

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Lucas Marques Silva

ASSISTENTES BASEADOS EM LLMS NO
DESENVOLVIMENTO DE SOFTWARE E NA PROVA DE
TEOREMAS

PORTO ALEGRE

2024

Lucas Marques Silva

ASSISTENTES BASEADOS EM LLMS TEM
DESENVOLVIMENTO DE SOFTWARE E NA PROVA DE
TEOREMAS

**Trabalho de Conclusão de Curso sub-
metido à Universidade Federal do Rio
Grande do Sul, como requisito necessá-
rio para obtenção do grau de Bacharel
em Ciência da Computação**

Porto Alegre, fevereiro de 2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CIÊNCIA DA COMPUTAÇÃO

LUCAS MARQUES SILVA

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em
Ciência de Computação, sendo aprovada em sua forma final pela banca examinadora:

Orientador(a): Prof. Dr. Alvaro Moreira
Universidade Federal do Rio Grande do Sul -
UFRGS

Porto Alegre, 02 de fevereiro de 2024

Agradecimentos

Gostaria de dedicar este espaço para expressar minha sincera gratidão, principalmente à minha mãe, que dedicou sua vida para proporcionar-me uma qualidade de vida e educação que ela não teve a oportunidade de desfrutar quando estava na minha idade.

Além disso, desejo expressar meu profundo reconhecimento a todos os colaboradores da universidade, desde os membros da equipe de limpeza até o reitor, pela oportunidade de completar meus estudos na UFRGS.

Resumo

Este trabalho apresenta uma análise do impacto de assistentes de código baseados em LLM (Large Language Model) no desenvolvimento de software e na provas de teoremas. Além de uma análise de alguns trabalhos relacionados a este campo, o trabalho apresenta um relato da experiência do autor no desenvolvimento de um projeto de software, utilizando assistentes baseados em LLMs. Também fizemos uma análise de como os assistentes de código baseados em IA lidam com assistentes de prova de teoremas como o Lean. No final, é feita uma projeção como será a inserção dessas ferramentas no futuro da criação de software.

Palavras-chave: IA, LLM, Desenvolvimento de Software, Lean

Abstract

This paper presents an analysis of the impact of LLM-based code assistants on software development and theorem proving. In addition to a review of some works related to this field, it is presented an account of the author's experience in developing a software project using LLM-based assistants. We also conducted an analysis of how AI-based code assistants interact with theorem proving assistants such as Lean.

In conclusion, a analisis is made regarding the integration of these tools into the future of software development.

Keywords: AI, Machine Learning, Software Development, Lean

Lista de ilustrações

Figura 1 – Processo de tokenização	22
Figura 2 – Distribuição de tarefas de ES usando assistentes. Fonte: [Hou et al. 2023]	28
Figura 3 – VSCode com destaque para a extensão do Github Copilot	35
Figura 4 – Chat do Github Copilot	36
Figura 5 – Aplicação React	37
Figura 6 – Resposta do assistente Copilot, com as mudanças sugeridas	38
Figura 7 – Resposta do assistente Copilot, com a criação do <code>deleteBookById()</code> destacada em verde	39
Figura 8 – Resposta do assistente Copilot, com as mudanças na manipulação de string	40
Figura 9 – Resposta do assistente Copilot, com a documentação	41
Figura 10 – Resposta do assistente Copilot, com a inclusão do tratamento de erro .	42
Figura 11 – Resposta do assistente Copilot, com a geração dos testes unitários . . .	43
Figura 12 – Error de compilação nos testes unitários gerados pelo Copilot	43
Figura 13 – Código ReactJS	44
Figura 14 – Resposta do assistente Copilot, com a explicação de porque a imagem anterior não é uma boa prática	45
Figura 15 – Resposta do assistente Copilot, com a resposta de qual modo poderíamos seguir de maneira correta a convenção do framework	46
Figura 16 – Resposta do assistente Copilot, com a sugestão dos elementos HTML .	47
Figura 17 – Resposta do assistente Copilot, com a geração de um componente . . .	48
Figura 18 – Resposta do assistente Copilot, com a geração de um Hook customizado	49
Figura 19 – Resposta do assistente Copilot, com a geração de um <code>tuseReducer</code> . . .	50
Figura 20 – Resposta do assistente Copilot, com a sugestão de resolução	57
Figura 21 – Resposta do assistente Copilot, com a sugestão de resolução	57
Figura 22 – Resposta do assistente Copilot, com a sugestão de resolução	58
Figura 23 – A minha resolução para este exercício	59
Figura 24 – Resposta do assistente Copilot, com a sugestão de resolução	59
Figura 25 – Minha resolução para o problema	60
Figura 26 – Resposta sugerida pelo assistente	61
Figura 27 – Resposta desenvolvida por mim	61
Figura 28 – Resposta do assistente Copilot, com a sugestão de resolução	62
Figura 29 – Resposta do assistente Copilot, com a sugestão de resolução usando o 'term-style'	63
Figura 30 – Minha resolução usando Lean 4	63
Figura 31 – Sugestão de solução do Copilot	64

Figura 32 – Sugestão de solução do Copilot	65
Figura 33 – Minha solução em Lean 4	65
Figura 34 – Sugestão de solução do Copilot	66

Lista de tabelas

Tabela 1 – Comparação entre alguns assistentes que utilizam LLM [Pandey 2023]. 26

Lista de abreviaturas e siglas

LLM - Large Language Models

VSCoDe - Visual Studio Code

IA - Inteligência Artificial

GPT-4 - Generative Pre-trained Transformer 4

IDE - Integrated Development Environment

VGC - Video Game Chronicles

IaaS - Infrastructure as a Code

ES - Engenharia de Software

MIT - Massachusetts Institute of Technology

ATP - Automated Theorem Proving

HTTP - Hypertext Transfer Protocol

CRUD - Create Read Update Delete

URL - Uniform Resource Locator

CIC - Calculus of Inductive Constructions

Sumário

1	INTRODUÇÃO	19
2	COMPREENDENDO LLMS E ASSISTENTES DE PROGRAMAÇÃO	21
2.1	Como LLMS funcionam	21
2.2	Assistentes de código baseado em LLMS	22
2.2.1	Visão geral do uso dos assistentes baseados em IA	23
2.3	Discussão sobre como os assistentes baseados em LLMS estão sendo usados no desenvolvimento de software	24
2.3.1	LLMs e Impacto na Programação	24
2.4	Comparação entre alguns assistentes usados na indústria e na academia	25
2.4.1	LLMs no desenvolvimento de software	27
3	EXPERIMENTOS COM ASSISTENTES DE PROGRAMAÇÃO BASEADOS EM LLM	31
3.1	Metodologia	31
3.1.1	Exploração e avaliação do experimento	32
3.2	Funcionamento da aplicação	33
3.2.1	React Hooks e <i>prop system</i>	33
3.2.2	Interação com o Copilot durante o desenvolvimento	34
3.2.3	Como configurar o ReactJS na sua máquina	34
3.3	Resultados do Experimento	37
3.3.1	Complexidade de lógica de programação	37
3.3.2	Manipulação com os métodos de uma classe	38
3.3.3	Manipulação de Strings	39
3.3.4	Criação de Documentação	40
3.3.5	Tratamento de erros	42
3.3.6	Geração de Testes Unitários	42
3.3.7	Análise do comportamento do assistente em conceitos ReactJS	43
3.3.8	Criação de elementos de HTML	47
3.3.9	Criação de um componente	47
3.3.10	Criação de React Hooks	48
3.3.11	Criação de um componente usando <i>useReducer</i>	49
4	USANDO LLM COM ASSISTENTES DE PROVA DE TEOREMAS	51
4.1	Equivalência Curry-Howard	51

4.1.1	Relação entre Tipos e Proposições	51
4.1.2	Relação programas e provas	52
4.2	O que é o Lean	54
4.2.1	Quem criou o Lean?	55
4.3	Usando o Copilot com Lean	55
4.3.1	Como conFigurar o Lean na sua máquina	55
4.4	Metodologia	56
4.5	Testes com Lean e Copilot	56
4.5.1	Proposições e Provas	56
4.5.2	Quantificadores e Igualdade	60
4.5.3	Demonstrações por cálculo	63
4.5.4	Tipos Indutivos	64
4.5.5	Indução e Recursão	65
5	DISCUSSÃO DOS RESULTADOS	67
6	CONCLUSÃO	69
	REFERÊNCIAS	71

1 Introdução

Nos últimos meses fomos inundados com notícias sobre como as LLMs (Large Language Models) estão e irão impactar no nosso trabalho e na nossa sociedade. Como profissional no campo do desenvolvimento de software, e atualmente trabalhando como desenvolvedor React, vi de perto o impacto significativo da Inteligência Artificial (IA) nesse campo. A IA tem causado mudanças profundas no trabalho e revolucionado o processo de desenvolvimento de software nos últimos anos, na área de testes isso também ocorre, pois ao aproveitar algoritmos de aprendizado de máquina, os sistemas de IA podem analisar códigos, identificar possíveis vulnerabilidades e gerar casos de teste direcionados a áreas específicas. [Team]

Ferramentas como Copilot [Copilot], CodeLLama [CodeLLama], e Tabnine [Tabnine] estão se tornando extremamente sofisticadas nas tarefas de analisar códigos, encontrar padrões, compreender contexto, e oferecer ajuda instantânea, portanto elevando a produtividade e qualidade do software. Algoritmos de IA conseguem, por exemplo, antecipar próximos segmentos de código, oferecendo assinaturas de métodos, e identificando potenciais problemas, habilitando desenvolvedores a gerar código mais idiomático, ou seja, conciso, fácil de verificar e fácil de expandir.

Queremos analisar os assistentes de código que utilizam IA, para conferir se realmente eles habilitam os desenvolvedores a se concentrar em resolver problemas de um nível mais complexo por automatizar atividades repetitivas e diminuindo a carga cognitiva no desenvolvedor, aumentando a criatividade e inovação.

Dentro do escopo deste estudo, é fundamental considerar que os experimentos serão conduzidos levando em consideração que o participante, ao se envolver com o assistente, estará iniciando sua jornada de aprendizado no assunto.

Sendo assim, o objetivo deste trabalho é estudar a realidade e as promessas dessa novíssima geração de assistentes procurando (i) entender como os assistentes de codificação baseados em IA têm causado transformações não somente no processo de desenvolvimento de software, como também no âmbito corporativo e educacional, e (ii) entender quais são os riscos envolvidos na utilização deste tipo de ferramenta. Vale ressaltar que nosso objetivo não é “estressar” o assistente a ponto de extrair o máximo de sua capacidade, mas sim, no contexto de aprendizado, analisar suas respostas a solicitações comuns de usuários menos experientes.

Além disso, exploraremos a fascinante conexão entre a computação e a lógica matemática, conhecida como a equivalência de Curry-Howard. Esta relação estabelece um elo entre programas de computador e provas matemáticas, ampliando nossa compreensão sobre como a IA não apenas transforma o desenvolvimento de software, mas também se entrelaça com

fundamentos teóricos da matemática. No contexto deste trabalho, uma atenção especial será dedicada à análise dos assistentes de prova, com foco no Lean [Lean], dentro do cenário de desenvolvimento de software. O Lean, como um assistente de teoremas, desempenha um papel crucial ao proporcionar suporte inteligente na verificação formal de propriedades e na construção de argumentos matemáticos.

O trabalho apresenta inicialmente uma visão geral sobre os assistentes baseados em LLMs e relatos de seus impactos na prática do programador e do matemático. A seguir, o trabalho descreve experimentos práticos que realizei visando obter uma experiência pessoal com tais assistentes e contrastá-la com os impactos relatados na literatura. Esses experimentos práticos por mim realizados consistem em pequenos projetos de desenvolvimento de software. O trabalho está organizado da seguinte forma:

No Capítulo 2, é feita apresentação inicial dos conceitos básicos de LLMs, junto da revisão feita de alguns artigos sobre LLMs no desenvolvimento de programas e uma comparação entre as ferramentas existentes.

Já no Capítulo 3, é realizada uma exploração prática do funcionamento dos assistentes, escrevendo projeto de software em ReactJS.

No Capítulo 4, é feito um estudo de como os assistentes de LLM lidam com assistentes de provadores de teoremas como o Lean.

No Capítulo 5, começamos uma discussão dos resultados do nosso trabalho, abordaremos o que experimentamos com uma visão mais crítica e opinativa.

E por final, no Capítulo 6 terminamos o trabalho com a conclusão.

2 Compreendendo LLMs e Assistentes de Programação

Neste capítulo apresentaremos uma introdução sobre como funciona um LLM, faremos uma passagem sobre alguns aspectos de uso geral dos assistentes e no final faremos uma análise de alguns artigos que tratam do impacto que os assistentes têm no desenvolvimento de software.

2.1 Como LLMs funcionam

No universo dos LLMs, a transição de texto para números desempenha um papel crucial. Este processo, conhecido como tokenização, é essencial para a compreensão de como esses modelos recebem informações e geram respostas textualmente coerentes. Neste contexto, exploraremos detalhadamente o intrincado mecanismo por trás da tokenização, examinando como os LLMs processam dados e antecipam a produção de tokens como resposta.

Os LLMs, especialistas em transformar texto em informações processáveis, recebem textos como entrada e geram textos como saída. No entanto, por serem modelos estatísticos, sua eficiência é notavelmente aprimorada quando lidam com números em vez de sequências de texto. Por esta razão, cada entrada passa por um processo de tokenização, conduzido por um tokenizador antes de ser utilizada pelo núcleo do modelo. O termo "token" refere-se a um trecho de texto composto por um número variável de caracteres. A missão primordial do tokenizador é fragmentar a entrada em conjuntos de arrays de inteiros. Cada token, por sua vez, é mapeado por um indexador, um inteiro codificado que representa aquele extrato específico do texto original.

Na Figura 1, apresentamos uma visualização esclarecedora do processo de tokenização, fornecendo um guia visual para compreender essa etapa vital no ciclo de funcionamento dos LLMs. [Castelluccio 2023]

Ao receber n tokens como entrada, o modelo é capaz de prever um token como saída. Essa previsão é fundamentalmente guiada pelo conceito de "janela". Nesse contexto, a janela refere-se à extensão da sequência de tokens considerada pelo modelo durante a previsão. Em outras palavras, a janela define a amplitude da vizinhança de tokens que influencia a escolha do próximo token. Ao incorporar o token previsto como entrada na próxima iteração, a janela se expande, permitindo que o modelo leve em conta contextos mais amplos e, assim, aprimore a experiência do usuário ao capturar sentenças mais extensas ou múltiplas como resposta. Essa dinâmica da janela desempenha um papel



Figura 1 – Processo de tokenização

crucial na capacidade do modelo de compreender e gerar sequências de texto coesas e relevantes.

O token de saída é escolhido pelo modelo conforme a probabilidade de ocorrer após a sequência de texto atual. Isso acontece porque o modelo prediz a distribuição sobre todos os possíveis tokens, calculados baseado nos dados usados para treinar o modelo. Contudo, nem sempre o token de maior probabilidade é escolhido, pois existe um grau de aleatoriedade nesta escolha. Desta forma, o modelo se comporta de uma maneira não determinística, assim saídas diferentes podem ser produzidas para uma mesma entrada. Esse grau de aleatoriedade é adicionado para simular o processo criativo e pode ser modificado usando um parâmetro chamado temperatura.

Esses modelos têm como base a arquitetura Transformer [What Is the Transformer Architecture and How Does It Work?], que é capaz de processar sequências de texto em paralelo. Essa estrutura possibilita ao modelo compreender o contexto e as relações entre os dados de maneira mais eficaz do que os algoritmos utilizados anteriormente.

2.2 Assistentes de código baseado em LLMs

Um assistente de codificação baseado em IA é uma ferramenta que emprega a técnica de machine learning para auxiliar programadores em suas tarefas de codificação, assemelhando-se a uma entidade humana treinada para colaborar em suas atividades. Assistentes de programação com inteligência artificial adquirem conhecimento a partir de uma vasta quantidade de repositórios de códigos distribuídos e acessíveis online [Team 2023]. A nova geração de assistentes de programação baseada em IA são construídos com base nos LLMs (ou Large Language Models), como o GPT-4 desenvolvido pela OpenAI [ChatGPT]. A grande maioria dos assistentes de códigos de IA utilizam técnicas de processamento de linguagem natural para fazer um scanning do que está sendo escrito no código e a partir disso sugerir mudanças ou correções no código do desenvolvedor, utilizando identificação de padrões e estruturas no código e comparando com o que já existe nos dados que foram treinados. Duas das funções mais básicas que os assistentes

implementam é o *autocomplete* e predição de código, esse recurso permite que os assistentes antecipem o que seria escrito e forneça sugestões de código. Outra função importante é o de refatoração de código, os assistentes sugerem modificações na escrita do código, com a intenção de melhorar aspectos como a manutenibilidade do código. Além disso, os assistentes também desempenham um papel na otimização do código, e correção de erros sintáticos e semânticos. [Kerner 2023]

2.2.1 Visão geral do uso dos assistentes baseados em IA

É muito comum em empresas a utilização de pair programming [Pair programming] para aumentar produtividade e aumentar o conhecimento sobre a base de código entre os desenvolvedores, mas alguns desenvolvedores acreditam que trabalhar de um modo mais independente faz com que eles produzam mais. Com o progresso das inteligências artificiais nos auxiliares de programação, tornou-se evidente que as tarefas do desenvolvedor podem ser complementadas pelas assistentes de códigos em IA na geração de código. [Jadav 2023]

Os programadores têm agora a capacidade de empregar a inteligência artificial para criar sugestões de arquitetura de software. Esta constitui mais uma forma pela qual os desenvolvedores podem otimizar seu tempo, deixando de lado tarefas rotineiras e direcionando seus esforços para a implementação de novas funcionalidades em seus projetos. [Christopher 2023]

Devido ao fato de esses auxiliares de código terem acesso a uma infinidade de repositórios contendo códigos open source, isso resulta na geração de código semelhante ao que foi previamente escrito e testado em diversos contextos. Essa modalidade de geração de código pode ser empregada para acelerar o processo de desenvolvimento e mitigar erros humanos. [Christopher 2023] A redação de testes unitários configura-se como uma atividade desafiadora, demandando investimento significativo de tempo e esforço para cobrir uma extensa gama de combinações de testes [MAITRA 2023].

No contexto das LLMs para a produção de teste unitário, frequentemente, o tempo dedicado a tarefas mais monótonas, como a elaboração de testes, poderia ser mais eficientemente utilizado na criação de novos trechos de código.

A inteligência artificial possui a aptidão para gerar documentação de maneira similar aos seres humanos. Ela é capaz de identificar padrões e estabelecer conexões em extensas bases de dados, uma tarefa que possivelmente seria mais desafiadora para os humanos realizar de forma autônoma. Uma funcionalidade presente no Github Copilot consiste em criar documentação para as variáveis, funções e métodos de uma classe, além de exibir essa documentação diretamente na própria IDE. Isso se torna valioso quando buscamos compreender a razão de existir da classe, sua finalidade, como suas operações são executadas e demais informações relacionadas. [Christopher 2023]

2.3 Discussão sobre como os assistentes baseados em LLMs estão sendo usados no desenvolvimento de software

Nos últimos anos diversos estudos têm sido realizados a respeito das transformações no processo de desenvolvimento de software causadas pela adoção de assistentes de programação baseadas em IA.

A maioria dos especialistas acreditam que ferramentas como Copilot vão impactar na velocidade com que o código é escrito e, principalmente vão ajudar desenvolvedores menos experientes a escrever mais código.

De acordo com um estudo feito pelo Github, desenvolvedora do assistente Copilot, cerca de 30% das sugestões dadas pelo assistente são aceitas pelos desenvolvedores [Raj 2023]. Não se sabe porém, quais sugestões são essas e como elas se diferenciam de um autocomplete que não utiliza IA. Existem ainda preocupações em relação a essas sugestões pois como a base de dados do assistente é basicamente qualquer código disponível na Internet, não se pode garantir que a sugestão da ferramenta é livre de bugs ou não esteja inserindo brechas de segurança no código.

2.3.1 LLMs e Impacto na Programação

Citando um artigo da revista Technology Review [Heaven 2023], um professor da universidade de Duke, trocou a linguagem que seria usada nas aulas, ao invés de Python que é uma linguagem mais simples, seria usado Rust uma linguagem mais desafiadora principalmente para desenvolvedores menos experientes, mas os alunos poderiam usar o Copilot nas tarefas pedidas nas aulas. O Professor relata que com o apoio do Copilot os alunos conseguiram criar aplicações web completas, “Rust tem a reputação de ser uma linguagem muito difícil”, diz ele. “Mas fiquei agradavelmente chocado com o desempenho dos alunos e com os projetos que construíram – como eram complexos e úteis.”

Outro artigo publicado pela VGC [Scullion 2024], foi mencionado que cerca de 50% dos profissionais na indústria de jogos já experimentaram o uso de ferramentas de IA generativa, seja diretamente ou através de colegas em suas equipes. Essa pesquisa contou com mais 3000 participantes de estúdios indies até estúdios AAA¹, e 31% dos respondentes já utilizaram essas ferramentas, enquanto que 18% disseram que não utilizaram mas colegas de trabalho já o fizeram. Há uma pressão significativa pelo uso de IA generativa em vários pontos da indústria criativa, incluindo a indústria dos videos games.

¹ Estúdios indie são empreendimentos independentes de menor escala, muitas vezes criados por pequenas equipes, focados na inovação e originalidade, enquanto estúdios AAA representam grandes empresas do setor de jogos, investindo consideráveis recursos para produzir títulos de grande orçamento, alta qualidade e ampla distribuição

2.4 Comparação entre alguns assistentes usados na indústria e na academia

Na Tabela 1 [Pandey 2023], temos uma breve comparação entre três assistentes de código utilizados na indústria hoje.

GitHub Copilot, desenvolvido pelo GitHub em colaboração com OpenAI, é uma ferramenta de *code complete* com tecnologia de IA projetada para ajudar os desenvolvedores a escrever código de forma mais rápida e eficiente. Ele é baseado no modelo de linguagem GPT e fornece sugestões em tempo real, preenchimentos automáticos e geração de código com reconhecimento de contexto. [Pandey 2023]

Inclusive, nos experimentos que serão detalhados nos capítulos subsequentes, faremos uso deste assistente de programação. Sua configuração revela-se relativamente simples em diversas IDEs, e eu tenho acesso à licença gratuita através do meu atual emprego.

AWS CodeWhisperer, desenvolvido pela Amazon Web Services (AWS), é uma ferramenta baseada em IA que visa aprimorar o processo de desenvolvimento de aplicativos especificamente para o ambiente de nuvem AWS. Ele auxilia os desenvolvedores a escrever código específico da AWS, otimizando o desempenho e garantindo as melhores práticas. [Pandey 2023]

O Code Llama é um modelo de IA desenvolvido com base no Llama 2 [Llama 2], ajustado para gerar e revisar código, ele é gratuito para pesquisa e uso comercial. [CodeLLama Llama 2]

Fonte: GitHub Copilot vs. AWS CodeWhisperer: A Comparative Analysis

Tabela 1 – Comparação entre alguns assistentes que utilizam LLM [Pandey 2023]

Aspecto	Github Copilot	AWS CodeWhisperer	CodeLLama
Foco Principal	Assistência na geração de código e capacidade de completar código.	Desenvolvimento na AWS Cloud e otimizações dentro desse ambiente de desenvolvimento	Desenvolvimento e geração de código e linguagem natural.
Audiência	Desenvolvedores	Desenvolvedores criando no ambiente AWS	Usuário interessados em uma LLM open source, de graça e bastante capaz.
Fonte dos Dados para o modelo	Aprende através de uma variedade de repositórios na internet	Sugestões baseadas na melhores práticas AWS	Treinado em repositórios específicos de código.
Objetivo de existência do assistente	Auxilia na geração de código e dar sugestões	Ênfase nas otimizações específicas do ecossistema AWS	O objetivo é tornar o fluxo de trabalho do desenvolvedor mais eficiente e menos repetitivo.
Funcionalidades específicas para segurança	Limitado conhecimento nas melhores práticas de segurança	Tem scans de segurança embutidos no modelo.	Não tem conhecimento específico sobre diretrizes de segurança.
Foco em IaaS (Infrastructure as a Code)	Não foca em IaaS	Ajuda com templates de IaaS	Não tem foco em IaaS
Utilidade	Útil para vários projetos	Mais benéfico para o desenvolvimento na AWS	Útil para vários projetos
Limitações	Pode faltar o contexto para que consiga fazer as melhores sugestões	Focado no contexto AWS	Não conseguir pegar o contexto pode limitar sua capacidade de asserção.
Preço	\$10/mês ou \$100/ano para indivíduos. \$19/mês/usuário para empresas	De graça para o uso individual. \$19/user/mês para empresas	De graça.
Linguagens Suportadas	C, C++, C#, Go, Java, Javascript, PHP, Python, Ruby, Scala e Typescript	Python, Java, Javascript, TypeScript, C#, Rust, Go, Ruby, Scala, Kotlin, C, C++, PHP, Shell, SQL.	Python, C++, Java, PHP, Typescript (Javascript), C#, Bash e outras.
Suporte de IDEs	Intellij IDEA, Android Studio, AppCode, VS-Code e outras	Intellij IDEA, VSCode, AWS Cloud9, AWS Lambda console, JupyterLab e Amazon SageMaker Studio	Não tem suporte nativo para nenhuma IDE.

2.4.1 LLMs no desenvolvimento de software

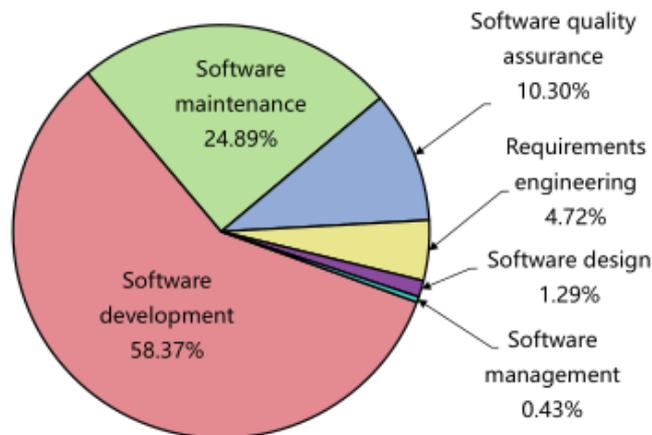
Este capítulo tem por objetivo fazer uma revisão sobre alguns artigos sobre os assistentes baseados em LLM no desenvolvimento de software e na ajuda para criar provas usando assistentes de provas de teoremas, também queremos saber como tem sido o processo de aceitação deles.

O primeiro artigo que analisaremos [Hou et al. 2023], foi publicado em Setembro de 2023 com o título de “*Large Language Models for Software Engineering: A Systematic Literature Review*”. Dentre os objetivos do artigo estão: quais casos de usos usando LLM tem tido sucesso na engenharia de software e quais modelos baseados em LLM têm sido empregados para resolução de problema em ES. Este paper analisou 229 papers de pesquisa durante 2017 à 2023. Na introdução do artigo os autores citam que os modelos baseados em LLM tem tido um impacto significativo em vários domínios inclusive o da Engenharia de Software. A utilização de LLM na ES emerge principalmente do fato de que inúmeros desafios dessa área podem ser transformados em dados, código e análise de texto. A aplicabilidade das LLM tem sido proeminente nas tarefas de sumarização de código, sendo esta uma tarefa que envolve produção de uma explicação sobre o código em linguagem natural, a geração de código e artefatos de códigos como anotações, comuns em alguns frameworks como Spring Boot.²

Engenharia de Prompt, caracterizado pelo design especializado de prompts, tem se tornado uma técnica fundamental para melhorar as interações com os modelos, esses prompts personalizados tem dois propósitos: direcionar os modelos a um tipo específico de resposta e como uma interface para acessar o conhecimento extensivo dos modelos. Pesquisas recentes ressaltam a importância do papel da engenharia de prompt em melhorar a performance das LLM para tarefas específicas de ES, portanto contribuindo com a evolução de metodologias de desenvolvimento de automação de software.

Ainda de acordo com o artigo, a distribuição de usos de LLM nas atividades de ES é a seguinte: 58.37% fica por conta de desenvolvimento de software, manutenção de software 24.89%, atividades relacionadas à qualidade de software têm 10.30% e engenharia de requisitos tem 4.72%, outras atividades completam os cem por cento. O gráfico abaixo ilustra esses números.

² O Spring Boot é um framework que simplifica e acelera o desenvolvimento de aplicativos Java, oferecendo configurações padrão e facilitando a criação de aplicativos robustos baseados na plataforma Spring.



(a) Distribution of LLM usages in SE activities.

Figura 2 – Distribuição de tarefas de ES usando assistentes. Fonte: [Hou et al. 2023]

O artigo [Peng et al. 2023] descreve um experimento usando o Github Copilot, usando 95 desenvolvedores com idades entre 25 à 34 anos a maioria deles tem graduação e uma média de 6 anos de experiência. Nele é pedido que os participantes desenvolvam uma aplicação em JavaScript o mais rápido possível. Os desenvolvedores com acesso ao Copilot terminaram a tarefa 55.8% mais rápido em relação ao grupo controle.

O grupo controle tinha acesso a outras ferramentas como a internet para fazer consultas, então o uso de fóruns como o StackOverflow eram permitidos. Desenvolvedores com menos experiência, programadores mais velhos e aqueles que programam mais do que uma hora por dia se beneficiaram do uso do assistente Copilot. Foram utilizadas duas métricas de performance de cada grupo, uma delas é o sucesso da tarefa, ou seja, se os usuários conseguiram terminar a atividade e a segunda métrica foi o tempo de desenvolvimento. Os resultados sugerem que o Copilot aumenta a produtividade significativamente na população experimentada.

Em [Sandoval et al. 2023], os autores buscam entender o impacto que o assistentes têm na introdução de falhas de cibersegurança nos códigos. Foi pedido para que 58 estudantes de programação escrevessem programas que continham estruturas de dados em C sendo assistido pelos modelos, no final do artigo a conclusão foi de não houve uma introdução maior do que o normal de bugs.

As preocupações sobre a utilização de assistentes de LLM, vêm do fato de que os dados com os quais o modelo foi treinado continham código com falhas de segurança, já que a base de dados dos modelos é basicamente qualquer código na internet. Além disso código que pode ser considerado seguro em um contexto, não necessariamente é seguro em outro contexto. Foi mostrado também que 63% dos bugs eram originados dos códigos escritos

por humanos, enquanto que 36% eram de códigos sugeridos pelos assistentes.

O artigo LeanDojo chamado de “*LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*” [Yang et al. 2023], introduz o LeanDojo que é um assistente LLM usado para escrever automação de provas de teoremas. A relação entre raciocínio e automação de provas de teoremas reside no fato de que as ATP (automated theorem proving) são destinadas a emular certos aspectos do raciocínio humano em um modo formal e algorítmico, os objetivos das ATPs são gerar automaticamente provas para teoremas expressados em lógica formal.

O paper apresenta no final o primeiro conjunto de provadores de teoremas baseados em Modelos de Linguagem por Aprendizado Profundo de código aberto, sem o uso de conjuntos de dados proprietários, e o disponibilizando sob uma licença MIT permissiva para facilitar pesquisas adicionais.

3 Experimentos com assistentes de programação baseados em LLM

Neste capítulo descrevemos a metodologia utilizada na criação de uma aplicação usando os assistentes LLM e como será feita a análise dos resultados. Apresentamos também o experimento propriamente dito e analisamos os resultados.

3.1 Metodologia

Na condução deste experimento, adotamos a seguinte abordagem metodológica para avaliar o desempenho e a utilidade dos assistentes de código no processo de desenvolvimento de aplicações utilizando a biblioteca React. A metodologia empregada é delineada pelos seguintes passos.

Começando pela definição do escopo, neste trabalho iremos desenvolver uma aplicação usando React para poder medir o quanto os assistentes conseguem nos ajudar no processo de desenvolvimento, tanto na velocidade em que o processo é feito tanto como em aspectos subjetivos, como satisfação e aceitação do que vai ser desenvolvido, essa aplicação foi criada se baseando em um curso que eu fiz chamado Modern React with Redux através da plataforma da Udemey.

Quanto à identificação das ferramentas, neste projeto, empregamos o editor de código Visual Studio Code (VSCode), o assistente GitHub Copilot e o framework ReactJS.

Analisaremos a qualidade por meio dos trechos de código e dos testes unitários disponibilizados pelos assistentes. O projeto será de escala reduzida para facilitar uma avaliação mais precisa.

Na etapa de coleta de dados, os elementos fundamentais consistem nas respostas fornecidas pelo assistente durante a fase de desenvolvimento. Estas respostas, que abrangem desde sugestões de código até esclarecimentos sobre conceitos específicos, compõem a base informacional essencial para a avaliação do desempenho do assistente. A meticulosidade na documentação e análise desses dados será crucial para extrair insights relevantes sobre a eficácia e a capacidade do assistente ao longo do processo experimental.

Sobre as métricas de avaliação que usamos neste trabalho, fizemos uma avaliação qualitativa das respostas do assistente, considerando respostas como Ruim, Média e Boa.

Ao término da experimentação, planejamos dedicar um capítulo à análise dos resultados. Ao concluir o experimento, conduziremos uma discussão sobre as respostas

geradas pelo assistente. O foco será avaliar a eficácia da experiência de aprendizado de um novo framework com o suporte fornecido pelo assistente.

3.1.1 Exploração e avaliação do experimento

Durante a fase de desenvolvimento da aplicação utilizando o framework React, optamos por incorporar o assistente de código Github Copilot como parte integrante do processo. A proposta central do experimento foi explorar a capacidade do Copilot em auxiliar em tarefas rotineiras no contexto de um projeto de software. Para tanto, ao longo do experimento, solicitamos a intervenção do assistente em diversas atividades, abrangendo desde o desenvolvimento de novas funcionalidades até a criação de testes unitários e a refatoração de código.

Um aspecto fundamental do experimento consistiu na observação cuidadosa das respostas geradas pelo Github Copilot ao longo de cada tarefa realizada. Ao finalizar cada atividade, conduzimos uma análise qualitativa rigorosa, avaliando a eficácia e pertinência de cada resposta obtida. A análise qualitativa não se limitou apenas à exatidão técnica, mas também considerou aspectos como a clareza do código gerado, a aderência às boas práticas de programação e a capacidade de compreender as nuances específicas do contexto do projeto.

Esse processo permitiu uma compreensão mais profunda das contribuições do Github Copilot no contexto do desenvolvimento da aplicação React, delineando tanto os pontos fortes quanto as possíveis limitações do assistente. O objetivo final é proporcionar uma avaliação criteriosa das capacidades do Copilot em facilitar e aprimorar o fluxo de trabalho de desenvolvedores em um projeto real, contribuindo assim para uma compreensão mais abrangente do papel dessas ferramentas no desenvolvimento de software atual.

Na seção de resultados do experimento, adicionaremos comentários abaixo das imagens, apresentando nossas impressões e justificando nossas análises para cada requisição ao assistente.

Para reprodução dos experimentos conduzidos ao longo deste trabalho, compartilharei as configurações específicas da minha máquina:

Sistema Operacional: Fedora Linux 37 (Workstation Edition) x86 64

Host: 81S9 Lenovo IdeaPad S145-15IWL

Kernel: 6.5.12-100.fc37.x86 64

3.2 Funcionamento da aplicação

Na aplicação a ser desenvolvida, propomos um sistema que permite ao usuário adicionar, editar ou remover livros de uma lista. Cada interação com a aplicação será suportada por um "banco de dados", representado, neste caso, por um arquivo .json, no qual os registros serão armazenados. Este projeto utilizará recursos avançados do framework React, como React Hooks e o uso de contexto para a compartilhamento de funções entre componentes. A ideia principal é investigar como assistentes, baseados em dados extensos de código React disponíveis na internet, lidam com a complexidade desse tipo de software.

3.2.1 React Hooks e *prop system*

Os React Hooks são recursos essenciais no desenvolvimento de componentes funcionais no React, proporcionando uma maneira eficiente de gerenciar o estado e o ciclo de vida do componente. O *useEffect* é um Hook utilizado para realizar efeitos colaterais em componentes funcionais, ou seja, ações secundárias que ocorrem além da renderização principal do componente. Ele é frequentemente empregado para lidar com operações assíncronas, requisições a APIs, ou para gerenciar a subscrição e desinscrição de eventos.

Os Hooks são os elementos que conferem funcionalidades aos componentes no ReactJS. À medida que a complexidade dos Hooks aumenta, é comum extrair o código do Hook do próprio componente.

Por sua vez, o *useReducer* é outro Hook fundamental que oferece uma abordagem mais poderosa para lidar com o estado complexo de um componente. Ao contrário do *useState*, que é mais adequado para estados simples, o *useReducer* é preferível quando o estado do componente envolve lógicas mais avançadas e interações complexas. Ele é particularmente útil para gerenciar estados que dependem do estado anterior.

O sistema de propriedades, ou "prop system", no React, é uma abordagem fundamental para a passagem de dados entre componentes. As *props* (abreviação de propriedades) permitem que informações sejam transmitidas de um componente pai para seus componentes filhos, estabelecendo uma comunicação eficaz entre eles. As *props* são essenciais para criar componentes reutilizáveis e modulares, pois possibilitam a configuração dinâmica de um componente com base nos dados fornecidos pelo componente pai. Além disso, o sistema de props contribui para a manutenção de um estado centralizado e coeso, melhorando a legibilidade e a estrutura do código React.

3.2.2 Interação com o Copilot durante o desenvolvimento

Nesta etapa do projeto, a interação com o Copilot será central para a criação e aprimoramento de componentes. Inicialmente, desenvolverei alguns componentes no contexto do projeto. Em seguida, solicitei a intervenção do Copilot para estender funcionalidades existentes ou criar novos componentes com base no código já presente. Caso a resolução não atenda às expectativas, farei solicitações adicionais ao assistente, iterando até obter resultados considerados satisfatórios para dar continuidade ao projeto.

Os exemplos iniciais enfatizarão a habilidade do assistente em gerar código e sugestões a partir de um código fonte preexistente. Em seguida, a análise será direcionada para avaliar a capacidade do Copilot em criar código com base em entradas de texto, explorando assim a versatilidade do assistente em diversas abordagens durante o desenvolvimento.

Vale ressaltar que, devido à natureza do assistente, nem todos os experimentos são passíveis de reprodução exata. A "criatividade" inerente ao processo de geração de código pelo Copilot pode resultar em respostas variadas mesmo diante de solicitações idênticas. Portanto, ao longo deste estudo, será levado em consideração esse elemento dinâmico do assistente, reconhecendo que a diversidade nas respostas pode contribuir para uma compreensão mais completa de suas capacidades. Essa variabilidade será contemplada nas análises e avaliações, destacando a capacidade única do Copilot em oferecer soluções adaptáveis e, por vezes, inesperadas, no contexto do desenvolvimento de software.

3.2.3 Como configurar o ReactJS na sua máquina

No contexto do desenvolvimento de aplicações com ReactJS no Visual Studio Code (VSCode) e a utilização do Copilot, são requisitos essenciais a instalação do Node.js e npm, fundamentais para a execução do React, e a configuração da IDE VSCode, que desfruta de ampla popularidade no desenvolvimento web. Além disso, é crucial incorporar extensões relevantes como "ESLint" e "Prettier" para garantir a consistência e formatação do código.

A opção pelo uso do Create React App para inicializar novos projetos e a consideração do Git para controle de versão são práticas sugeridas. No âmbito opcional, a ativação do Copilot requer a assinatura do GitHub Copilot, proporcionando sugestões contextuais e agilizando o processo de desenvolvimento. Ao atender a esses requisitos, estabelece-se um ambiente propício para explorar efetivamente o potencial do ReactJS no VSCode.

Após a instalação bem-sucedida da extensão GitHub Copilot no VSCode, um indicativo visual é a presença de um ícone de chat destacado na barra lateral, conforme ilustrado na Figura 3. Ao clicar nesse ícone, iniciamos a interação com o assistente, como evidenciado na Figura 4, onde é possível enviar nossos requests ao assistente.

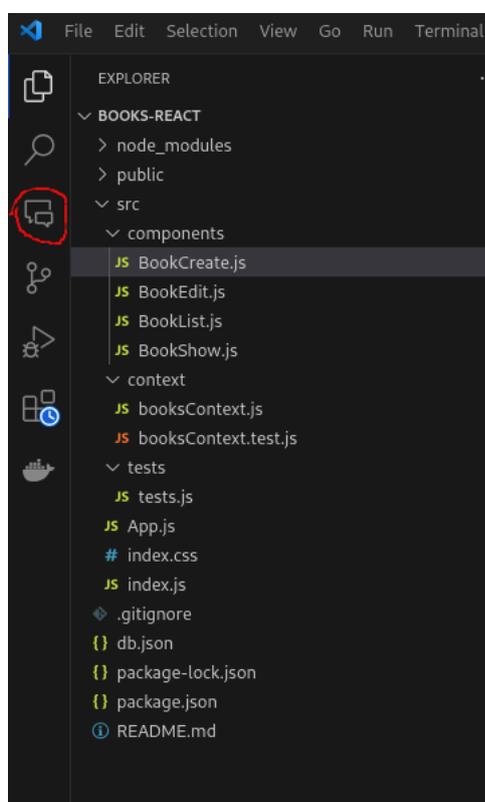


Figura 3 – VSCode com destaque para a extensão do Github Copilot

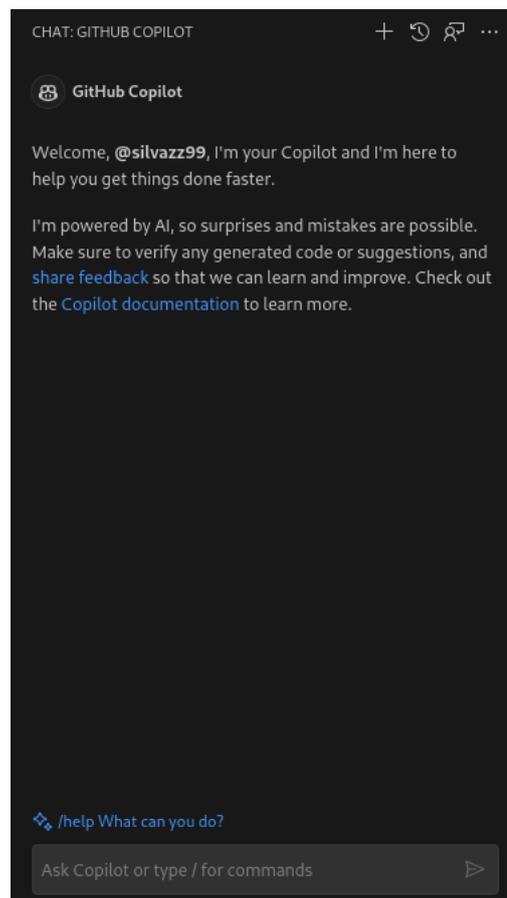


Figura 4 – Chat do Github Copilot

3.3 Resultados do Experimento

Nas subseções abaixo iremos apresentar os resultados dos experimentos envolvendo a nossa aplicação e o Copilot.

Abaixo temos uma imagem da aplicação.

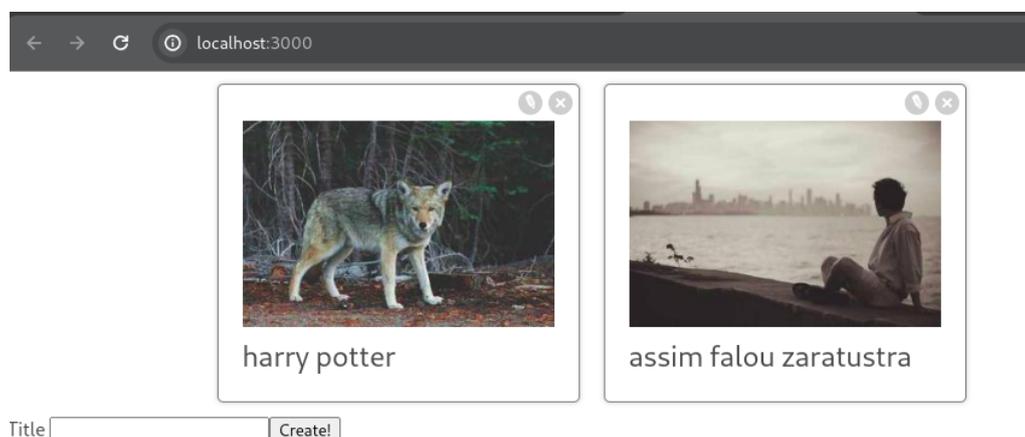


Figura 5 – Aplicação React

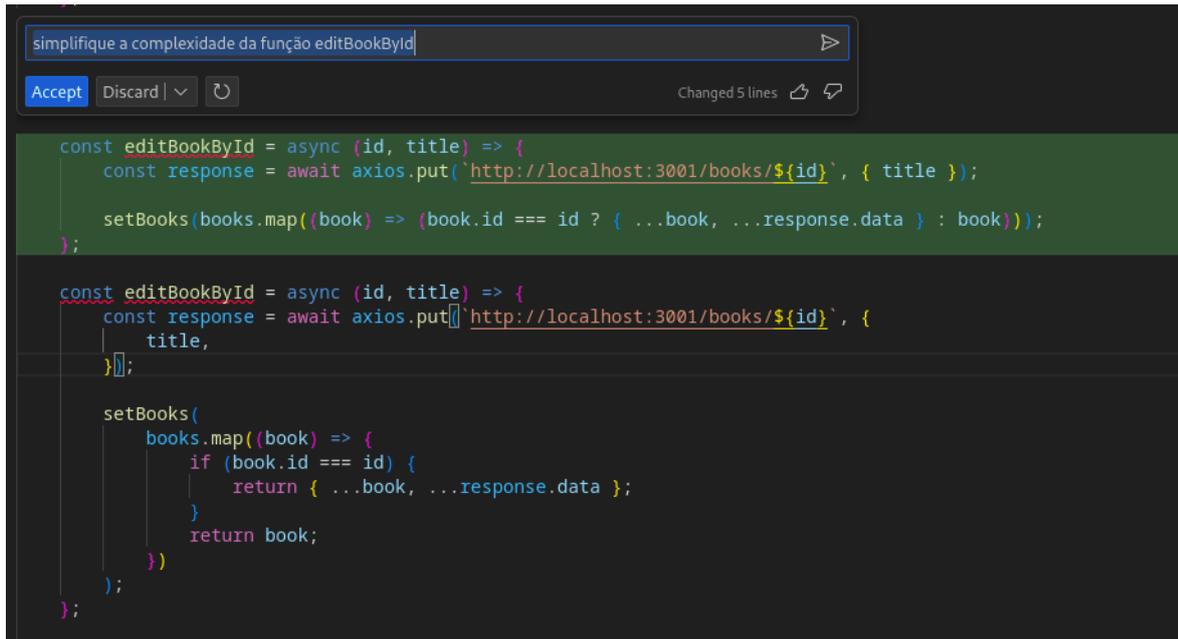
3.3.1 Complexidade de lógica de programação

Neste primeiro segmento, buscamos examinar a capacidade do assistente em lidar com a complexidade do código e compreender a intenção por trás de determinado método. A abordagem adotada foi solicitar ao assistente a simplificação de um método específico. No cenário proposto, o método envolve uma chamada externa utilizando o framework Axios (o Axios facilita requisições HTTP, simplificando a comunicação do cliente com servidores), seguida pela manipulação da resposta para atualizar a lista de livros na aplicação e invocar o método de alteração de estado do framework React, denominado neste caso como `setBooks()`.

Notavelmente, o assistente foi capaz de simplificar a lógica, mesmo quando a solução original já apresentava uma simplicidade intrínseca.

A análise das próximas interações serão complementadas pelas representações visuais associadas, a qual fornece uma visão mais abrangente do contexto e das respostas geradas pelo assistente. A Figura 6, mostra o resultado.

Vamos concentrar nossa análise nas seções destacadas em verde, as quais foram sugestões providenciadas pelo assistente.



```
simplifique a complexidade da função editBookById

Accept Discard Changed 5 lines

const editBookById = async (id, title) => {
  const response = await axios.put(`http://localhost:3001/books/${id}`, { title });

  setBooks(books.map((book) => (book.id === id ? { ...book, ...response.data } : book)));
};

const editBookById = async (id, title) => {
  const response = await axios.put([`http://localhost:3001/books/${id}`, {
    title,
  }]);

  setBooks(
    books.map((book) => {
      if (book.id === id) {
        return { ...book, ...response.data };
      }
      return book;
    })
  );
};
```

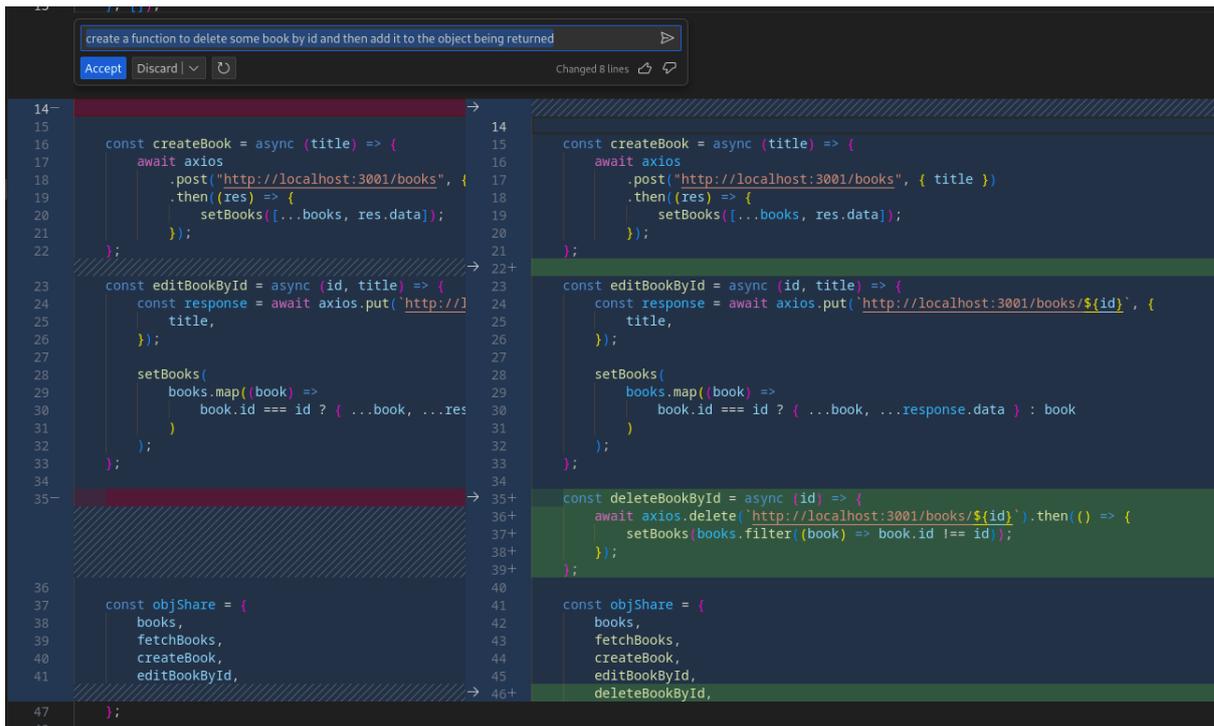
Figura 6 – Resposta do assistente Copilot, com as mudanças sugeridas

Avaliação: Boa. A resolução do assistente assemelha-se à minha abordagem. No entanto, o Copilot substituiu as condicionais if/else por métodos ternários, tornando o código mais legível. É importante observar que o exemplo apresentado é notavelmente simples.

3.3.2 Manipulação com os métodos de uma classe

Dado os métodos já incorporados na aplicação de livros, primordialmente um CRUD, solicitamos ao assistente a geração de um novo método específico para o sistema, desta vez, focando na implementação de um método de exclusão.

Especificamente, requisitamos a criação de um método de deleção, o qual recebe o id do elemento a ser removido. O resultado desta solicitação é ilustrado na Figura 7.



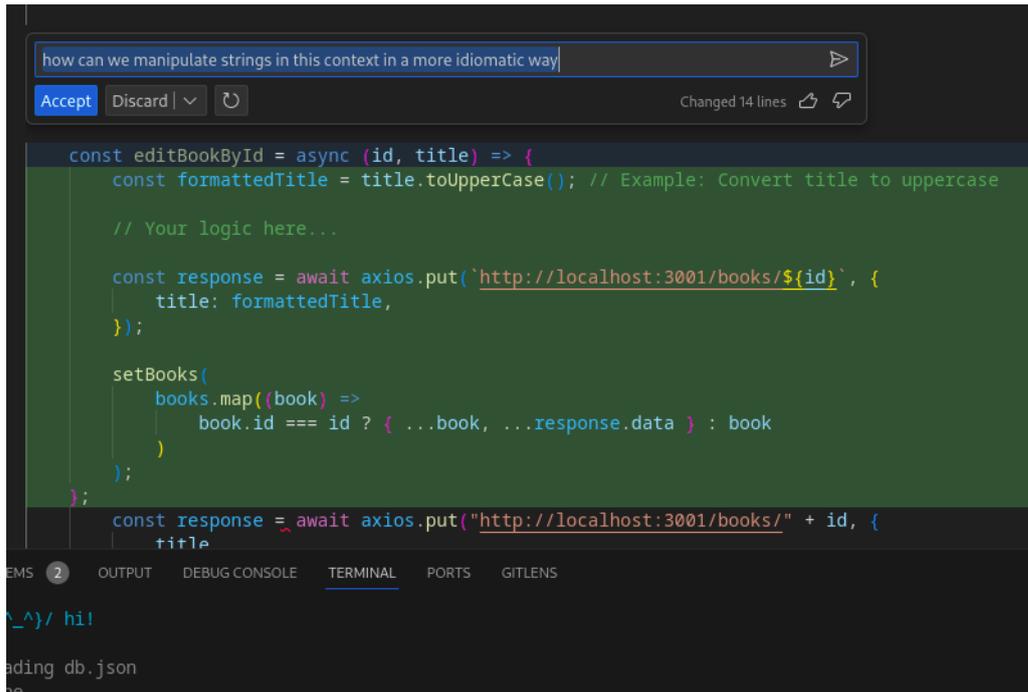
```
14- 14-
15- 15-
16- 16- const createBook = async (title) => {
17- 17-   await axios
18- 18-     .post("http://localhost:3001/books", {
19- 19-       .then(res) => {
20- 20-         setBooks([...books, res.data]);
21- 21-       });
22- 22-   };
23- 23-
24- 24- const editBookById = async (id, title) => {
25- 25-   const response = await axios.put('http://
26- 26-     title,
27- 27-   );
28- 28-   setBooks(
29- 29-     books.map((book) =>
30- 30-       book.id === id ? { ...book, ...res
31- 31-     });
32- 32-   );
33- 33- };
34- 34-
35- 35+
36- 36+ const deleteBookById = async (id) => {
37- 37+   await axios.delete('http://localhost:3001/books/${id}').then(() => {
38- 38+     setBooks(books.filter(book => book.id !== id));
39- 39+   });
40- 40- };
41- 41-
42- 42- const objShare = {
43- 43-   books,
44- 44-   fetchBooks,
45- 45-   createBook,
46- 46+   editBookById,
47- 47-   deleteBookById,
48- 48- };
49- 49-
```

Figura 7 – Resposta do assistente Copilot, com a criação do *deleteBookById()* destacada em verde

Avaliação: Boa. O método criado pelo Copilot é satisfatório, vale considerar que o assistente já utiliza como dados o que já tem no arquivo e baseando-se nisso, me dá uma sugestão parecida.

3.3.3 Manipulação de Strings

Solicitamos ao assistente uma abordagem mais idiomática - concisa, fácil de verificar e fácil de expandir -, na manipulação de strings. O método anterior não empregava interpolação de strings, prática comum para casos que envolvem links de URL com parâmetros. A Figura 8 traz o resultado.



```
how can we manipulate strings in this context in a more idiomatic way
```

```
Accept Discard [refresh] Changed 14 lines
```

```
const editBookById = async (id, title) => {  
  const formattedTitle = title.toUpperCase(); // Example: Convert title to uppercase  
  
  // Your logic here...  
  
  const response = await axios.put(`http://localhost:3001/books/${id}`, {  
    title: formattedTitle,  
  });  
  
  setBooks(  
    books.map((book) =>  
      book.id === id ? { ...book, ...response.data } : book  
    )  
  );  
};  
  
const response = await axios.put("http://localhost:3001/books/" + id, {  
  title
```

```
TERMINAL
```

```
^_^) / hi!  
loading db.json
```

Figura 8 – Resposta do assistente Copilot, com as mudanças na manipulação de string

Avaliação: Média. Ao solicitar ao assistente uma manipulação mais convencional das strings, ele criou uma nova variável usando `toUpperCase()`. Preferíamos que ele não tivesse introduzido outra variável, podendo utilizar a funcionalidade do JavaScript que permite acessar o campo da variável no objeto sem repetir o nome da variável (*shorthand property notation*). No final acredito que o assistente criou código desnecessário e aumentou a complexidade do método.

3.3.4 Criação de Documentação

Pedimos para que o assistente gerasse documentação sobre os métodos e classes do nosso componente React. Nas imagens nas quais não tem o request solicitado ao assistente, colocarei aqui no texto da subseção o texto de input do request:

Request: "Create documentation for all methods, parameters and this file class". A Figura 9 traz o resultado.

```
1  /**
2   * @file Provides the BooksContext and Provider components for managing books data.
3   * @module booksContext
4   */
5
6  import { createContext, useState, useCallback } from "react";
7  import axios from "axios";
8
9  /**
10   * Context object for managing books data.
11   * @type {object}
12   * @property {Array} books - The array of books.
13   * @property {Function} fetchBooks - Function to fetch books data from the server.
14   * @property {Function} createBook - Function to create a new book.
15   * @property {Function} editBookById - Function to edit a book by its ID.
16   * @property {Function} deleteBookById - Function to delete a book by its ID.
17   */
18  const BooksContext = createContext();
19
20  /**
21   * Provider component for managing books data.
22   * @param {object} props - The component props.
23   * @param {ReactNode} props.children - The child components.
24   * @returns {ReactNode} The rendered component.
25   */
26  function Provider({ children }) {
27    const [books, setBooks] = useState([]);
28
29    /**
30     * Fetches books data from the server.
31     * @async
32     */
33    const fetchBooks = useCallback(async () => {
34      await axios.get("http://localhost:3001/books").then((res) => {
35        setBooks(res.data);
36      });
37    }, []);
38
39    /**
40     * Creates a new book.
41     * @param {string} title - The title of the book.
42     * @async
43     */
44    const createBook = async (title) => {
45      await axios
46        .post("http://localhost:3001/books", { title })
47        .then((res) => {
48          setBooks([...books, res.data]);
49        });
50    };
51  }
```

Figura 9 – Resposta do assistente Copilot, com a documentação

Avaliação: Boa. O Copilot consegue criar uma boa documentação dos métodos de um componente React, em forma de comentário temos os tipos dos parâmetros, o objetivo semântico daquela classe ou método.

3.3.5 Tratamento de erros

Foi feita a requisição para que o assistente gerasse tratamento de erros caso alguma exceção retornasse do banco de dados.

Como cortei o input com o request na imagem, vou colocar abaixo em texto.

Request: "Create error treatment for all external request made in this class"

A Figura 10 traz o resultado.

```
28
29
30  /**
31   * Fetches books data from the server.
32   * @async
33   */
34  const fetchBooks = useCallback(async () => {
35    try {
36      const res = await axios.get("http://localhost:3001/books");
37      setBooks(res.data);
38    } catch (error) {
39      console.error("Error fetching books:", error);
40    }
41  }, []);
42  /**
```

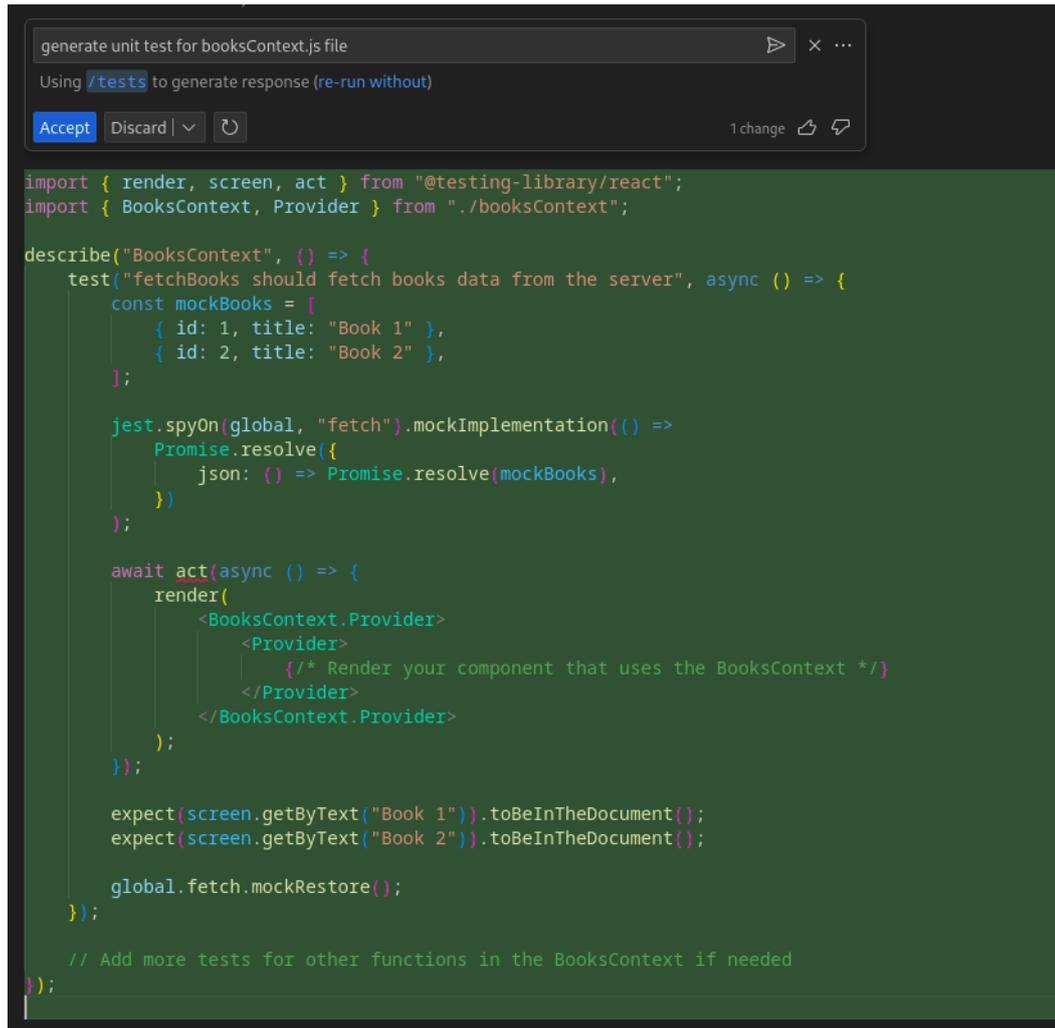
Figura 10 – Resposta do assistente Copilot, com a inclusão do tratamento de erro

Avaliação: Média. Essencialmente é feito um try/catch em todas as chamadas externas, porém o Copilot não costuma criar tipos específicos para os tipos de erros, talvez se o request fosse feito em outra linguagem, o assistente desse essa sugestão.

Nestes primeiros exemplos quisemos destacar a capacidade do assistente em gerar código e sugestões a partir de um código fonte já existente, nos próximos exemplos vamos querer avaliar a capacidade do assistente em criar código a partir do input de texto.

3.3.6 Geração de Testes Unitários

Pedimos para o assistente gerar testes unitários para um dos componentes da nossa aplicação. A Figura 11 traz o resultado.



```

generate unit test for booksContext.js file
Using /tests to generate response (re-run without)
Accept Discard 1 change

import { render, screen, act } from "@testing-library/react";
import { BooksContext, Provider } from "../booksContext";

describe("BooksContext", () => {
  test("fetchBooks should fetch books data from the server", async () => {
    const mockBooks = [
      { id: 1, title: "Book 1" },
      { id: 2, title: "Book 2" },
    ];

    jest.spyOn(global, "fetch").mockImplementation(() =>
      Promise.resolve({
        json: () => Promise.resolve(mockBooks),
      })
    );

    await act(async () => {
      render(
        <BooksContext.Provider>
          <Provider>
            { /* Render your component that uses the BooksContext */ }
          </Provider>
        </BooksContext.Provider>
      );
    });

    expect(screen.getByText("Book 1")).toBeInTheDocument();
    expect(screen.getByText("Book 2")).toBeInTheDocument();

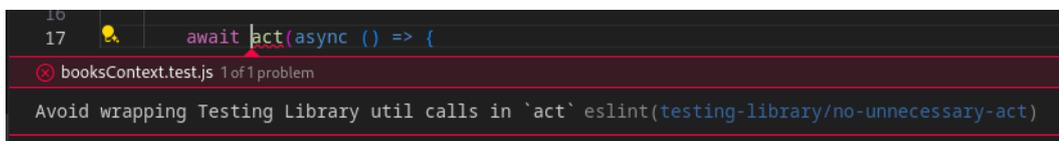
    global.fetch.mockRestore();
  });

  // Add more tests for other functions in the BooksContext if needed
});

```

Figura 11 – Resposta do assistente Copilot, com a geração dos testes unitários

Avaliação: Ruim. O Copilot não consegue criar os testes unitários dos métodos da classe, ele cria apenas de um método, e o arquivo ainda contém erros de compilação como destacado na Figura abaixo.



```

16
17 await act(async () => {
  booksContext.test.js 1 of 1 problem
  Avoid wrapping Testing Library util calls in `act` eslint(testing-library/no-unnecessary-act)

```

Figura 12 – Error de compilação nos testes unitários gerados pelo Copilot

3.3.7 Análise do comportamento do assistente em conceitos ReactJS

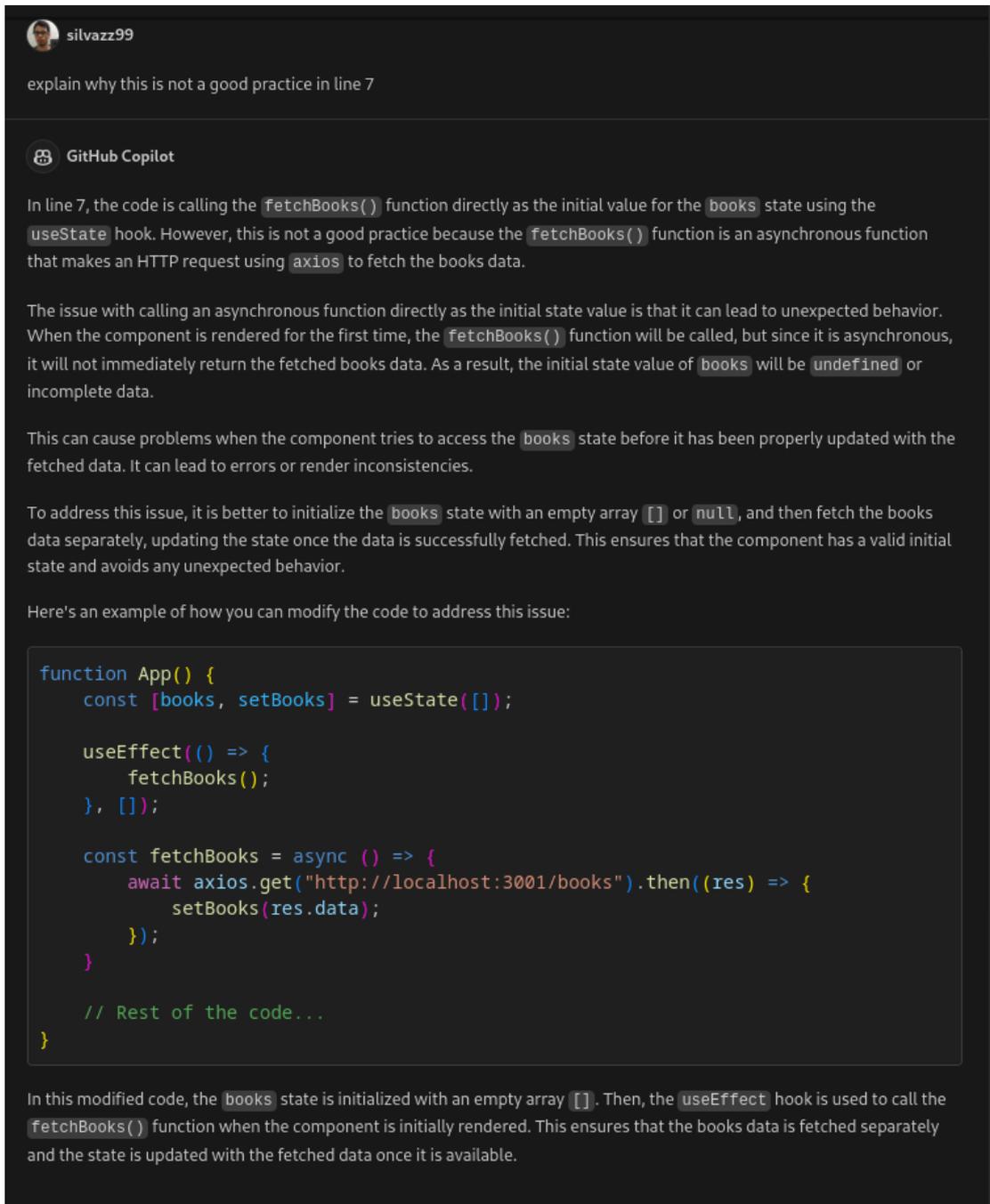
Nesta seção, exploramos a interação com o assistente para obter explicações sobre conceitos fundamentais e melhores práticas no contexto do desenvolvimento ReactJS. Ao solicitarmos insights sobre programação e orientações relacionadas ao ReactJS, observamos

o comportamento do assistente, buscando avaliar sua capacidade de fornecer esclarecimentos relevantes e orientações coerentes. Essa análise visa não apenas compreender a habilidade do assistente em explicar conceitos complexos, mas também a sua eficácia na promoção de boas práticas no desenvolvimento com ReactJS.

Nesta requisição, Figura 14, pedimos ao assistente a explicar os motivos pelos quais a chamada de função apresentada na Figura 13 não é considerada a melhor prática.

```
5
6 function BookComponent({ children }) {
7   const [books] = fetchBooks();
8 }
```

Figura 13 – Código ReactJS



silvazz99

explain why this is not a good practice in line 7

GitHub Copilot

In line 7, the code is calling the `fetchBooks()` function directly as the initial value for the `books` state using the `useState` hook. However, this is not a good practice because the `fetchBooks()` function is an asynchronous function that makes an HTTP request using `axios` to fetch the books data.

The issue with calling an asynchronous function directly as the initial state value is that it can lead to unexpected behavior. When the component is rendered for the first time, the `fetchBooks()` function will be called, but since it is asynchronous, it will not immediately return the fetched books data. As a result, the initial state value of `books` will be `undefined` or incomplete data.

This can cause problems when the component tries to access the `books` state before it has been properly updated with the fetched data. It can lead to errors or render inconsistencies.

To address this issue, it is better to initialize the `books` state with an empty array `[]` or `null`, and then fetch the books data separately, updating the state once the data is successfully fetched. This ensures that the component has a valid initial state and avoids any unexpected behavior.

Here's an example of how you can modify the code to address this issue:

```
function App() {
  const [books, setBooks] = useState([]);

  useEffect(() => {
    fetchBooks();
  }, []);

  const fetchBooks = async () => {
    await axios.get("http://localhost:3001/books").then((res) => {
      setBooks(res.data);
    });
  }

  // Rest of the code...
}
```

In this modified code, the `books` state is initialized with an empty array `[]`. Then, the `useEffect` hook is used to call the `fetchBooks()` function when the component is initially rendered. This ensures that the books data is fetched separately and the state is updated with the fetched data once it is available.

Figura 14 – Resposta do assistente Copilot, com a explicação de porque a imagem anterior não é uma boa prática

Avaliação: Boa. O assistente me dá uma explicação concisa sobre o request, consegue me trazer um exemplo de como eu poderia resolver esse problema da falta de boa prática.

Na requisição subsequente, buscamos orientação conceitual junto ao assistente, indagando sobre a melhor prática no contexto do ReactJS referente à passagem de objetos através de um *Provider*.

Provider, é um componente que facilita o compartilhamento de dados ou funcionalidades com seus componentes filhos.

Figura 15 mostra a requisição.

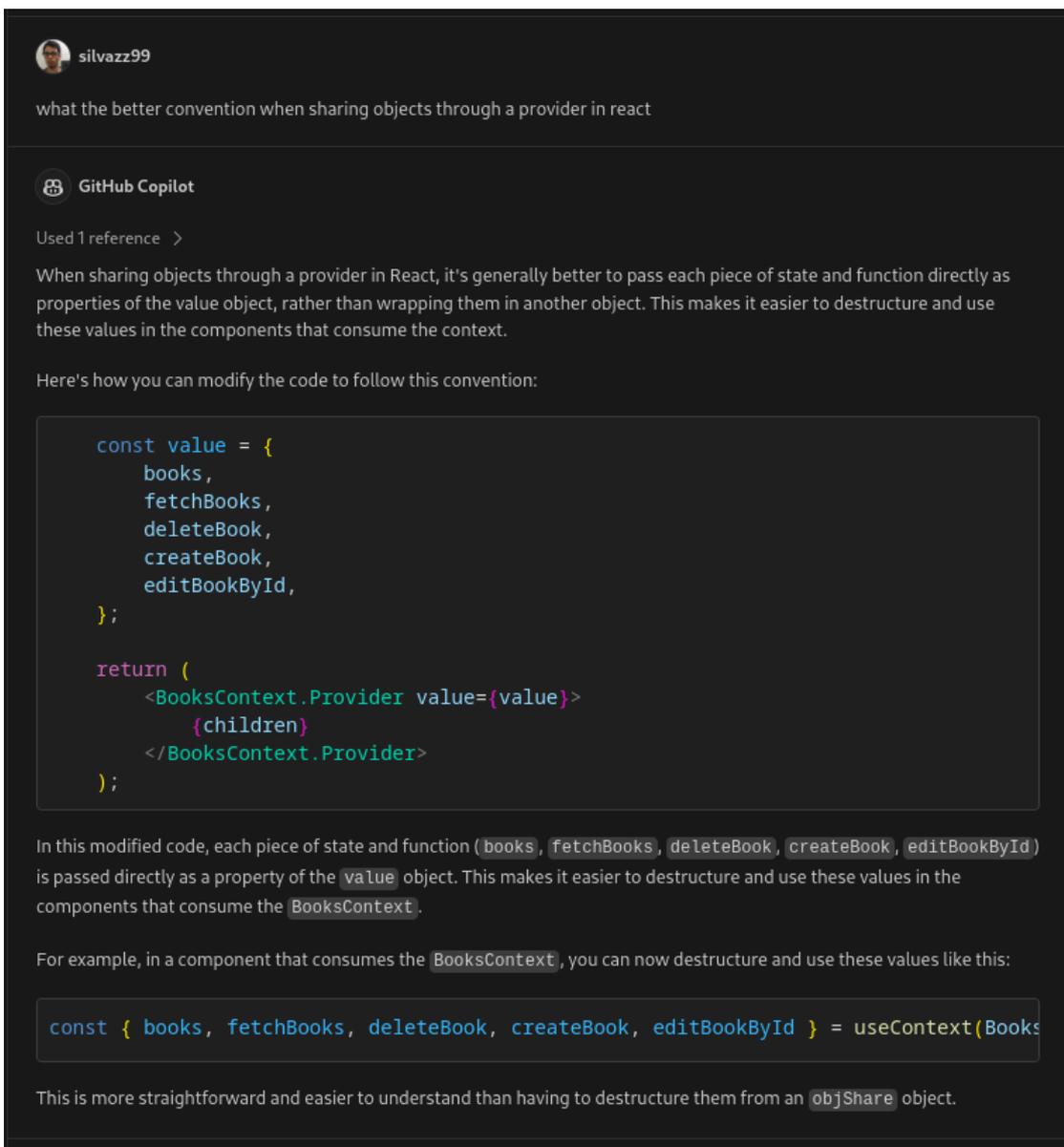


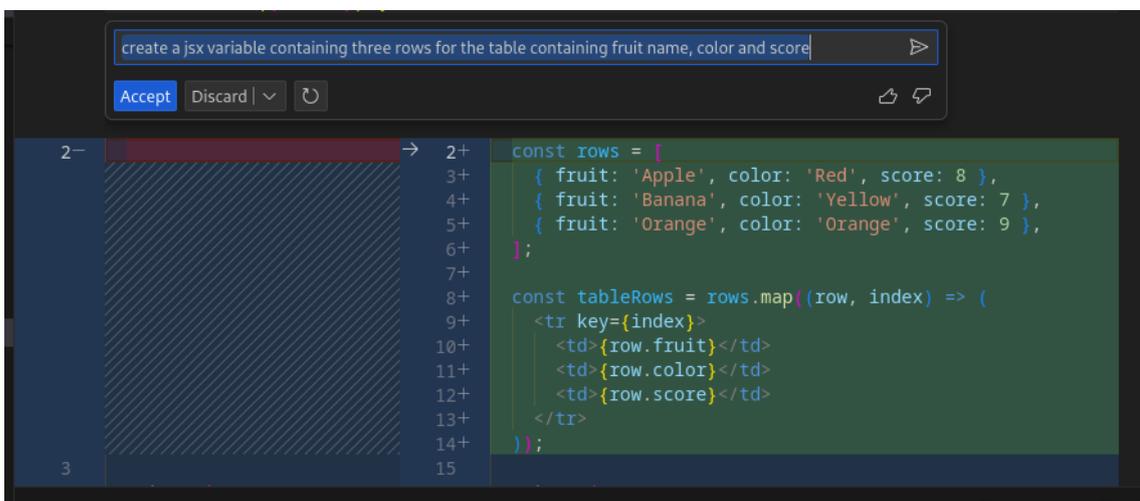
Figura 15 – Resposta do assistente Copilot, com a resposta de qual modo poderíamos seguir de maneira correta a convenção do framework

Avaliação: Boa. O assistente usa exemplos do projeto que eu estou criando para me ajudar no entendimento sobre a questão do Provider.

Nestes primeiros exemplos quisemos destacar a capacidade do assistente em gerar código e sugestões a partir de um código fonte já existente, nos próximos exemplos vamos querer avaliar a capacidade do assistente em criar código a partir do input de texto. Essas solicitações não guardam relação com o projeto desenvolvido até o momento, pois visamos realizar novos experimentos que se desviam do escopo inicial.

3.3.8 Criação de elementos de HTML

Foi feita uma requisição ao assistente para que ele criasse uma tabela com três linhas com as colunas sendo nome, cor e pontuação. A Figura 16 traz o resultado.



The image shows a code editor interface with a Copilot suggestion. At the top, a text input field contains the prompt: "create a jsx variable containing three rows for the table containing fruit name, color and score". Below the input are buttons for "Accept", "Discard", and a refresh icon. The suggestion itself is a code block with the following content:

```
const rows = [
  { fruit: 'Apple', color: 'Red', score: 8 },
  { fruit: 'Banana', color: 'Yellow', score: 7 },
  { fruit: 'Orange', color: 'Orange', score: 9 },
];

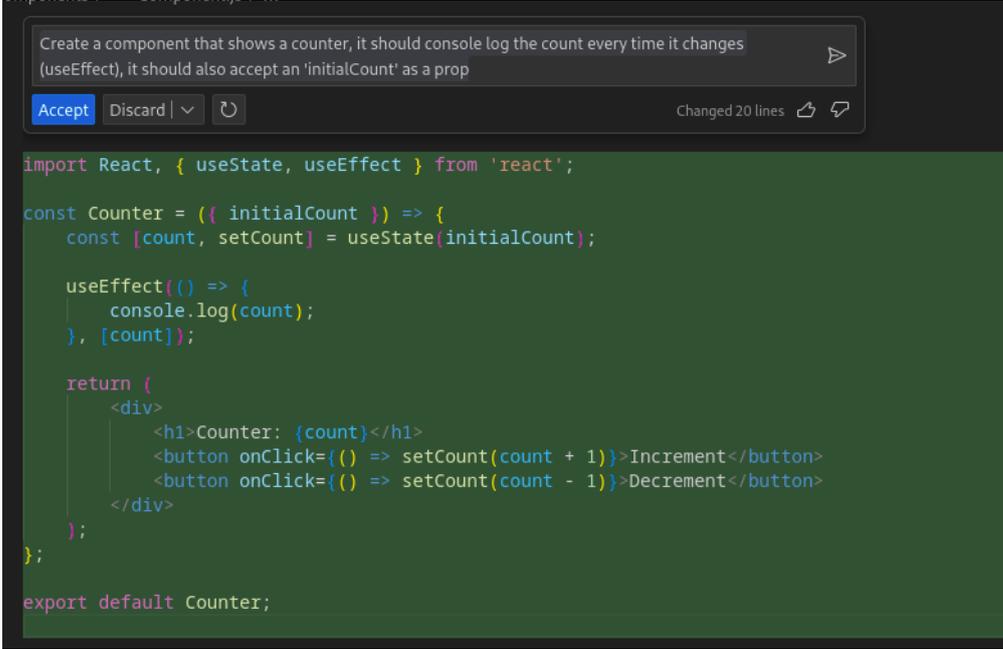
const tableRows = rows.map((row, index) => (
  <tr key={index}>
    <td>{row.fruit}</td>
    <td>{row.color}</td>
    <td>{row.score}</td>
  </tr>
));
```

Figura 16 – Resposta do assistente Copilot, com a sugestão dos elementos HTML

Avaliação: Boa. Foi pedido para que se fizesse uma criação simples de alguns elementos HTML, e o assistente foi capaz de fazer.

3.3.9 Criação de um componente

Requisição para que o assistente crie um componente que mostre um contador, faz o logging dos valores desse contador toda vez que houver uma mudança, usando o hooks *useEffect*, e deve aceitar também uma prop com um valor inicial. A Figura 17 traz o resultado.



```
import React, { useState, useEffect } from 'react';

const Counter = ({ initialCount }) => {
  const [count, setCount] = useState(initialCount);

  useEffect(() => {
    console.log(count);
  }, [count]);

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
};

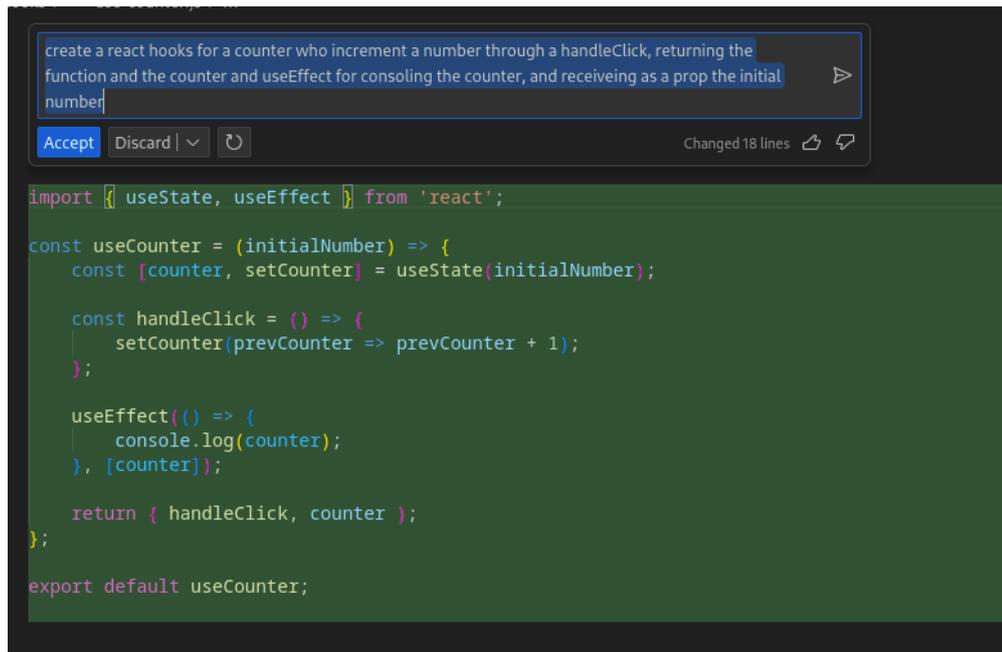
export default Counter;
```

Figura 17 – Resposta do assistente Copilot, com a geração de um componente

Avaliação: Boa. Quando pedido para se fazer um componente com um contador, o Copilot foi capaz de criar um sozinho, não foi especificado se teria ou não uma função de decremento mas ele criou mesmo assim.

3.3.10 Criação de React Hooks

Outra requisição ao assistente foi a criação de um Custom Hook para que seja incrementado um contador através de um clique que seja disparado pelo usuário, retornando uma função para a mudança de estado e o contador, usar o *useEffect()* para logar o contador e que receba uma prop como valor inicial. A Figura 18 traz o resultado.

A screenshot of a code editor interface. At the top, a text input field contains the prompt: "create a react hooks for a counter who increment a number through a handleClick, returning the function and the counter and useEffect for consoling the counter, and receiveing as a prop the initial number". Below the input are buttons for "Accept", "Discard", and a refresh icon. To the right of the buttons, it says "Changed 18 lines". The main area of the editor shows the following JavaScript code:

```
import { useState, useEffect } from 'react';

const useCounter = (initialNumber) => {
  const [counter, setCounter] = useState(initialNumber);

  const handleClick = () => {
    setCounter(prevCounter => prevCounter + 1);
  };

  useEffect(() => {
    console.log(counter);
  }, [counter]);

  return { handleClick, counter };
};

export default useCounter;
```

Figura 18 – Resposta do assistente Copilot, com a geração de um Hook customizado

Avaliação: Boa. Quando pedido para criar um Hook o Copilot foi capaz de me ajudar, eu pedi um Hook apenas com incrementador dessa vez, e uma função para printar o valor do contador toda vez que houvesse mudança no valor, e isso foi feito.

3.3.11 Criação de um componente usando *useReducer*

A última requisição feita para o assistente foi a criação de um componente que tivesse um contador com as funcionalidades de incremento e decremento, tivesse uma entrada através de um componente de input para adicionar valores arbitrários, usando *useReducer*.

A Figura 19 traz o resultado.

```
Create a component counter who will have a decrement, increment and an input to add an arbitrary value to add options using useReducer, and will have an props as initial value
Accept Discard | v ↻ Changed 48 lines

import React, { useReducer, useState } from 'react';

const initialState = {
  count: 0,
};

function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + action.value };
    case 'DECREMENT':
      return { count: state.count - action.value };
    case 'ADD_VALUE':
      return { count: state.count + action.valueToAdd };
    default:
      throw new Error();
  }
}

function Counter({ initialValue }) {
  const [state, dispatch] = useReducer(reducer, { count: initialValue });
  const [valueToAdd, setValueToAdd] = useState(0);

  const handleIncrement = () => {
    dispatch({ type: 'INCREMENT', value: 1 });
  };

  const handleDecrement = () => {
    dispatch({ type: 'DECREMENT', value: 1 });
  };

  const handleAddValue = () => {
    dispatch({ type: 'ADD_VALUE', valueToAdd });
  };

  return (
    <div>
      <button onClick={handleIncrement}>Increment</button>
      <button onClick={handleDecrement}>Decrement</button>
      <input type="number" value={valueToAdd} onChange={(e) => setValueToAdd(parseInt(e.target.value))} />
      <button onClick={handleAddValue}>Add Value</button>
      <p>Count: {state.count}</p>
    </div>
  );
}
```

Figura 19 – Resposta do assistente Copilot, com a geração de um *useReducer*

Avaliação: Média. Apenas na terceira tentativa que eu consegui o que eu queria nessa requisição, no método do reducer ele utiliza um switch case, eu gostaria que ele utilizasse uma lookup table que é a forma mais moderna de se fazer testes nesses objetos em JavaScript.

4 Usando LLM com assistentes de prova de Teoremas

Em um mundo ideal um programa de computador seria capaz de determinar se uma afirmação lógica é verdadeira ou falsa, enquanto essa tarefa é possível para certas afirmações, uma revelação no começo do século 20 afirmou que seria impossível, de maneira geral, para um computador conseguir afirmar que qualquer afirmação lógica é verdade ou não. De forma independente, em 1936, Alonzo Church e Alan Turing demonstraram essa limitação. Church utilizou o cálculo lambda como um modelo de computação, enquanto Turing utilizou a máquina hoje conhecida como máquina de Turing. A tese de Church-Turing postula que tanto o cálculo lambda como as máquinas de Turing formalizam o conceito de "computação". [Clarkson et al. 2024]

4.1 Equivalência Curry-Howard

4.1.1 Relação entre Tipos e Proposições

Podemos conceituar funções como instrumentos para manipulação e transformação de evidências - em vez de interpretar “tem tipo”, interpretemos como “é evidência para” -, da mesma maneira que estamos acostumados a visualizar funções como dispositivos para manipulação e transformação de dados. Ao analisar as seguintes funções, que exemplificam a criação e decomposição de pares.

```

1 let pair x y = (x, y)
2 let fst (x, y) = x
3 let snd (x, y) = y
4
5 val pair : 'a -> 'b -> 'a * 'b = <fun>
6 val fst : 'a * 'b -> 'a = <fun>
7 val snd : 'a * 'b -> 'b = <fun>

```

Com o propósito de estabelecer a conjunção entre A e B, é imperativo demonstrar a validade de A e B de maneira independente. Além disso, ao possuir a comprovação da conjunção de A e B, é possível inferir a validade de A e B individualmente. Esses padrões podem ser formalizados por meio de fórmulas lógicas, empregando \wedge para denotar a conjunção e \rightarrow para indicar a implicação.

Provas representam um tipo de evidência, sendo argumentos lógicos acerca da veracidade de uma expressão. Ao visitar os tipos `pair`, `fst`, `snd` e contrastá-los com as formulações lógicas que delineiam padrões válidos, podemos afirmar que:

```

1 val pair : 'a -> 'b -> 'a * 'b           A -> B -> A /\ B
2 val fst  : 'a * 'b -> 'a                A /\ B -> A
3 val snd  : 'a * 'b -> 'b                A /\ B -> B

```

A constatação acima revela que realizar computações com evidências corresponde à construção de provas lógicas válidas. Essa correspondência não é uma mera coincidência limitada a estes três expressões em particular, ao contrário, reflete uma ligação significativa entre os domínios da programação e da lógica.

4.1.2 Relação programas e provas

Dado que programas e provas constituem duas formas de transformar e manipular evidências, vamos nos ater aos casos de conjunção e implicação, ou equivalentemente, pares e funções. Uma variável adquire o tipo conforme o contexto em que está inserida.

```

1 env |- fun x -> e : t -> t'
2 if env [x -> t] |- e : t'

```

Uma função anônima `fun x → e` tem o tipo `t → t'` se `e` tem o tipo `t'` em um ambiente estático que vincula `x` para o tipo `t`.

```

1 env |- e1 e2 : t'
2 if env |- e1 : t -> t'
3 and env |- e2 : t

```

Uma aplicação de `e1 e2` tem tipo `t'` se `e1` tem o tipo `t → t'` e `e2` tem o tipo `t`.

```

1 env |- (e1, e2) : t1 * t2
2 if env |- e1 : t1
3 and env |- e2 : t2

```

O par `(e1, e2)` tem o tipo `t1 * t2` se `e1` tem o tipo `t1` e `e2` tem o tipo `t2`.

```

1 env |- fst e : t1
2 if env |- e : t1 * t2
3
4 env |- snd e : t2
5 if env |- e : t1 * t2

```

Se `e` tem tipo `t1 * t2`, então `fst e` tem tipo `t1`, e `snd e` tem tipo `t2`.

Árvore de prova: Vamos reescrever cada uma dessas árvores deixando apenas os tipos.

```

1 env , p |- p
2
3
4 env , p1 |- p2
5 -----
6 env |- p1 -> p2
7
8
9 env |- p1 -> p2      env |- p1
10 -----
11          env |- p2
12
13
14 env |- p1      env |- p2
15 -----
16          env |- p1 /\ p2
17
18
19 env |- p1 /\ p2
20 -----
21          env |- p1
22
23
24 env |- p1 /\ p2
25 -----
26          env |- p2

```

Cada proposição pode ser interpretada como uma afirmação lógica formalmente válida. Sempre que expressamos $\text{env} \vdash p$, estamos afirmando que "a partir de todas as premissas em env , podemos inferir que p é verdadeiro".

Agora, examinando a árvore subsequente, que apresenta a dedução dos tipos do programa:

```

1 {p : a * b} |- p : a * b      {p : a * b} |- p : a * b
2 -----
3 {p : a * b} |- snd p : b      {p : a * b} |- fst p : a
4 -----
5          {p : a * b} |- (snd p, fst p) : b * a

```

```

6 -----
7 {} |- fun p -> (snd p, fst p) : a * b -> b * a

```

Deixando apenas os tipos e reescrevendo as proposições, chegamos em:

```

1 a /\ b |- a /\ b          a /\ b |- a /\ b
2 -----
3 a /\ b |- b              a /\ b |- a
4 -----
5           a /\ b |- b /\ a
6 -----
7 {} |- a /\ b -> b /\ a

```

Esta é uma árvore de prova válida para a lógica proposicional.

O que nós vimos com as duas árvores de provas acima: programas são provas. Um programa bem tipado corresponde a uma prova de uma proposição lógica.

Com isso, concluímos que a avaliação do programa causa simplificação da árvore de prova, a árvore simplificada é uma prova mais simples da mesma proposição. Avaliação portanto corresponde a simplificação de uma prova.

A equivalência de Curry-Howard mostra que lógica e computação estão fundamentalmente ligadas de uma maneira profunda. Os blocos básicos da lógica (proposições, provas) acabam por corresponder aos componentes básicos da computação (tipos, programas funcionais). A própria computação, no âmbito da avaliação ou simplificação de expressões, acaba por corresponder à simplificação de provas. A tarefa que os computadores realizam, portanto, é a mesma tarefa que os humanos fazem ao tentar mostrar uma prova da maneira mais simples possível. [Clarkson et al. 2024]

4.2 O que é o Lean

Lean é uma implementação de uma fundação lógica conhecida como teoria de tipos dependentes. Especificamente, ela implementa uma versão da teoria de tipos dependentes conhecida como Cálculo das Construções Indutivas (Calculus of Inductive Constructions - CIC). O CIC é uma linguagem formal com um conjunto pequeno e preciso de regras que regem a formação de expressões. Neste sistema formal, cada expressão possui um tipo. O tipo da expressão indica que tipo de objeto a expressão denota.

Uma característica importante da teoria de tipos dependentes é que cada expressão possui uma interpretação computacional, ou seja, existem regras que especificam como elas podem ser reduzidas a uma forma normal.

Devido à expressividade da teoria de tipos dependentes, podemos utilizar todos os métodos e técnicas comuns de programação funcional, incluindo *higher types*, classes de tipos, registros, mônadas e outras abstrações. Na verdade, temos toda a biblioteca do Lean à nossa disposição. [Avigad et al. 2017]

4.2.1 Quem criou o Lean?

A linguagem Lean foi criada por Leonardo de Moura, um pesquisador brasileiro, em colaboração com outros pesquisadores, como Soonho Kong e Jeremy Avigad. A criação da linguagem Lean teve como objetivo fornecer um ambiente poderoso para a formalização matemática e a verificação formal de teoremas. Ela foi projetada para ser uma linguagem de programação, baseada na teoria de tipos dependentes, permitindo a expressão precisa de conceitos matemáticos e a construção de provas formais. O desenvolvimento do Lean é parte integrante da pesquisa em assistentes de prova e lógica computacional.

4.3 Usando o Copilot com Lean

Dentro dos nossos experimentos, queremos utilizar o assistente de programação no contexto resolução de provas utilizando o Lean. Nosso objetivo é verificar se as experiências que temos ao utilizar o assistente com linguagens de programação e framework mais comuns se repetem ao utilizar com um provador de teoremas, como o Lean.

Na próxima seção iremos apresentar a metodologia utilizada para os experimentos feitos com Lean e o Copilot queremos ao final do capítulo, lembrando que definição da coleta de dados que estamos utilizando aqui é menos rigorosa do que o de costume, pois estamos interagindo com um modelo de linguagens de programação, e as respostas que são enviadas para gente depende do que escrevemos na entrada como requisição.

4.3.1 Como configurar o Lean na sua máquina

No experimento realizado para instalar e executar o Lean em minha máquina, a escolha foi integrá-lo como uma extensão na IDE VSCode. A instalação do Lean como extensão no VSCode proporcionou um ambiente de desenvolvimento mais acessível e facilitou a interação com a linguagem. A integração também possibilitou a utilização do Copilot durante o processo de desenvolvimento.

A versão do Lean utilizada foi a 4, a versão do VSCode foi a 1.86 e a versão do Copilot foi a 1.7.

4.4 Metodologia

Similar ao que foi feito no capítulo anterior no projeto desenvolvido com React, neste experimento queremos testar as capacidades do Copilot em nos ajudar a desenvolver provas usando Lean. Nosso objetivo é conduzir um experimento, aplicando as respostas do modelo e assistente nos programas desenvolvidos em Lean para avaliar se o assistente de programação tem capacidade de nos ajudar a desenvolver provas e como isso se compara com assistentes de programação em geral. Nos nossos experimentos abordaremos testes com lógica proposicional, tipos indutivos, provas matemáticas, indução e recursão e incluiremos também testes com quantificadores e igualdade. Ao explorar as possibilidades do Copilot nesta perspectiva, buscamos contribuir significativamente para o entendimento de como ferramentas de programação assistida podem potencializar o desenvolvimento de provas usando assistentes de programação. A maioria dos exercícios e problemas que estaremos resolvendo vem do livro online disponível em [Avigad et al. 2024] no site da documentação do Lean. Ressaltando que nos testes apresentados abaixo, os trechos de código destacados em verde correspondem às sugestões fornecidas pelo assistente. Quando não for o caso, explicitarei essa condição. Todo o restante do código apresentado foi desenvolvido por mim.

4.5 Testes com Lean e Copilot

4.5.1 Proposições e Provas

Neste seção, iremos escrever requisições para o assistente com a finalidade de criar provas matemáticas usando tipos dependentes. A primeira propriedade que queremos demonstrar é a da comutatividade, então iremos requisitar ao Copilot para nos escrever uma prova da identidade $p \wedge q \leftrightarrow q \wedge p$. A Figura 20 mostra o resultado.

```

variable (p q : Prop)

given variable (p q : Prop) as a propositions, Prove the following identities, replacing the
"sorry" placeholders with actual proofs, using Lean 4.
example : p ^ q ↔ q ^ p := sorry

Accept Discard | v ↻ Changed 10 lines
example : p ^ q ↔ q ^ p :=
begin
  split,
  { intro h,
    cases h with hp hq,
    split; assumption },
  { intro h,
    cases h with hq hp,
    split; assumption }
end

```

Figura 20 – Resposta do assistente Copilot, com a sugestão de resolução

Neste caso temos uma prova que não funciona corretamente com o compilador do Lean 4, depois de alguns ajustes consegui fazer com que a prova fique correta, mostrando na Figura 21. É válido notar que quando o assistente usa o conteúdo do arquivo em questão para dar a resposta, ou seja além da entrada de texto da requisição o assistente também leva em conta o que está escrito no arquivo e isso pode influenciar na resposta de saída.

```

Prove the following identities, replacing the "sorry" placeholders with actual proofs, using
Lean 4
variable (p q r : Prop)
-- commutativity of ^ and v
example : p ^ q ↔ q ^ p := sorry

Accept Discard | v ↻ Changed 6 lines
example : p ^ q ↔ q ^ p :=
  Iff.intro
    (fun h : p ^ q =>
      And.intro (And.right h) (And.left h))
    (fun h : q ^ p =>
      And.intro (And.right h) (And.left h))

```

Figura 21 – Resposta do assistente Copilot, com a sugestão de resolução

Vamos tentar provar a propriedade da associatividade na seguinte identidade $(p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$, usando o assistente temos o resultado na Figura 22:

```

18
19 -- FILEPATH: /home/silvaz99/Documents/UFRGS/tcc/lean4-samples/HelloWorld/
20 -- BEGIN: abpxx6d04wxr
21 variable (p q r : Prop)
22
23 example : (p ∧ q) ∧ r ↔ p ∧ (q ∧ r) :=
24   iff.intro
25     (assume h : (p ∧ q) ∧ r,
26       have hpq : p ∧ q, from h.left,
27       have hr : r, from h.right,
28       have hp : p, from hpq.left,
29       have hq : q, from hpq.right,
30       show p ∧ (q ∧ r), from and.intro hp (and.intro hq hr))
31     (assume h : p ∧ (q ∧ r),
32       have hp : p, from h.left,
33       have hqr : q ∧ r, from h.right,
34       have hq : q, from hqr.left,
35       have hr : r, from hqr.right,
36       show (p ∧ q) ∧ r, from and.intro (and.intro hp hq) hr)
37 -- END: abpxx6d04wxr
38 Chris Lovett, 23 months ago • doc: first hello world sample

```

Figura 22 – Resposta do assistente Copilot, com a sugestão de resolução

Depois de alguns ajustes chegamos nesse resultado, Figura 23.

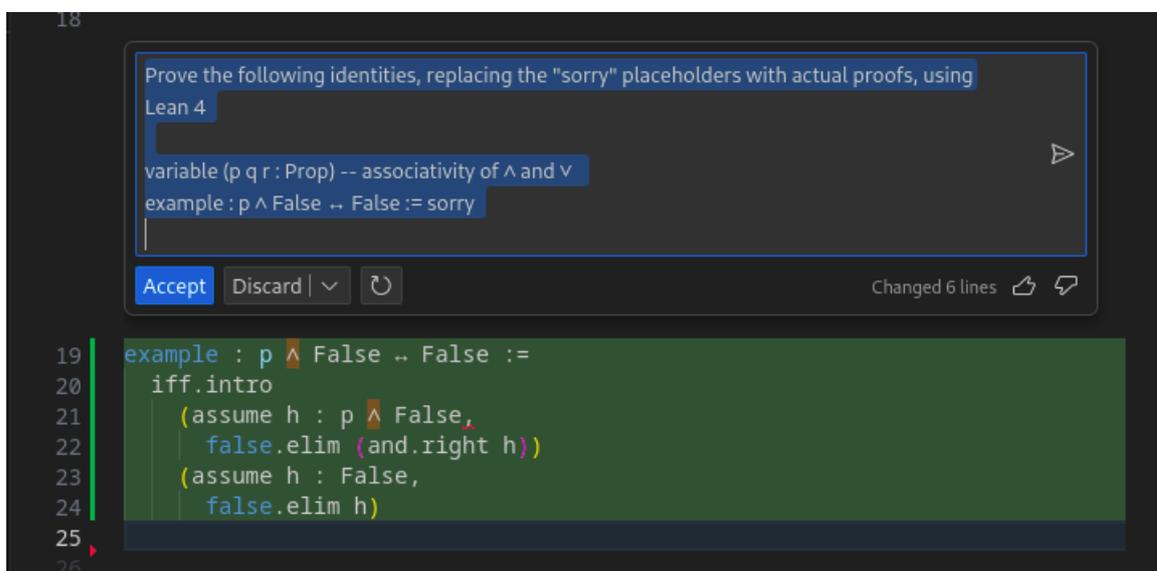
```

16
17 variable (p q r : Prop)
18
19 example : (p ∧ q) ∧ r ↔ p ∧ (q ∧ r) :=
20   Iff.intro
21     (fun h : (p ∧ q) ∧ r =>
22       have hpq : p ∧ q := h.left
23       have hr : r := h.right
24       have hp : p := hpq.left
25       have hqr : q ∧ r := ⟨hpq.right, hr⟩
26       ⟨hp, hqr⟩)
27     (fun h : p ∧ (q ∧ r) =>
28       have hp : p := h.left
29       have hqr : q ∧ r := h.right
30       have hq : q := hqr.left
31       have hr : r := hqr.right
32       have hpq : p ∧ q := ⟨hp, hq⟩
33       ⟨hpq, hr⟩)
34

```

Figura 23 – A minha resolução para este exercício

Agora vamos tentar provar outra propriedade usando Lean $p \wedge \text{False} \leftrightarrow \text{False}$, a resposta do assistente mostrado na Figura 24:



```

18
19 example : p ∧ False ↔ False :=
20   iff.intro
21     (assume h : p ∧ False,
22      false.elim (and.right h))
23     (assume h : False,
24      false.elim h)
25
26

```

Figura 24 – Resposta do assistente Copilot, com a sugestão de resolução

Como essa prova não está correta sintaticamente fazemos alguns ajustes, como mostrado na Figura 25:

```

16
17 variable (p q r : Prop)
18
19 example : p ^ False ↔ False :=
20   Iff.intro
21     (fun h : p ^ False =>
22       False.elim (And.right h))
23     (fun h : False =>
24       False.elim h)
25
26

```

Figura 25 – Minha resolução para o problema

4.5.2 Quantificadores e Igualdade

Nesta seção vamos expandir os nossos experimentos, dessa vez utilizando construções lógicas para incluir quantificadores universais e existenciais e a relação de equidade. O quantificador universal, $\forall x : \alpha, p x$ denota que “para todo $x : \alpha$, $p x$ ” é verdade. Sendo assim dado uma prova que $p x$ é verdade em um contexto onde $x : \alpha$ é arbitrário, então obtemos uma prova que $\forall x : \alpha, p x$.

A primeira equivalência que queremos demonstrar é $\forall x(p(x) \wedge q(x)) \leftrightarrow (\forall x p(x) \wedge \forall x q(x))$, nas duas imagens abaixo temos a resposta do assistente, e a resposta que eu desenvolvi (Figura 26 e Figura 27), é válido destacar que eu peço pro assistente usar o modo *proof term* na hora da resolução da prova, o que não acontece:

```

Prove these equivalences, using "term-style":
variable (α : Type) (p q : α → Prop)
example : (∀ x, p x ∧ q x) ↔ (∀ x, p x) ∧ (∀ x, q x) := sorry

Accept Discard | v ↻ Changed 29 lines
variable (α : Type) (p q : α → Prop)
example : (∀ x, p x ∧ q x) ↔ (∀ x, p x) ∧ (∀ x, q x) :=
begin
  split,
  {
    intro h,
    split,
    {
      intro x,
      exact (h x).left,
    },
    {
      intro x,
      exact (h x).right,
    },
  },
  {
    intro h,
    intro x,
    split,
    {
      exact (h.left x),
    },
    {
      exact (h.right x),
    },
  },
end

```

Figura 26 – Resposta sugerida pelo assistente

```

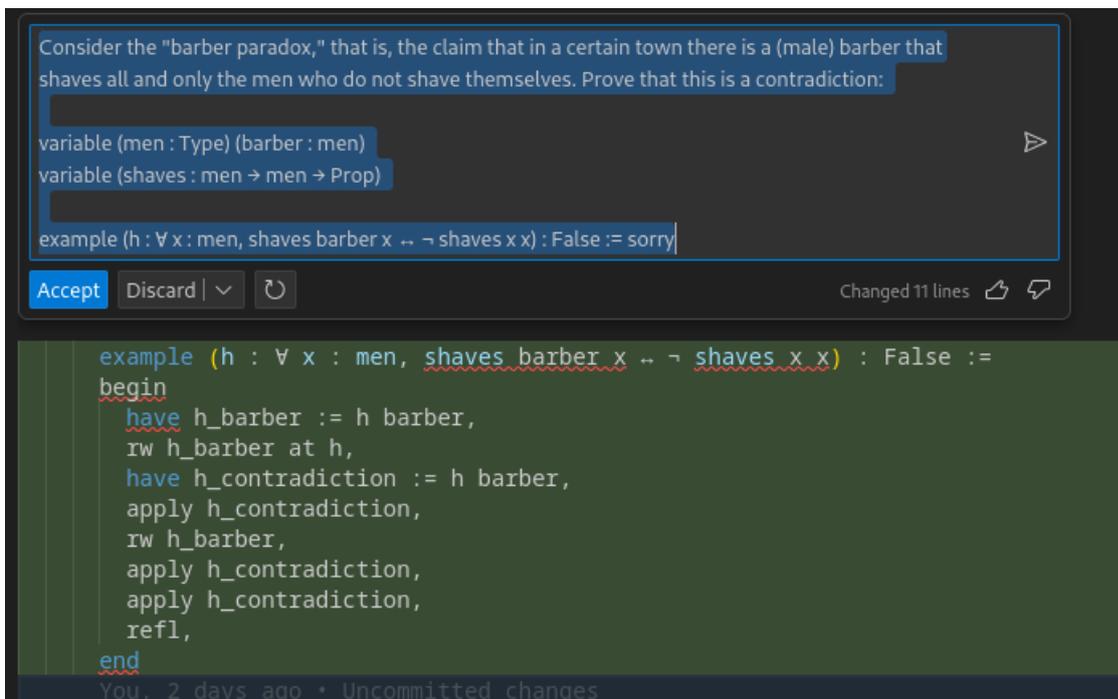
You, 2 days ago | 2 authors (You and others)
1 | variable (α : Type) (p q : α → Prop)
2 |
3 |
4 |
5 | example : (∀ x, p x ∧ q x) ↔ (∀ x, p x) ∧ (∀ x, q x) :=
6 |   Iff.intro
7 |     (fun h : ∀ x, p x ∧ q x =>
8 |       And.intro
9 |         (fun x : α => (h x).left)
10 |        (fun x : α => (h x).right))
11 |     (fun h : (∀ x, p x) ∧ (∀ x, q x) =>
12 |       fun x : α => And.intro (h.left x) (h.right x))
13 |

```

Figura 27 – Resposta desenvolvida por mim

Na segunda sequência de resolução pedimos ao assistente para nos ajudar a provar um paradoxo do barbeiro, que afirma que, numa determinada cidade existe um barbeiro (homem) que faz a barba de todos e apenas dos homens que não se barbeiam, nas duas imagens (Figura 28 e Figura 29) abaixo estão a resposta do assistente e na segunda é pedido que se use o *term-style*.

Vale uma breve explicação aqui, no contexto do Lean, o termo *term-style* refere-se geralmente ao modo *proof term* ou “tática termo” em português. Nesse modo, você constrói a prova diretamente por meio da manipulação de termos e expressões, sem depender de táticas específicas.



The screenshot shows a chat window with a dark background. The top part contains a text prompt in a light blue box: "Consider the 'barber paradox,' that is, the claim that in a certain town there is a (male) barber that shaves all and only the men who do not shave themselves. Prove that this is a contradiction:". Below the prompt is a code editor with the following Lean code:


```
variable (men : Type) (barber : men)
variable (shaves : men → men → Prop)

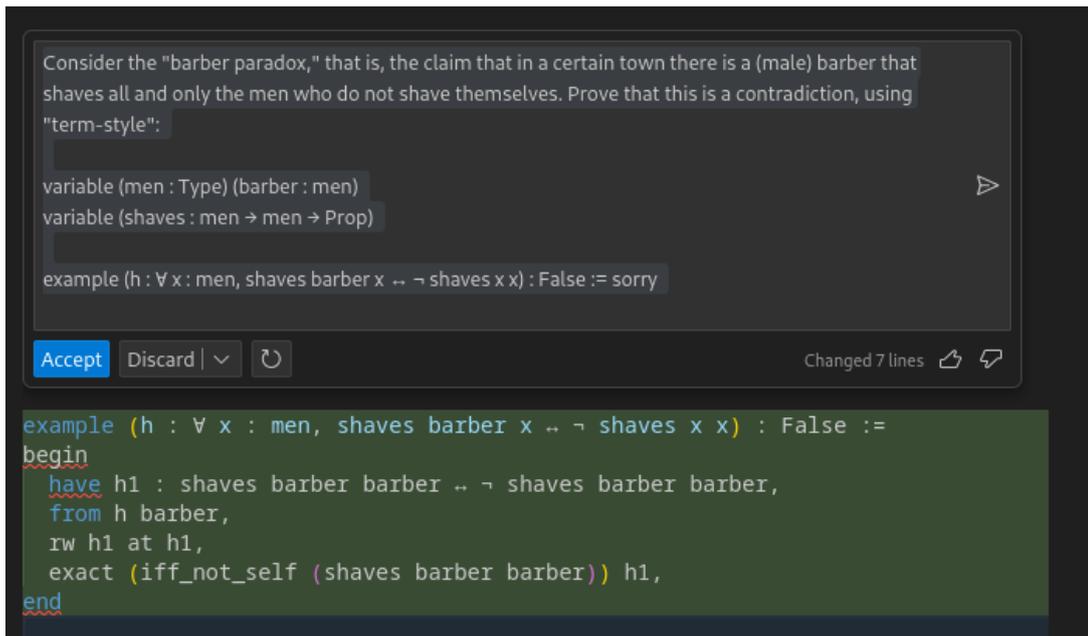
example (h : ∀ x : men, shaves barber x ↔ ¬ shaves x x) : False := sorry
```

 Below the code editor are buttons for "Accept", "Discard", and a refresh icon. To the right, it says "Changed 11 lines". The bottom part of the screenshot shows the assistant's response, which is a Lean proof in *term-style*:


```
example (h : ∀ x : men, shaves barber x ↔ ¬ shaves x x) : False :=
begin
  have h_barber := h barber,
  rw h_barber at h,
  have h_contradiction := h barber,
  apply h_contradiction,
  rw h_barber,
  apply h_contradiction,
  apply h_contradiction,
  refl,
end
```

 At the bottom, it says "You, 2 days ago • Uncommitted changes".

Figura 28 – Resposta do assistente Copilot, com a sugestão de resolução



```

Consider the "barber paradox," that is, the claim that in a certain town there is a (male) barber that
shaves all and only the men who do not shave themselves. Prove that this is a contradiction, using
"term-style":

variable (men : Type) (barber : men)
variable (shaves : men → men → Prop)

example (h : ∀ x : men, shaves barber x ↔ ¬ shaves x x) : False := sorry
  
```

Accept Discard | v ↻ Changed 7 lines

```

example (h : ∀ x : men, shaves barber x ↔ ¬ shaves x x) : False :=
begin
  have h1 : shaves barber barber ↔ ¬ shaves barber barber,
  from h barber,
  rw h1 at h1,
  exact (iff_not_self (shaves barber barber)) h1,
end
  
```

Figura 29 – Resposta do assistente Copilot, com a sugestão de resolução usando o 'term-style'

4.5.3 Demonstrações por cálculo

Nesta seção iremos criar provas de expressões matemáticas, iremos utilizar as tactics da linguagem Lean para a construções das provas, cada uma das tactics utilizadas nesta seção é apenas um passo da prova que foi construída. É válido citar que 'ring' nesse contexto significa apenas manipulações algébricas enquanto que 'rw' significa rewrite, que é quando estamos fazendo uma substituição. Na Figura 30 é mostrada a minha solução em seguida a solução do assistente (Figura 31).

```

1 import Mathlib
2
3
4 example {r s : ℝ} (h1 : s = 3) (h2 : r + 2 * s = -1) : r = -7 :=
5   calc
6     r = r + 2 * s - 2 * s := by ring
7     _ = -1 - 2 * s := by rw [h2]
8     _ = -1 - 2 * 3 := by rw [h1]
9     _ = -7 := by ring
  
```

Figura 30 – Minha resolução usando Lean 4

```

complete the proof below substituting sorry expression for the actual step of proof
Accept Discard | v ↻ Changed 8 lines ↵ ↶
-- END: abpxx6d04wxr → 3+ -- FILEPATH: /home/silvaz99/Documentos/UFRGS/tcc/lean4-samples/Hé
-- Example 1.2.2. 4+ -- BEGIN: be15d9bcejpp
-- Exercise: replace the words
example {r s : real} (h1 : s = 5 example {r s : real} (h1 : s = 3) (h2 : r + 2 * s = -1) : r = -7
  calc 6
  r = r + 2 * s - 2 * s := by 7
  _ = -1 - 2 * s := by sorry → 8+ r = r + 2 * s - 2 * s := by ring
  _ = -1 - 2 * 3 := by sorry 9+ _ = -1 - 2 * s := by rw h2
  _ = -7 := by sorry 10+ _ = -1 - 2 * 3 := by rw h1
  _ = -7 := by ring

```

Figura 31 – Sugestão de solução do Copilot

4.5.4 Tipos Indutivos

Tipos indutivos são usados em lógica matemática e teoria dos tipos para definir estruturas de dados recursivas. Eles são fundamentais para modelar e trabalhar com dados que podem ser construídos de maneira recursiva, como listas, árvores e outros tipos de dados. Nesta seção iremos fazer testes junto ao Copilot de alguns problemas usando tipos indutivos, na biblioteca do Lean, todo tipo concreto, exceto o do tipo universal e todo construtor de um tipo, exceto o de tipos dependentes, é uma instância de uma família geral de construções de tipos conhecidas como tipos indutivos. É válido destacar que seja possível construir boa parte dos conceitos matemáticos a partir dos tipos universais, tipos dependentes e tipos indutivos [Avigad et al. 2024].

O primeiro exercício que nós iremos testar no nosso assistente é mais um retirado do livro do Lean book, é pedido para definir um tipo de dado indutivo consistindo de termos construídos a partir dos seguintes construtores:

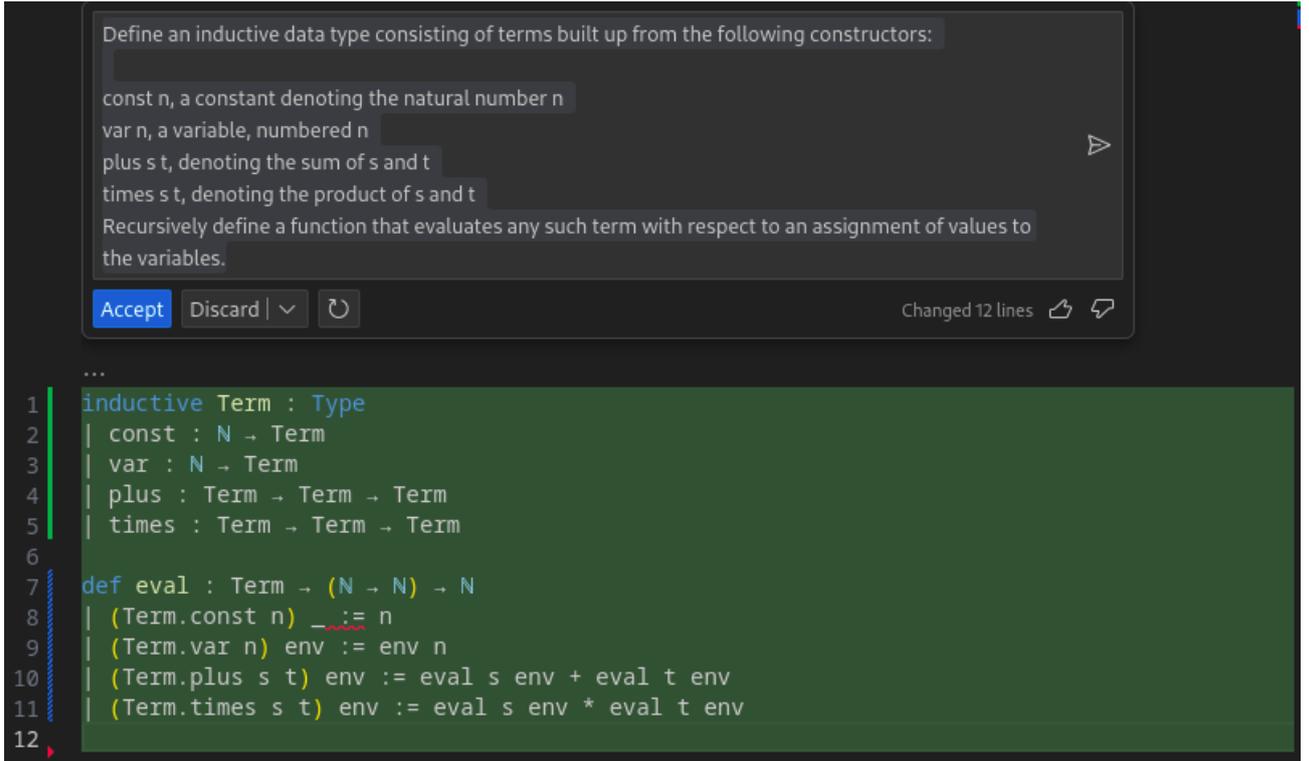
const n , uma constante que denota o número natural n

var n , uma variável numerada n

plus s t , denotar a soma de s e t

times s t , denotar o produto de s e t

Ainda dentro do exercício pede-se para definir recursivamente uma função que avaliará qualquer termo desse tipo em relação a uma atribuição de valores às variáveis. A Figura 32 tem a solução provida pelo assistente, perceba como há um erro sintático na solução na linha 8.



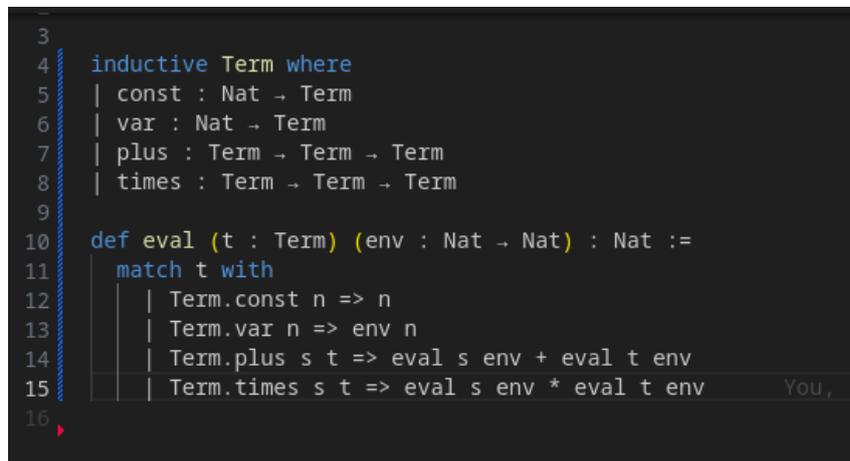
```

...
1 | inductive Term : Type
2 |   | const : N → Term
3 |   | var : N → Term
4 |   | plus : Term → Term → Term
5 |   | times : Term → Term → Term
6
7 | def eval : Term → (N → N) → N
8 |   | (Term.const n) _ := n
9 |   | (Term.var n) env := env n
10 |   | (Term.plus s t) env := eval s env + eval t env
11 |   | (Term.times s t) env := eval s env * eval t env
12

```

Figura 32 – Sugestão de solução do Copilot

A Figura 33 tem a minha solução em Lean 4.



```

3
4 | inductive Term where
5 |   | const : Nat → Term
6 |   | var : Nat → Term
7 |   | plus : Term → Term → Term
8 |   | times : Term → Term → Term
9
10 | def eval (t : Term) (env : Nat → Nat) : Nat :=
11 |   | match t with
12 |     | Term.const n => n
13 |     | Term.var n => env n
14 |     | Term.plus s t => eval s env + eval t env
15 |     | Term.times s t => eval s env * eval t env
16

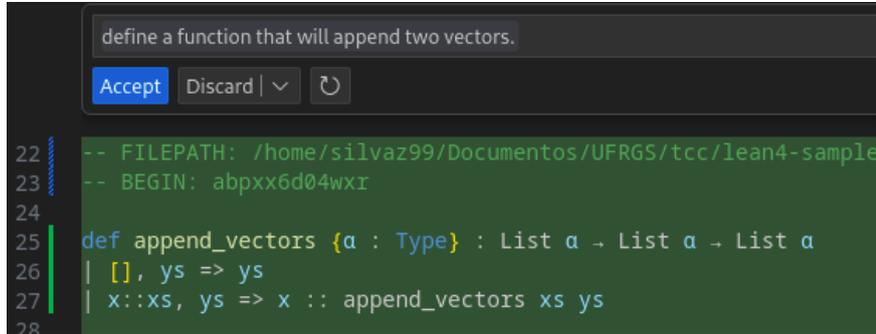
```

Figura 33 – Minha solução em Lean 4

4.5.5 Indução e Recursão

Lean nos fornece a possibilidade de definir funções recursivas, pattern matching e escrever provas indutivas, nos permite definir uma função especificando equações que ela deve satisfazer e nos permite provar um teorema por especificar como lidar com vários cases que podem aparecer [Avigad et al. 2024]. O nosso exercício que mostraremos nessa

seção é definir uma função que irá acrescentar um vetor à outro, a solução provida pelo Copilot está na Figura 34, a minha solução é praticamente idêntica então preferi omitir a imagem pela brevidade:



The image shows a code editor interface with a dark theme. At the top, there is a search bar containing the text "define a function that will append two vectors." Below the search bar are three buttons: "Accept" (highlighted in blue), "Discard" (with a dropdown arrow), and a refresh icon. The main area of the editor displays a code snippet in Lean 4. The code is as follows:

```
22 -- FILEPATH: /home/silvaz99/Documents/UFRGS/tcc/lean4-sample
23 -- BEGIN: abpXX6d04wxr
24
25 def append_vectors {a : Type} : List a → List a → List a
26 | [], ys => ys
27 | x::xs, ys => x :: append_vectors xs ys
28
```

Figura 34 – Sugestão de solução do Copilot

5 Discussão dos resultados

Neste capítulo vamos apresentar uma discussão da minha experiência utilizando o Copilot para desenvolver software em ReactJS e Lean. Começando a discussão pelo experimento com ReactJS, as respostas do assistente foram apresentadas no Capítulo 3. Apesar do projeto que foi criado ter sido um projeto relativamente simples, conseguimos utilizar o assistente a nosso favor, o fato do projeto ser pequeno acaba transformando a grande maioria do nosso código descomplicado o que facilita o assistente a entender o que queremos fazer desde o princípio.

Na maioria dos requests feitos ao assistente não havia a necessidade dele entender o contexto da aplicação, o que tornou o desempenho dele melhor. Quando foi pedido para que o assistente criasse testes unitários para a nossa aplicação, o assistente teve mais dificuldade em nos dar um código que funcionasse, pois a havia erros nos imports dos arquivos, isso é devido ao Copilot (pelo menos nesta versão utilizada) não conseguir extrair o contexto dos outros arquivos fontes do projeto.

Nas refatorações que foram solicitadas ao assistente, foi possível perceber ganhos de legibilidade no código melhorando assim a manutenibilidade.

Quando pedido para que o Copilot criasse novos componentes, dependendo da complexidade, o assistente foi bem sucedido, porém em alguns casos não foi produzido um código tão legível quanto poderia, o que nos faz questionar de certa forma se os dados que o Copilot utiliza no treinamento, nem sempre vão ser mais concisos e fáceis de entender, como nem sempre o código que ele me sugeria me satisfazia eu reformulava a minha frase para que o assistente tivesse mais possibilidades de me oferecer uma resposta melhor, o que nos leva ao questionamento de como Engenharia de prompt vai ser discutida nos próximos anos como uma forma de extrair o que esses modelos e assistentes têm a oferecer.

De forma geral, eu vejo como positiva a experiência tanto na busca para que o Copilot gerasse código do zero para mim, como no processo de refatoração de algum componente ReactJS. A documentação de métodos e de componentes criada pelo assistente foram boas pois ficaram bem detalhadas e completas.

Agora passando ao segundo experimento, vamos falar da minha experiência usando o Copilot e Lean na resolução de teoremas. Em que pese o fato do Copilot conseguir me explicar o código Lean com bastante precisão, tive mais dificuldades em usar o assistente. Primeiro porque o Copilot me sugere respostas em Lean 3 ou utilizando o modo tático, mesmo quando eu explicitamente pedia para que não o fizesse.

Isso acabou me gerando insegurança em relação ao uso da ferramenta na geração de código. Em vários requests feitos ao Copilot, me foi sugerido código que não compilava diretamente sem imports de bibliotecas específicas e na hora de pedir uma sugestão de como resolver,

o assistente não conseguia me ajudar sem que eu interviesse na suas sugestões, como por exemplo, sugerir imports dentro de funções, o que não é permitido dentro da sintaxe do Lean 4.

Ainda assim, foi positiva minha experiência usando o Copilot com Lean, porém acredito que a experiência teria sido mais positiva se eu pudesse ter usado o LeanCopilot [Yang et al. 2023], pois esta é uma ferramenta que já sugere ao seu usuário algumas táticas para serem usadas na construção da prova. O Lean Copilot porém, deve rodar localmente e ele exige um poder computacional que a minha máquina de uso pessoal não suporta.

6 Conclusão

Como este trabalho tem sido até aqui uma tentativa de entender mais como funciona e funcionará o desenvolvimento de software com a inserção dos assistentes baseados em IA, vamos finalizá-lo com uma reflexão sobre as implicações futuras dessas tecnologias e as possíveis transformações que elas podem trazer para o cenário do desenvolvimento de software, contribuindo assim para uma visão mais abrangente e informada sobre o papel crescente da inteligência artificial nesse contexto. No decorrer desta análise, mergulhamos na interseção entre a programação convencional e a inteligência artificial, examinando como esses elementos convergentes moldam não apenas o presente, mas também delineiam o horizonte do que está por vir.

Ao longo desta pesquisa, foram examinados casos de uso específicos, tanto em desenvolvimento mais tradicional - no sentido do trabalho de desenvolvimento de software -, como no campo da automação de provedores de teoremas, destacando as contribuições positivas e os possíveis obstáculos que os assistentes baseados em LLMs apresentam. A compreensão dessas nuances é essencial para a formulação de estratégias eficazes que maximizem os benefícios e minimizem os riscos associados a essa inovação.

Quanto ao impacto educacional, os assistentes fundamentados em LLMs evidenciaram-se como ferramentas preciosas para aprimorar o processo de aprendizado. Eles capacitam estudantes a explorar conceitos avançados de programação de maneira mais acessível. Contudo, com base nos experimentos conduzidos com o Lean, não considero essa abordagem como a mais adequada em minha perspectiva, uma vez que o assistente ainda apresenta confusões em determinadas solicitações.

Já para o aprendizado do framework ReactJS, o Copilot revelou-se uma ferramenta confiável, uma vez que forma raraz as vezes em que ele me respostas incorreta. É importante salientar que não foram realizadas várias iterações de solicitações para otimizar as respostas do assistente. Diante dessas considerações, acredito que o Copilot pode, sem dúvida, desempenhar um papel valioso no processo de aprendizagem do framework, proporcionando uma experiência consistente e eficaz.

Como foi mostrado tanto pelos experimentos como pelos artigos que li, estamos em um momento de desvendamento das implicações dessas ferramentas e estamos também em um processo de descobrimento de como utilizar esses tipo de ferramentas para nos auxiliar ou de como gerar insumos para nós.

E para nos despedirmos vale destacar por último, que além dos códigos a interação entre a criatividade humana e capacidade analítica dos assistentes em IA, revela um futuro no qual a colaboração entre homem-máquina redefine os paradigmas do desenvolvimento

de software, estabelecendo novos padrões de eficiência e inovação.

Referências

- AVIGAD, J. et al. *An introduction to Lean*. 2017. Disponível em: <https://leanprover.github.io/introduction_to_lean/>. 55
- AVIGAD, J. et al. *Lean Documentation*. 2024. Disponível em: <https://lean-lang.org/theorem_proving_in_lean4/>. 56, 64, 65
- CASTELLUCCIO, C. *Introduction to Generative AI and Large Language Models*. 2023. Disponível em: <https://microsoft.github.io/generative-ai-for-beginners/#/01-introduction-to-genai/README?wt.mc_id=academic-105485-koreyst&id=introduction-to-generative-ai-and-large-language-models>. 21
- CHATGPT. Disponível em: <<https://openai.com/chatgpt>>. 22
- CHRISTOPHER, J. *Paired Programming Redefined: The Rise of the AI Code Assistants*. 2023. Disponível em: <<https://cursivewriting.medium.com/paired-programming-redefined-the-rise-of-the-ai-code-assistants-a17536cc403>>. 23
- CLARKSON, M. R. et al. *OCaml Programming: Correct + Efficient + Beautiful*. 2024. Disponível em: <https://cs3110.github.io/textbook/ocaml_programming.pdf>. 51, 54
- CODELLAMA. Disponível em: <<https://github.com/facebookresearch/codellama>>. 19
- CODELLAMA Llama 2. Disponível em: <<https://about.fb.com/br/news/2023/08/apresentamos-o-code-llama-uma-ferramenta-de-ia-para-programacao/>>. 25
- COPILOT. Disponível em: <<https://github.com/features/copilot>>. 19
- HEAVEN, W. D. *How AI assistants are already changing the way code gets made*. 2023. Disponível em: <<https://www.technologyreview.com/2023/12/06/1084457/ai-assistants-copilot-changing-code-software-development-github-openai>>. 24
- HOU, X. et al. *Large Language Models for Software Engineering: A Systematic Literature Review*. 2023. 11, 27, 28
- JADAV, R. *Using GitHub Copilot Effectively*. 2023. Disponível em: <<https://medium.com/@rajesh.jadav/using-github-copilot-effectively-8b46604299ed>>. 23
- KERNER, S. M. *large Language Models*. 2023. Disponível em: <<https://www.techtarget.com/whatis/definition/large-language-model-LLM>>. 23
- LEAN. Disponível em: <<https://lean-lang.org/>>. 20
- LLAMA 2. Disponível em: <<https://llama.meta.com/llama2>>. 25
- MAITRA, I. *How AI Has Impacted Developer's Job*. 2023. Disponível em: <<https://browsee.io/blog/how-ai-has-impacted-developers-job/>>. 23
- PAIR programming. Disponível em: <https://en.wikipedia.org/wiki/Pair_programming#:~:text=Pair%20programming%20is%20a%20software,two%20programmers%20switch%20roles%20frequently.> 23

- PANDEY, A. *GitHub Copilot vs. AWS CodeWhisperer: A Comparative Analysis*. 2023. Disponível em: <<https://medium.com/@pandeyarpit88/github-copilot-vs-aws-codewhisperer-a-comparative-analysis-b8264a12124e>>. 13, 25, 26
- PENG, S. et al. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. 2023. 28
- RAJ, A. *GitHub: AI for developers boosts global productivity*. 2023. Disponível em: <<https://techwireasia.com/07/2023/github-ai-for-developers-boosts-global-productivity/#:~:text=As%20a%20matter%20of%20fact,get%20comfortable%20with%20the%20tool.>> 24
- SANDOVAL, G. et al. *Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants*. 2023. 28
- SCULLION, C. *Nearly 50 used at their workplace*. 2024. Disponível em: <<https://www.videogameschronicle.com/news/nearly-50-of-developers-say-generative-ai-tools-are-being-used-at-their-workplace/>>. 24
- TABNINE. Disponível em: <<https://www.tabnine.com/>>. 19
- TEAM, B. E. *The Impact of AI on Software Testing: Challenges and Opportunities*. Disponível em: <<https://www.bairesdev.com/blog/impact-ai-software-testing-challenges/>>. 19
- TEAM, S. *AI Code Assistants: Key Capabilities and 5 Tools to Know About*. 2023. Disponível em: <<https://swimm.io/learn/ai-tools-for-developers/ai-code-assistants-key-capabilities-and-5-tools-to-know-about>>. 22
- WHAT Is the Transformer Architecture and How Does It Work? Disponível em: <<https://datagen.tech/guides/computer-vision/transformer-architecture/#>>. 22
- YANG, K. et al. *LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*. 2023. 29, 68