluem a correção da posição inicial do modelo, geralmente oferecem uma explicação detalhada sobre a natureza e o contexto do erro detectado.

Por exemplo, o ChatGPT 4 conseguiu fornecer explicações satisfatórias para 31 dos 41 erros detectados, e o Le Chat Mistral explicou adequadamente 23 dos 28 erros detectados. Estes números refletem uma taxa relativamente alta de sucesso na explicação correta dos problemas.

Seguindo nossas métricas para avaliação da "Explanation", os resultados foram considerados satisfatórios. A capacidade dos modelos de ajustar suas respostas e finalmente oferecer explicações coerentes é um indicativo de que, apesar das

alucinações potenciais e da variação inicial nas detecções, os LLMs estão se aproximando de uma aplicação prática robusta. Eles mostram uma habilidade notável de refinar suas análises com base no desenvolvimento da interação, o que é crucial para aplicações em ambientes de desenvolvimento de software real, onde a precisão da informação é crítica.

3.4 Ameaças à Validade

Algumas ameaças à validade podem surgir no contexto do uso de LLMs [6]. Um fator crucial que pode comprometer a validade dos resultados é o viés na seleção de códigos, pois a escolha de amostras que não representem adequadamente a diversidade de erros reais pode distorcer a avaliação dos LLMs. Além disso, embora os compiladores utilizados como baseline sejam geralmente confiáveis, eles não estão isentos de bugs. Para mitigar esse risco, cada código foi verificado manualmente pelo autor, assegurando a precisão dos dados de compilação. Essa abordagem busca minimizar as ameaças à validade do estudo e garantir uma interpretação correta dos resultados, refletindo uma análise honesta e precisa do desempenho dos LLMs em detectar e corrigir erros de compilação.

3.5 Respostas às Questões de Pesquisa

A seguir apresentamos as respostas às questões de pesquisa:

- **QP1.** Até que ponto o ChatGPT 4 consegue detectar erros de compilação em produtos?
 - O ChatGPT 4 demonstrou uma capacidade robusta na detecção de erros de compilação, identificando corretamente 41 dos 50 erros em

produtos. Esse resultado reflete uma taxa de sucesso de 82%, indicando que o ChatGPT 4 é bastante eficaz em identificar uma ampla gama de erros de compilação, incluindo tanto erros sintáticos quanto semânticos. A eficiência do modelo na detecção de erros complexos e variados mostra que ele pode ser uma ferramenta valiosa para desenvolvedores no processo de depuração e validação de código.

- **QP₂.** Até que ponto o Le Chat Mistral consegue detectar erros de compilação em produtos?
 - Le Chat Mistral identificou 28 dos 50 erros, resultando em uma taxa de sucesso de 56%. Embora este resultado mostre que o Mistral tem a capacidade de detectar mais da metade dos erros de compilação apresentados, ele é consideravelmente menos eficaz que o ChatGPT 4. Essa performance sugere que, enquanto o Mistral pode ser útil em cenários onde a detecção de erros menos complexos é necessária, ele pode não ser tão confiável quanto o ChatGPT 4 para aplicações críticas que exigem alta precisão na detecção de erros de compilação em produtos de software.

4. AVALIAÇÃO 2: LPS

Nesta seção apresentamos a avaliação para detecção de erros de compilação em linhas de produtos de software.

4.1 Metodologia

A seguir apresentamos a metodologia usada para avaliar a detecção de erros de compilação em linha de produtos.

4.1.1 GQM

O objetivo é avaliar a eficácia dos LLMs, especificamente ChatGPT 4 e Le Chat Mistral, com o propósito de identificar erros de compilação do ponto de vista dos desenvolvedores no contexto de linha de produtos.

Nós avaliamos as seguintes questões de pesquisa:

QP₁. Até que ponto o ChatGPT 4 consegue detectar erros de compilação em linha de produtos?

QP₂. Até que ponto o Le Chat Mistral consegue detectar erros de compilação em linha de produtos?

4.1.2 Planejamento

Seleção de Linhas de Produto: Incluímos 30 Linhas de Produto de Software que variam de 4 a 33 linhas de código (mediana: 16,8 LOC e média: 16 LOC). Cada linha de produto possui 1 a 5 macros, contendo de um a dois tipos de erros de compilação.

- Origem dos Códigos:
 - 14 linhas de produto são criações originais dos autores.
 - 6 linhas de produtos são baseadas nos estudos de Braz et al. sobre análise e compilação de sistemas configuráveis com diretivas `#ifdef` [15, 16].
 - As linhas restantes são adaptadas do estudo de Abal et al. sobre bugs de variabilidade no

kernel do Linux, apresentando versões simplificadas dos códigos originais [14].

Formulação do Prompt: Utilizamos o *prompt* "Does the following C code compile? `{code}`", onde `{code}` é o trecho de código. Esse prompt foi escolhido para manter a simplicidade e focar na interação direta com os LLMs, visando avaliar a capacidade dos modelos de entender e processar a compilação condicional. Usamos Inglês já que os LLMs são treinados em uma quantidade bem maior de dados neste idioma.

Compilação e Validação: Cada linha de produto é compilada usando o compilador GNU GCC 11 para C. Durante o processo, realizamos uma análise manual para contar quantos produtos únicos podem ser gerados a partir da ativação de diferentes features. Cada produto único é compilado manualmente para verificar quantas configurações possuem erros de compilação. As 30 linhas de produto de software geram um total de 103 produtos únicos, dos quais 40 contêm erros de compilação.

Análise Qualitativa das Respostas: A análise das respostas dos LLMs segue a mesma estrutura utilizada na avaliação de produtos individuais, detalhada na Seção 3.1.2, com adaptações específicas para o contexto de linhas de produto de software:

- **Detect:** Neste contexto, "Detect" indica a quantidade de produtos distintos com erros de compilação que foram efetivamente identificados pelos LLMs.
- **Fix:** Para considerar uma correção como "Yes", a solução proposta pelo LLM deve ser geral e aplicável a todos os produtos afetados, sem recorrer a ajustes específicos como a definição direta de macros no código que garantam a compilação apenas daquela configuração específica. O critério busca avaliar a capacidade do modelo de propor correções que sejam sustentáveis e generalizáveis, mantendo a integridade da funcionalidade do produto sem intervenções manuais específicas.
- **Explanation:** A avaliação da explicação mantém a mesma abordagem da análise individual de produtos, considerando se o LLM consegue elucidar satisfatoriamente o problema detectado. Isso inclui identificar corretamente o elemento de código causador do erro, a natureza do erro e a localização específica do problema dentro do código.

Modelos Avaliados: A análise inclui dois modelos de LLMs, ChatGPT 4 e Le Chat Mistral, permitindo uma comparação direta das capacidades de cada um em contextos similares de teste em linhas de produtos. As análises foram realizadas em Abril de 2024.

4.2 Resultados

Os resultados da avaliação das LPS usando os LLMs ChatGPT 4 e Le Chat Mistral são apresentados na Tabela 5, com um sumário presente na tabela 3, destacando a eficiência de cada modelo em identificar e resolver erros de compilação em contextos de desenvolvimento de sistemas configuráveis.

LLM	Detected Product Lines	Detected Products	Fix	Explanation
GPT4	29/30	38/40	12/30	26/29
Le Chat Mistral	26/30	31/40	9/30	18/26

Tabela 3. Desempenho dos modelos de LLM ChatGPT 4 e Le Chat Mistral na detecção, correção e explicação de erros em Linhas de Produto de Software.

ChatGPT 4 demonstrou uma capacidade notável na identificação de linhas de produtos com erros, detectando algum erro de compilação em 29 das 30 linhas de produtos testadas. Este modelo também identificou erros em 38 dos 40 produtos individuais derivados dessas linhas, sendo os erros não detectados "Type Mismatch" e "Variable not declared". Em um caso, o ChatGPT 4 detectou apenas um dos dois produtos com erro de compilação. Em termos de correções, ChatGPT 4 propôs soluções eficazes para 12 das 30 linhas de produtos. Quando se trata de explicações, o modelo foi capaz de fornecer explicações adequadas para 26 das 29 LPS detectadas, refletindo uma competência em não apenas detectar, mas também explicar os erros detectados.

Le Chat Mistral, enquanto isso, detectou erros de compilação em 26 das 30 linhas de produtos e em 31 dos 40 produtos individuais. O modelo conseguiu propor correções para 9 das 30 linhas de produtos e forneceu explicações adequadas para 18 das 26 linhas

de produtos detectadas. Em duas LPS, Le Chat Mistral detectou erros de compilação em alguns produtos, mas não em todos. Um sumário dos erros de compilação não detectados pelo Le Chat Mistral pode ser encontrado na Tabela 4. Esses resultados mostram que, embora eficaz em detectar uma gama razoável de erros, Le Chat Mistral enfrenta mais desafios em propor correções efetivas e fornecer explicações detalhadas.

Error	#Produtos não detectados
Type not declared	2
Function redefinition	1
Function signature mismatch	4
Variable not declared	1
Invalid return type	1

Tabela 4. Erros de compilação não detectados pelo Le Chat Mistral.

id	Language	Macros	#Prod.	LOC	Type	Error	#NC	GPT4 text						Mistral					
								Detect	Fix?	Explanation			Detect	Fix?	Explanation				
										Code Elem.	Type	Location			Code Elem.	Type	Location		
1	C	2	4	9	Semantic	Type not declared	1	1	Yes	Yes	Yes	Yes	0	No	No	No	No		
2	C	2	4	16	Semantic	Func. not defined; Func. redefinition	2	2	Partially	Yes	Yes	Yes	1	No	Yes	Partially	Yes		
3	C	1	2	10	Syntax	Mismatching quotes	1	1	Yes	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
4	C	1	2	11	Semantic	Function not defined	1	1	Yes	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
5	C	1	2	8	Semantic	Variable not declared	1	1	Yes	Yes	Yes	Yes	1	No	Yes	Yes	Yes		
6	C	1	2	13	Syntax	Mismatching brackets	1	1	Yes	Yes	Yes	Yes	1	No	No	No	Yes		
7	C	1	2	9	Syntax	Missing semicolon	1	1	Yes	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
8	C	2	4	13	Semantic	Type not declared	1	1	No	Yes	Yes	No	0	No	No	No	No		
9	C	2	4	16	Semantic	Variable not declared	1	1	No	Yes	Yes	Yes	1	No	Yes	Yes	Yes		
10	C	2	4	22	Semantic	Function signature mismatch (2x)	2	2	No	Yes	Yes	Yes	0	No	No	No	No		
11	C	2	4	23	Semantic	Func. sig. mismatch (2x); Var. not decl.	3	3	No	Yes	Yes	Yes	0	Yes	No	No	No		
12	C	1	2	17	Semantic	Label not declared	1	1	Yes	Yes	Yes	Yes	1	No	No	No	No		
13	C	2	4	11	Semantic	Type not declared	1	1	No	Yes	Yes	Yes	1	No	Yes	No	Yes		
14	C	1	2	5	Syntax	Missing struct keyword	1	1	No	Yes	No	Yes	1	Yes	Yes	Yes	Yes		
15	C	1	2	10	Semantic	Label not declared	1	1	No	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
16	C	1	2	4	Syntax	Invalid return type	1	1	Yes	Yes	Yes	Yes	1	Partially	Yes	Yes	Yes		
17	C	1	2	14	Semantic	Continue statement outside a loop	1	1	Yes	Yes	Yes	Yes	1	No	Yes	Yes	Yes		
18	C	1	2	12	Semantic	Address of bitfield requested	1	1	No	Yes	Yes	Yes	1	No	No	No	No		
19	C	3	5	32	Semantic	Func. not defined; Var. not declared	2	1	No	Partially	Partially	Partially	2	No	Yes	Yes	Yes		
20	C	3	5	27	Semantic	Variable not declared	1	1	No	Yes	Yes	Yes	1	No	No	No	No		
21	C	2	4	25	Syntax	Invalid return type (3x)	3	3	No	Yes	Yes	Yes	2	No	No	No	No		
22	C	2	4	20	Semantic	Function not defined	1	1	Yes	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
23	C	2	4	19	Semantic	Function not defined	1	1	Yes	Yes	Yes	Yes	1	No	Yes	Yes	Yes		
24	C	2	4	19	Semantic	Function not defined	1	1	No	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
25	C	5	8	33	Semantic	Variable not declared (3x)	3	3	No	Yes	Yes	Yes	3	No	Yes	Yes	Yes		
26	C	4	8	26	Semantic	Function not defined (2x)	2	2	No	Yes	Yes	Yes	2	No	No	No	No		
27	C	2	3	26	Semantic	Type mismatch	1	0	No	No	No	No	1	No	Yes	Yes	Yes		
28	C	1	2	11	Semantic	Function signature mismatch	1	1	No	Yes	Yes	Yes	1	Yes	Yes	Yes	Yes		
29	C	2	4	22	Semantic	Function redefinition	1	1	Yes	Yes	Yes	Yes	1	No	Yes	Yes	Yes		
30	C	3	2	21	Semantic	Macro not defined	1	1	No	Yes	Yes	Yes	1	No	Yes	Yes	Yes		

Tabela 5. Resultados da avaliação de linha de produtos.

The code you've provided will compile, but whether it will run correctly or not depends on the definitions of the preprocessor macros 'A' and 'B'.

If 'A' is defined, the function 'f' expects an integer argument. If 'A' is not defined, 'f' does not expect any arguments. Similarly, if 'B' is defined, the function 'f' is called in 'main', otherwise it is not.

If 'A' is not defined and 'B' is defined, the code will compile but will cause a runtime error because 'f' is called without any arguments. If 'A' is defined and 'B' is not defined, 'f' is never called so there's no issue. If both 'A' and 'B' are defined, 'f' is called with an argument which is fine. If neither 'A' nor 'B' is defined, 'f' is never called so there's no issue.

So, the code will compile in all cases, but it will run correctly only if 'A' and 'B' are defined appropriately.

ChatGPT

The provided C code can compile and run, assuming the environment in which it is compiled does not define the macros 'A' and 'B'. Here's a breakdown of how the compilation and execution would proceed based on whether 'A' and 'B' are defined or not:

- Macro 'A' is not defined:**
 - The function 'f()' is defined as 'int f() { return val++; }'. This version of the function takes no arguments and simply returns the value of 'val', incrementing 'val' by one post-return.
 - Since 'A' is not defined, the function 'f(int x)' that requires an argument is not used.
- Macro 'B' is not defined:**
 - In 'main()', the variable 'foo' is initialized to '3'.
 - Since 'B' is not defined, the statement 'foo += f();' is not included in the compilation. Thus, 'foo' remains unchanged and retains the value '3'.
- Output:**
 - The 'printf()' statement in 'main()' outputs the value of 'foo', which is '3'.

If either 'A' or 'B' were defined, the behavior would change accordingly:

- If 'A' is defined, 'f(int x)' would be used instead, and calling 'f()' without an argument (as done when 'B' is also defined) would result in a compilation error due to the wrong number of arguments.
- If 'B' is defined, the statement 'foo += f();' would attempt to add the result of 'f()' to 'foo'. However, this would result in a compilation error if 'A' is not defined because the function 'f()' does not take arguments but would be called without them.

To ensure that the code runs without issues, you would need to check if the macros 'A' and 'B' are defined and adjust the function calls and definitions accordingly.

Figura 3. Análise comparativa das respostas dos LLMs, Le Chat Mistral (à esquerda) e ChatGPT 4 (à direita), em relação à linha de produto apresentada na Listagem 1.

Na Figura 3, é apresentada as respostas dos dois LLMs em relação à linha de produto apresentada na Listagem 1. Observamos que o ChatGPT 4 corretamente identifica o problema quando A e B estão definidos, enquanto que o Le Chat Mistral não identifica o problema.

4.3 Discussão

Nessa seção, analisaremos os resultados obtidos.

4.3.1 Detecção de Erros de Compilação

A análise dos resultados obtidos na avaliação das Linhas de Produto de Software utilizando LLMs revela *insights* significativos sobre a capacidade dessas ferramentas em identificar erros de compilação em programas escritos em linguagem C. A performance de ambos os modelos, ChatGPT 4 e Le Chat Mistral, sugere que, embora eficazes em muitos casos, ainda existem áreas significativas para melhoria e ajustes. Notavelmente, a maioria dos erros não detectados por ambos os modelos foram de natureza semântica: os dois erros não detectados do ChatGPT 4 e oito dos nove erros não detectados do Le Chat Mistral eram semânticos, revelando uma possível área de melhoria.

Uma observação crítica é a compreensão incompleta da linguagem C pelos LLMs, como evidenciado em uma situação específica descrita na Figura 4. Nesse exemplo, a função *norm* adapta seus cálculos dependendo da definição das macros A e B. Quando a macro A não está definida, a função *norm* é

configurada para aceitar apenas dois parâmetros, o que contraria a chamada feita na função main, onde *norm* é invocada com três argumentos. Essa inconsistência deveria resultar em um erro de compilação, mas o Le Chat Mistral, ao analisar os possíveis produtos, apenas identifica isso como um aviso (*warning*) e não como um erro crítico, falhando assim em reconhecer a gravidade do problema.

Com base nos resultados, observamos que os LLMs, especialmente o ChatGPT 4, buscam identificar padrões lógicos nas linhas de produto ao invés de tentar abordagens para checar a boa formação de todos os produtos da LPS usando a estratégia força bruta. Um exemplo dessa capacidade é ilustrado na Figura 5, onde a função *f* depende da definição das macros A, B e C. O ChatGPT 4 demonstrou entendimento de que, se a macro A não está definida, a função *f* e qualquer código dentro de seu bloco não serão compilados, tornando irrelevantes outras verificações internas como as de B e C para essa compilação.

Essa capacidade de reconhecer e priorizar as dependências de macros evita a análise redundante de configurações que não afetam o resultado final da compilação, otimizando o processo de detecção de erros. Este raciocínio não apenas melhora a eficiência da análise, mas também destaca a sofisticação do modelo em compreender e aplicar a lógica de pré-processamento, que é crucial para gerenciar a complexidade das linhas de produtos de software.

<p>LA Does the following C code compile? #include <stdio.h></p> <pre> 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; } </pre>	<p> The C code you've provided will compile, but the behavior of the <code>norm</code> function will depend on whether the preprocessor macros <code>A</code> and <code>B</code> are defined or not.</p> <p>If both <code>A</code> and <code>B</code> are defined, the function will compile and run correctly, because it will include the <code>y</code> parameter and the line of code that uses it. The output will be <code>5</code>, because <code>w</code> is the sum of <code>x * z</code> and <code>y</code>, which is <code>1 * 3 + 2</code>.</p> <p>If <code>A</code> is not defined, the function will still compile, but the <code>y</code> parameter will not be included in the function signature. This means that the call <code>norm(1, 2, 3)</code> will pass <code>2</code> as the value of <code>z</code> and <code>3</code> will be ignored. The output will be <code>2</code>, because <code>w</code> is just <code>x * z</code>, which is <code>1 * 2</code>.</p> <p>If <code>A</code> is defined but <code>B</code> is not, the function will still compile, but the line of code that uses <code>y</code> will be excluded. This means that the value of <code>y</code> will be ignored. The output will be <code>3</code>, because <code>w</code> is just <code>x * z</code>, which is <code>1 * 3</code>.</p> <p>If neither <code>A</code> nor <code>B</code> is defined, the function will still compile, but both the <code>y</code> parameter and the line of code that uses it will be excluded. This means that the call <code>norm(1, 2, 3)</code> will pass <code>2</code> as the value of <code>z</code> and <code>3</code> will be ignored. The output will be <code>2</code>, because <code>w</code> is just <code>x * z</code>, which is <code>1 * 2</code>.</p> <p>So, the code will always compile, but the behavior of the <code>norm</code> function will depend on the definition of the preprocessor macros <code>A</code> and <code>B</code>.</p>
--	--

Figura 4. Exemplo de código C demonstrando condições de pré-processamento. O Le Chat Mistral falha em reconhecer que a variação no número de argumentos em uma chamada de função, dependendo das definições de macro, resulta em um erro de compilação, apesar de identificar o problema.

<p>You</p> <p>Does the following C code compile? #include <stdio.h></p> <pre> #ifdef A int f(int a[], int n) { #ifdef B int x = 0; #endif #ifdef C int y = 0; #endif #if defined(B) && defined(C) int z = 0; #endif int sum = 0; for (int i = 0; i < n; i++) { sum += a[i]; } #ifdef B x += sum + y; #endif #if defined(B) && defined(C) y += a[i]; z += a[i]; sum += x; #endif } return sum; } #endif int main() { int a[5] = {1, 2, 3, 4, 5}; printf("%d\n", f(a, 5)); return 0; } </pre>	<p>ChatGPT</p> <p>The provided C code does not compile as currently written unless the preprocessor symbol <code>A</code> is defined before or during the compilation process. This is due to the use of conditional compilation directives (<code>#ifdef</code>, <code>#endif</code>) that control which parts of the code are included during compilation.</p> <p>In your code snippet, the function <code>f(int a[], int n)</code> is wrapped within <code>#ifdef A ... #endif</code>. This means that if <code>A</code> is not defined, the function <code>f</code> will not be included in the compiled program. Consequently, when <code>main</code> attempts to call <code>f(a, 5)</code>, the function <code>f</code> will not exist, leading to a compilation error due to the undefined reference to <code>f</code>.</p> <p>If <code>A</code> is defined, the function <code>f</code> is included, and the code within <code>f</code> depends further on whether <code>B</code> and <code>C</code> are defined:</p> <ul style="list-style-type: none"> If neither <code>B</code> nor <code>C</code> is defined, the code compiles and runs, treating <code>x</code>, <code>y</code>, and <code>z</code> as undeclared identifiers that are not used. If <code>B</code> is defined but <code>C</code> is not, the variable <code>x</code> is defined and updated in the loop, but <code>y</code> remains undeclared and unused. If both <code>B</code> and <code>C</code> are defined, <code>x</code>, <code>y</code>, and <code>z</code> are all declared and used in calculations inside the loop. <p>To ensure the code compiles under all conditions:</p> <ol style="list-style-type: none"> Ensure <code>A</code> is defined if you want the function <code>f</code> to be part of the compiled program. If <code>B</code> and <code>C</code> have specific roles as shown, make sure their usage is enclosed appropriately within the defined checks to prevent undeclared identifiers. If the intent is to compile and run this code snippet as it is, you would need to define <code>A</code> (at least) for the compilation to succeed and for the main function to work as expected. You can define <code>A</code> (and optionally <code>B</code> and <code>C</code>) either by adding <code>#define A</code> (and <code>#define B</code>, <code>#define C</code> as needed) at the beginning of your source file or by using compiler options to define these preprocessor symbols (e.g., <code>-DA -DB -DC</code> if using gcc or clang).
--	---

Figura 5. Análise do ChatGPT de um código C com diretivas de pré-processamento. A figura mostra a resposta do ChatGPT ao código que inclui múltiplas condições de compilação baseadas nas macros A, B, e C.

4.3.2 Correção de Erros de Compilação

Durante a avaliação das Linhas de Produto de Software, observou-se uma diminuição na proatividade dos LLMs em sugerir correções para os erros de compilação detectados, em comparação com a avaliação de produtos individuais. Apesar dessa diminuição, os resultados ainda foram promissores, indicando que, mesmo em cenários mais complexos, os LLMs são capazes de identificar e sugerir intervenções válidas, embora com menos frequência.

Uma observação importante é que, enquanto os LLMs demonstraram competência em detectar e explicar os erros, a capacidade de sugerir correções eficazes foi menos consistente. Isto sugere que, embora os modelos tenham uma compreensão adequada dos erros, eles podem se beneficiar de direcionamentos mais específicos na hora de propor soluções. Isso nos leva a considerar melhorias para trabalhos futuros, onde o *prompt* utilizado para interagir com os LLMs pode ser ajustado para solicitar explicitamente uma correção. Alterar o *prompt* para incluir uma solicitação mais explícita de solução pode ajudar a orientar os modelos a não apenas identificar o problema, mas também a focar mais diretamente na geração de uma solução aplicável.

4.3.3 Explicação

O ChatGPT 4 mostrou um bom desempenho na explicação de erros, proporcionando detalhes consistentes e claros que ajudam na compreensão dos problemas detectados. Este modelo foi eficaz em elucidar os contextos dos erros de compilação e suas implicações, especialmente em casos de erros complexos como "Function not defined" e "Type not declared", onde as explicações foram detalhadas e informativas. Além disso, as respostas do ChatGPT são maiores do que as do Le Chat Mistral.

Le Chat Mistral, embora competente em alguns aspectos, apresentou limitações na elaboração de explicações detalhadas. O modelo foi menos consistente, especialmente em casos que requerem uma compreensão mais profunda das interações de múltiplos macros e suas consequências na lógica do código. A discrepância entre detecção de erros de compilação e a qualidade das explicações foi mais pronunciada, indicando que ainda há

espaço para melhorias significativas na precisão e profundidade das respostas.

Ambos os modelos enfrentaram desafios ao explicar erros envolvendo condições complexas de pré-processamento, frequentemente falhando em fornecer explicações que capturassem totalmente a natureza e a causa dos erros. Este desafio sugere que, apesar de úteis, os modelos ainda precisam de refinamento para lidar efetivamente com a complexidade das linhas de produto de software.

4.3.4 Prompt

Uma observação que surgiu durante a análise é a margem para melhorias na formulação do *prompt* utilizado para interagir com os LLMs. Observou-se, como exemplificado na Figura 6, que o Le Chat Mistral, apesar de interpretar corretamente o código em termos de sintaxe e funcionalidade básica, não conseguiu avaliar adequadamente o impacto das configurações de pré-processamento no comportamento do código. Mais especificamente, se a *macro* `ENABLE_PAM` estiver ativa, mas a *macro* `ENABLE_AUTH_MD5` não, o tipo `pam_userinfo` não estará definido e ocasionará em um erro de tipo não declarado.

Especificamente, o Le Chat Mistral parece abordar a análise do código como se ele fosse compilado em uma instância isolada, sem levar em conta diretivas de pré-processador como `#define`. Este comportamento indica uma compreensão limitada das implicações das condições de compilação variáveis, que são essenciais em ambientes de desenvolvimento real onde diferentes configurações podem afetar significativamente a execução do programa.

Essa observação sugere que um ajuste no *prompt* poderia potencialmente melhorar a capacidade do modelo de considerar diferentes cenários de compilação. Alterar a forma como o *prompt* é apresentado aos LLMs para incluir explicitamente considerações sobre as diretivas de pré-processamento poderia orientá-los a analisar não apenas a correção do código em uma configuração padrão, mas também sob diversas condições configuráveis.

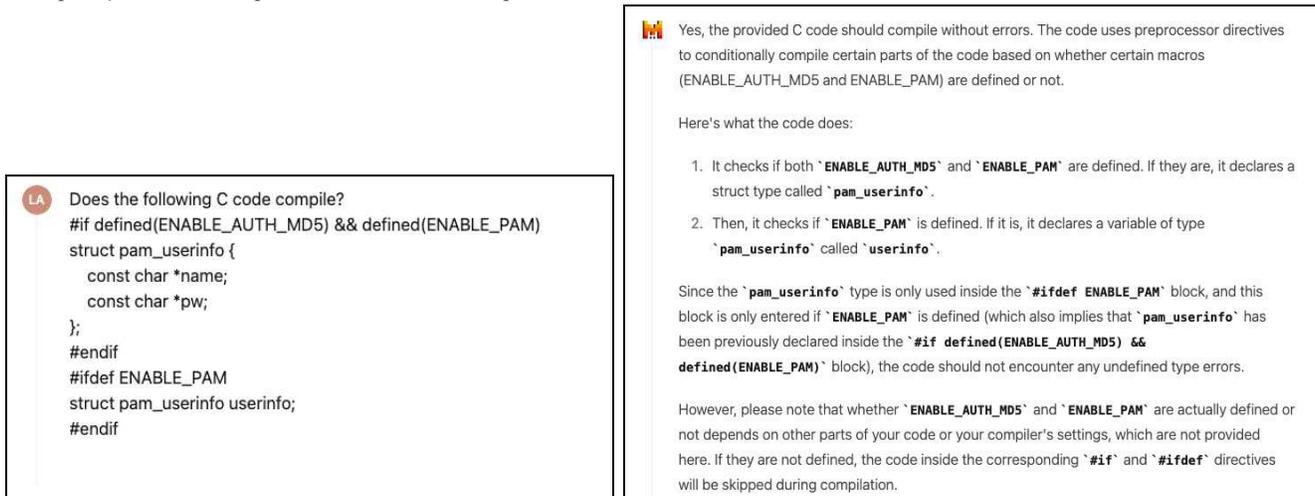


Figura 6. Le Chat Mistral não analisa a LPS em termos de possíveis configurações e acaba não detectando o problema.

Nesse exemplo da Figura 6, podemos pedir ao Le Chat Mistral para gerar encontrar uma configuração com erro de compilação, como é indicado na Figura 7. A adaptação do *prompt* para o Le Chat Mistral demonstrou ser uma estratégia eficaz para explorar e entender melhor as capacidades do modelo em cenários de teste mais complexos.

Este teste específico revelou que, quando desafiado de forma direta e específica, o Le Chat Mistral foi capaz de identificar uma configuração de macros que causaria um erro de compilação. Este

sucesso sugere que os LLMs, quando devidamente orientados por *prompts* que refletem cenários de uso mais detalhados e técnicos, podem de fato fornecer análises mais profundas e úteis, abrindo margens para futuras melhorias de *prompt*. Como trabalhos futuros, iremos avaliar o impacto de uso de outros padrões de *prompts* [8, 2] na detecção de erros de compilação em LPS.

LA In the provided code, find a way of defining macros such that there is a compilation error.

To cause a compilation error in the provided code, you can define the `ENABLE_PAM` macro without defining the `ENABLE_AUTH_MDS` macro. This will result in the `pam_userinfo` struct not being declared, but the `userinfo` variable of that type will still be declared, causing an "unknown type" error.

Here's an example of how to define the macros to cause a compilation error:

```
#define ENABLE_PAM

#if defined(ENABLE_AUTH_MDS) && defined(ENABLE_PAM)
struct pam_userinfo {
    const char *name;
    const char *pw;
};
#endif

#ifdef ENABLE_PAM
struct pam_userinfo userinfo; // This line will cause a compilation error
#endif
```

By defining `ENABLE_PAM` without defining `ENABLE_AUTH_MDS`, the code inside the `#if defined(ENABLE_AUTH_MDS) && defined(ENABLE_PAM)` block will be skipped, and the `pam_userinfo` struct will not be declared. Consequently, the `userinfo` variable declaration inside the `#ifdef ENABLE_PAM` block will cause a compilation error due to an unknown type.

Figura 7. Le Chat Mistral consegue encontrar um erro de compilação ao ser questionado diretamente sobre a existência de uma configuração com erro de compilação.

4.4 Ameaças à Validade

Na avaliação das Linhas de Produto de Software utilizando LLMs, algumas ameaças à validade dos resultados persistem, semelhantes às observadas na avaliação de produtos individuais. O viés na seleção de códigos é uma preocupação significativa, pois amostras que não capturam adequadamente a diversidade de erros encontrados em ambientes de desenvolvimento real podem levar a uma avaliação distorcida das capacidades dos LLMs. Além disso, enquanto os compiladores usados como baseline são geralmente confiáveis, a possibilidade de conterem bugs não pode ser totalmente descartada. Para atenuar esses riscos, cada linha de código foi meticulosamente verificada manualmente pelo autor, garantindo a precisão dos dados de compilação.

Além desses aspectos, uma limitação específica deste estudo foi o tamanho relativamente pequeno das linhas de produtos avaliadas, com a maior contendo apenas 33 linhas de código. Muitas das linhas de produto eram simplificações de códigos mais complexos, o que poderia potencialmente ter facilitado a tarefa dos LLMs na detecção e correção de erros. Essa simplificação pode não refletir completamente os desafios encontrados em cenários de software mais extensos e complexos, possivelmente inflando a percepção de eficácia dos modelos.

4.5 Respostas às Questões de Pesquisa

A seguir apresentamos as respostas às questões de pesquisa:

- **QP₁**. Até que ponto o ChatGPT 4 consegue detectar erros de compilação em linha de produtos?
 - O ChatGPT 4 mostrou uma excelente capacidade de detecção, identificando 29 das 30 linhas de produto e 38 dos 40 produtos individuais. Esses resultados confirmam a alta eficácia do ChatGPT 4 em reconhecer erros de compilação em cenários complexos de múltiplas configurações.
- **QP₂**. Até que ponto o Le Chat Mistral consegue detectar erros de compilação em linha de produtos?
 - Le Chat Mistral também detectou uma quantidade significativa de erros, identificando 26 das 30 linhas de produto e 31 dos 40 produtos individuais. Embora eficiente, sua performance é moderada em comparação ao ChatGPT 4, sugerindo que melhorias são necessárias para otimizar sua precisão na detecção de erros em ambientes complexos.

5. TRABALHOS RELACIONADOS

Algumas ferramentas conscientes de variações foram propostas anteriormente, como o TypeChef [12] e o SuperC [11], para a detecção de alguns erros de sintaxe e de tipos em sistemas configuráveis em C. Essas ferramentas propuseram técnicas para em seguida implementar análises estáticas não triviais para identificar erros de compilação em sistemas reais. Para a utilização dessas ferramentas, os usuários precisam realizar algumas configurações. O nosso trabalho avalia até que ponto LLMs conseguem realizar análises conscientes de variações. O usuário não possui esforço na sua utilização, bastando usar um *prompt* e adicionar o código do sistema configurável.

Abal et al. [13, 14] identificaram 98 bugs em sistemas configuráveis em C e estudaram as suas características. Alguns desses bugs estão relacionados a erros de compilação, e foram considerados no nosso trabalho. Como trabalhos futuros, pretendemos avaliar até que ponto LLMs conseguem identificar outros bugs, como fraquezas, além de erros de compilação, e utilizaremos esse conjunto de exemplos catalogados [13, 18]. Por exemplo, podemos avaliar um conjunto de fraquezas em sistemas configuráveis que foi identificado por Muniz et al. [17].

Medeiros et al. [21] propuseram uma técnica para identificar um conjunto de erros de sintaxe em sistemas configuráveis em C. Posteriormente, Medeiros et al. [22] propuseram uma técnica para a identificação de nomes utilizados mas não declarados. As duas técnicas conseguiram identificar erros de compilação reais em sistemas configuráveis em C. Na nossa avaliação, alguns exemplos remetem a alguns dos bugs encontrados pelos trabalhos anteriores.

Braz et al. [15, 16] propuseram uma técnica para detectar erros de compilação em sistemas configuráveis em C, que se baseia na análise do impacto da mudança. Uma análise estática não trivial foi proposta para a identificação de novos erros de compilação introduzidos pela mudança. A técnica conseguiu identificar vários erros de compilação em sistemas errais. O nosso trabalho usa uma abordagem simples através do uso de LLMs para detectar erros de compilação em sistemas configuráveis. Como trabalhos futuros, pretendemos avaliar não só sistemas reais, mas LLMs com *context window* [8] maiores como o Gemini, para poder avaliar códigos grandes.

Souza e Gheyi [5] investigam a capacidade do ChatGPT3.5, um chatbot de modelo de linguagem em grande escala, de resolver problemas de programação. De um total de 100 problemas submetidos, o modelo de linguagem resolveu corretamente 71 problemas em 3 tentativas, sendo 50 da plataforma LeetCode e 21 da plataforma BeeCrowd. No nosso trabalho, avaliamos 2 LLMs na identificação de erros de compilação em sistemas configuráveis.

6. CONCLUSÕES

Neste trabalho, avaliamos até que ponto LLMs como o ChatGPT 4 e o Le Chat Mistral são capazes de identificar erros de compilação em produtos e em sistemas configuráveis. O ChatGPT 4 identificou 41 de 50 possíveis erros em produtos e 29 de 30 linhas de produto, demonstrando uma alta eficácia na detecção de erros de compilação. Por outro lado, o Le Chat Mistral identificou 28 de 50 erros em produtos e 26 de 30 linhas de produto. Com base nas duas avaliações, os LLMs mostraram bastante potencial em auxiliar desenvolvedores na realização de análises conscientes

de variações em sistemas configuráveis no contexto de erros de compilação.

Como trabalhos futuros, pretendemos avaliar mais exemplos, incluindo sistemas reais. Além disso, pretendemos considerar mais LLMs como o Gemini, Claude, Llama 3, dentre outros. Adicionalmente, pretendemos realizar *fine tuning* de modelos como o Llama 3 para ver até que ponto conseguimos ter uma performance melhor dos modelos na detecção de erros de compilação em sistemas configuráveis. Um outro ponto importante é realizar testes *metamorphic* [10] para diminuir potenciais ameaças à validade. Pretendemos também avaliar outros *prompts* [8, 2], bem como avaliar até que ponto as LLMs conseguem corrigir os erros de compilação em mais profundidade, especialmente em sistemas configuráveis reais. Possivelmente enfrentaremos problemas similares aos já enfrentados anteriormente por outras técnicas analisando sistemas altamente configuráveis [19]. Consideraremos o uso de algoritmos de amostragem nesses cenários no contexto de LLMs.

REFERÊNCIAS

- [1] 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].
- [2] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in Large Language Models. In Advances in Neural Information Processing Systems.
- [3] 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.
- [4] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with Large Language Models: survey, landscape, and vision. doi: 10.48550/arXiv.2307.07221.
- [5] Debora Souza and Rohit Gheyi. 2023. Estudo de caso: uso do ChatGPT para resolução de problemas de programação. In Brazilian Symposium on Software Engineering, CTIC, 80–89.
- [6] 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.
- [7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. Deep Learning. MIT Press.
- [8] Xinyi Hou et al. 2023. Large Language Models for software engineering: A systematic literature review. doi: 10.48550/ARXIV.2308.10620.
- [9] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The Goal Question Metric Approach, 528–532.
- [10] Chen, Tsong Yueh; Kuo, Fei-Ching; Liu, Huai; Poon, Pak-Lok; Towey, Dave; Tse, T. H.; and Zhou, Zhi Quan. "Metamorphic Testing: A Review of Challenges and

Opportunities" (2018). Faculty of Engineering and Information Sciences - Papers: Part B. 975.

- [11] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. In Proceedings of the Programming Language Design and Implementation. 323–334
- [12] Christian Kastner, Paolo 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 Object-Oriented Programming Systems Languages and Applications. 805–824.
- [13] Iago Abal, Jean Melo, Stefan Stănculescu, 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 (2018), 10:1–10:34.
- [14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In Proceedings of the International Conference on Automated Software Engineering. 421–432.
- [15] 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 (2018), 427–450.
- [16] 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.
- [17] Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade, Balduino 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.
- [18] 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 Foundations of Software Engineering. 50–61.
- [19] 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.
- [20] 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.
- [21] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating Preprocessor-Based Syntax Errors. In Proceedings of the International Conference on Generative Programming: Concepts & Experiences. 75–84.
- [22] 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 International Conference on Generative Programming: Concepts & Experiences. 35–44.
- [23] 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.
- [24] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kastner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In Proceedings of the International Conference on Software Engineering. 105–114.
- [25] Ira D. Baxter and Michael Mehlich. 2001. Preprocessor conditional removal by simple partial evaluation. In Proceedings of the Working Conference on Reverse Engineering. IEEE, Germany, 281–290.
- [26] S. Apel, D. Batory, C. Kästner, and G. Saake. Feature-Oriented Software Product Lines. Springer-Verlag, 2013.
- [27] What are large language models (LLMs)?. Disponível em: <<https://www.ibm.com/topics/large-language-models>>. Acesso em: Maio 2024.
- [28] Lucas Albuquerque e Rohit Gheyi. Avaliando a Capacidade de LLMs na Identificação de Erros de Compilação em Linhas de Produto de Software (Dados). 2024. Disponível em: <https://docs.google.com/spreadsheets/d/1e1FhUr3iOcaG4y8mCJC4KuVzYernvfJWzyIHrDN5bk8/edit?usp=sharing>