Rodrigo Ferreira Araújo

Vantagens e Barreiras da IA Generativa como Ferramenta Auxiliar no Processo de Programação e Desenvolvimento de Software

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Pesquisa Científica

Orientador: Pedro Olmo Stancioli Vaz de Melo

Belo Horizonte, Minas Gerais
2025

Sumário

1 INTRODUÇÃO . . . 3

2 REFERENCIAL TEÓRICO . . . 4

3 METODOLOGIA . . . 5

4 CONTRIBUIÇÃO . . . 6

4.1 Entendimento de Código . . . 7

4.1.1 Using an LLM to Help With Code Understanding . . . 7

4.1.2 Theory of Mind for an LLM-Based Code Understanding Assistant . . . 9

4.2 Aspectos do Processo e Resultados da Geração de Código com LLMs . . . 12

4.2.1 A Review on Code Generation with LLMs: Application and Evaluation . . . 12

4.2.2 Calibration and Correctness of Language Models for Code . . . 14

4.2.3 Can LLMs Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation . . . 18

4.2.4 Towards more realistic evaluation of LLM-based code generation: an experimental study and beyond 22

4.3 Segurança do Uso de IA Generativa Para Código . . . 26

4.3.1 Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers . . . 26

4.4 LLMs em Depuração de Código . . . 29

4.4.1 DebugBench: Evaluating Debugging Capability of Large Language Models . . . 29

4.4.2 DebugEval: Enhancing the Code Debugging Ability of LLMs via Communicative Agent Based Data Refinement . . . 33

4.4.3 MDEVAL: Massively Multilingual Code Debugging . . . 37

4.5 Comparativo entre Linguagens de Programação . . . 40

4.5.1 A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages . 40

5 FECHAMENTO . . . 44

5.1 Discussão . . . 44

5.1.1 Fronteiras da Literatura . . . 44

5.1.2 Limitações . . . 45

5.2 Conclusão . . . 46

REFERÊNCIAS . . . 47

Resumo

Este estudo investiga as vantagens e os desafios do uso da Generative Artificial Intelligence (*GenAI*) como uma ferramenta de suporte na programação e desenvolvimento de software. Ferramentas como ChatGPT e GitHub Copilot transformaram rapidamente os fluxos de trabalho, oferecendo benefícios significativos como produtividade aprimorada, conclusão mais rápida de tarefas e melhor compreensão de bases de código complexas, especialmente para programadores novatos. Por meio de uma revisão abrangente de 20 pesquisas e 5 surveys de 2023–2024, esta pesquisa examina o papel dos grandes modelos de linguagem (LLMs) em tarefas como geração de código, depuração e análise de segurança. Embora os LLMs sejam promissores na automação de tarefas de codificação de rotina e na assistência ao uso de API, o estudo destaca desafios, incluindo correção de código inconsistente, riscos de vulnerabilidade e limitações no suporte de programação de baixo nível. Além disso, barreiras como a complexidade da engenharia de prompt e performances subótimas são abordadas. As descobertas ressaltam o potencial transformador dos LLMs, juntamente com limitações críticas que precisam ser resolvidas para obter resultados confiáveis e sustentáveis. Potenciais pesquisas futuras devem constituir uma caracterização mais abrangente e realista do uso de *GenAI* para codificação, uma vez que grande parte dos estudos revelam um foco excessivo na performance prática em escopos limitados. Este estudo, como forma de uma revisão de literatura em escala reduzida, fornece insights valiosos para desenvolvedores, educadores e pesquisadores que navegam no papel em evolução da *GenAI* na engenharia de software moderna.

1 Introdução

Ferramentas de inteligência artificial generativa, tanto de texto como de imagens, músicas e até vídeos, tomaram de assalto a atenção do mundo da tecnologia nos últimos dois anos. Plataformas como *ChatGPT* para geração de texto em uma abordagem de propósito geral, *Midjourney* para geração de imagens com foco em arte digital e o *GitHub Copilot*[1], que sugere inserções no código sensíveis ao contexto, de rápida evolução em meses e aplicações variadas, trouxeram grande impacto e levantaram múltiplos questionamentos nas suas respectivas áreas de aplicação, seja na produção de textos, imagens ou, como assunto deste trabalho, na geração ou explicação de código.

Nesse sentido, tanto estudantes de programação como programadores de todas as senioridades perceberam de imediato o ganho de produtividade com o uso de ferramentas de IA generativa em suas seções de programação. Principalmente com o uso do *ChatGPT* para sugestões de soluções em código e explicações em mais alto nível, bem como o *GitHub Copilot*, uma extensão de IDEs populares como *Visual Studio Code* que, durante o ato de programar, sugere a inserção de trechos de código (em bloco ou em apenas uma linha) com base em informações contextuais do método, classe, arquivo e até mesmo da *codebase* em questão.

Apesar do surgimento de diversas ferramentas de IA generativa de texto robustas nos últimos dois anos, [1] o *ChatGPT* segue em primazia definitiva (2,3 bilhões de acessos somente em março deste ano) em relação a demais ferramentas como a concorrente da *Google*, o *Gemini* (133 milhões de acessos em março/24), a IA *Poe* (43 milhões), outra ferramenta generativa de propósito geral que mistura diversas *engines* para produzir melhores resultados, *Perplexity AI* (40 milhões), *Claude* (32 milhões), *Deep AI* (31 milhões) e, por último, o *Copilot Chatbot* da Microsoft (26 milhões). Somado a isso, o crescimento da aplicação de inteligência artificial como um todo em soluções de negócios proporcionou um crescimento significativo da adoção da sua variante generativa no último, como indica [2], o uso de IAs generativas para pelo menos uma aplicação de negócio praticamente dobrou desde 2023: de 33% dos consultados na pesquisa em 2023 para 65% em 2024 reportaram uso de *Gen AI*.

Nesse contexto, a principal conclusão é: a aplicação de ferramentas de IA generativa tanto para aprendizado quanto para geração de código em diferentes linguagens de programação tornou-se abrangente em velocidade sem precedentes, de tal modo que diversas lacunas na literatura científica sobre *Gen AI* para código rapidamente se

[1] https://github.com/features/copilot

formaram a partir da identificação da necessidade de se investigar os impactos positivos e negativos desta prática nas bases de código e nos próprios estudantes e desenvolvedores de software.

2 Referencial Teórico

De início, o conhecido fórum de ajuda entre programadores *Stack Overflow* realizou um *survey* [3] com os usuários da plataforma de modo a compilar estatísticas acerca da experiência de programação dos desenvolvedores. Nesta edição do survey, uma seção dedicada ao uso de *Gen AI* foi incluída e trouxe alguns *insights* importantes:

1. 76% dos consultados (60907 respostas) no *survey* estão usando (61%) ou planejam (13%) integrar ferramentas de IA no processo de desenvolvimento, um aumento desde o último ano com 70%. Mais relevante, houve um aumento de 18% dos desenvolvedores que aplicam IA no processo de programação desde o ano passado: de 44% para 62%, o que mostra um relativo sucesso das ferramentas de *Gen AI* para codificação desde a consolidação das principais ferramentas em 2023-2024.

2. Acerca da impressão e sentimentos dos desenvolvedores consultados acerca da aplicação de *Gen AI*, 72% de todas as 45873 respostas indicaram que são favoráveis ao uso, sendo 23% muito favoráveis, 48% favoráveis e 18% indiferentes, sem alterações significativas entre o perfil de respostas de desenvolvedores experientes e iniciantes. Esta estatística apresenta uma leve queda desde a última consulta, onde 77% dos consultados estavam favoráveis ao uso de *Gen AI* durante o desenvolvimento, o que, segundo o *survey*, pode indicar um leve descontentamento com os resultados gerados com o uso.

3. Sobre os benefícios que acompanham o uso de ferramentas de IA no processo de codificação, 36894 respostas indicaram que:

    a) 82% dos programadores profissionais alegaram que ferramentas de IA *AI Tools* aumentaram sua produtividade, comparado com 73% dos programadores iniciantes que relataram o mesmo, de modo que configura o maior benefício.

    b) Com uma diferença mais significativa, 70% dos iniciantes reportaram que *AI Tools* permitiram uma aceleração no aprendizado comparado a 60% dos *devs* profissionais. Desse modo, iniciantes veem um maior benefício de aprendizado com o uso de *AI Tools* no desenvolvimento.

    c) Outra diferença significativa acerca dos benefícios entre iniciantes e profissionais trata-se do aumento de precisão semântica ao codificar (28% para profissionais e 36% para iniciantes) e do alívio da carga de trabalho ao codificar (24% para profissionais e 29% para iniciantes).

O *survey* do *Stack Overflow* apresenta, então, outros insights importantes da comunidade de *devs* profissionais e iniciantes acerca do uso de ferramentas de IA, de modo que as principais conclusões a partir das estatísticas indicam que desenvolvedores iniciantes confiam mais nos *outputs* da *Gen AI* do que profissionais, mas a comunidade como um todo se mantém cética e dividida quanto ao uso. Além disso, há um consenso entre os desenvolvedores da comunidade de que as ferramentas de IA correntes podem até ser boas em tarefas mais práticas, mas falham em atividades mais complexas. Por fim, dados são amostrados acerca das atividades profissionais atuais e futuras que incorporam e incorporarão o uso de IA, e questões sobre a eficiência e ética do uso das ferramentas de IA.

Demais pesquisas que avaliam a produtividade percebida e outros aspectos da experiência do usuário com o uso do *GitHub Copilot* apresentam resultados bastante positivos com poucas ressalvas. Em [4], uma *survey* foi realizada com diferentes tipos de usuários do *GitHub Copilot* para levantar estatísticas e *insights* interessantes que cercam o uso da extensão de IDE. Nesse sentido, dentre as descobertas, a pesquisa demonstra um ganho significativo em produtividade percebida entre desenvolvedores de todos os níveis de experiência, desde estudantes até *seniors* com mais de uma década de experiência.

A principal métrica avaliada para corroborar esta hipótese foi a taxa de aceitação de sugestões efetivamente mostradas pelo *Copilot*. Além disso, programadores com diferentes níveis de experiência se beneficiam de formas diferentes: desenvolvedores mais iniciantes perceberam um maior crescimento de produtividade e ritmo de codificação, uma melhora na qualidade do código, bem como aceitam mais sugestões do *Copilot* e programadores mais experientes filtram mais as sugestões em termos de quantidade e qualidade, mas usam o *Copilot* para trechos de código *boilerplate*, automação de processos rotineiros ou como auxílio para tecnologias desconhecidas.

De maneira geral, os programadores reconhecem a corretude inconsistente da extensão, mas é considerado um aspecto secundário à utilidade de uma sugestão como um pontapé inicial para a conclusão de um método/classe/arquivo mais complexo.

Por fim, o *survey* em [5] apresenta uma completa e extensiva revisão sistemática da literatura acerca do uso de *Large Language Models* (LLMs) para geração de código. A pesquisa inicialmente provém de uma introdução teórico-matemática ampla acerca do funcionamento de LLMs e seu uso para geração de código, acompanhada de uma visão cronológica da evolução das ferramentas LLM desde o ano de 2020. Segundo o *survey*, o objetivo principal é construir um material científico completo e robusto para preencher uma necessidade na comunidade científica de uma revisão da literatura abrangente e atualizada dedicada ao uso de LLMs para geração de código.

O *survey* orbita em três principais perguntas de pesquisa:

1. **RQ1**: Como categorizar e avaliar os avanços recentes dos LLMs para geração de código;
2. **RQ2**: Quais são os principais *insights* conclusivos acerca de geração de código via LLMs,
3. **RQ3**: Quais são os principais desafios e oportunidades promissoras de pesquisa no uso de LLMs para geração de código.

Um dos objetivos principais desta pesquisa científica é abordar a **RQ2** deste *survey* sob a perspectiva de múltiplos estudos e demais *surveys* acerca do uso de IAs generativas de texto para a produção de código. Nesse sentido, algumas das principais conclusões e perspectivas produzidas pelos autores do *survey* consistem na necessidade de:

1. Explorar oportunidades acerca da lacuna existente entre a comunidade acadêmica e desenvolvimento prático de software;
2. Melhorar a geração de código complexo em uma escala a nível de repositório e software;
3. Ajustar e tunar modelos e arquiteturas inovadoras (além dos LLMs baseados em *Transformers* para estruturas de código;
4. Curar dados de código de alta qualidade para pré-treinamento e *fine-tuning* de LLMs;
5. Suportar linguagens de programação com recursos limitados, baixo nível e com domínios mais específicos, e, por fim,
6. Garantir a segurança do código e alinhando os resultados do LLM com as preferências de codificação humana.

3 Metodologia

Neste cenário, este trabalho científico tem como proposta central caracterizar o cenário e estado-da-arte do uso de inteligências artificiais generativas de texto como principais auxiliares no processo de codificação profissional e de aprendizado. Sobre os objetivos centrais, a ideia principal é compilar e analisar informações e estatísticas

agregadas de estudos e *surveys* científicos que analisam o cenário de uso de *Gen AI* para desenvolvimento de software, de modo a reportar diversos aspectos como: performance percebida das IAs como copilotas de desenvolvedores de software durante o exercício da profissão; vantagens e desvantagens técnicas, estratégicas e de aprendizado dos resultados produzidos pela *Gen AI* durante a codificação; identificação de casos de borda onde a inteligência artificial atinge um limite operacional de maneira que os códigos ou *insights* produzidos não sejam produtivos, e, por fim, encontrar *gaps* teóricos e práticos que os *surveys* e demais estudos analisados não cobriram acerca da praticidade ou danos possíveis causados pelo uso da IA como copilota no processo de codificação.

De modo mais específico, este trabalho explorará de forma extensiva a literatura científica, com foco principal em *surveys* e quaisquer estudos que forneçam *insights* estatísticos interessantes e produtivos acerca dos resultados e impressões dos desenvolvedores com o uso de ferramentas de IA generativa para geração de código de diferentes linguagens de programação. Para atingir os objetivos de caracterização do cenário de uso de *Gen AI* para desenvolvimento de software descritos acima, os passos, descritos nos seguintes itens, serão seguidos:

1. Levantamento dos *surveys* que abordem diretamente ou de forma breve/tangente o tópico de uso de *Gen AI*/LLMs para geração de código;

2. Levantamento dos artigos no mesmo sentido do item anterior;

3. Similar a uma revisão sistemática, mas sob um escopo menor, compilar resultados estatísticos, quantitativos e qualitativos entre estudos e *surveys*, de modo a produzir uma caracterização de contexto mais geral acerca do uso de IA para geração de código;

4. Produzir uma análise qualitativa de contexto geral acerca das potencialidades positivas provenientes do uso de *Gen AI*/LLMs para produção de código;

5. Produzir uma análise qualitativa de contexto geral sobre os principais reveses oriundos do uso corrente de *Gen AI*/LLMs para SE (*software engineering*);

6. Produzir uma análise qualitativa de contexto geral acerca da performance e capacidades esperadas e percebidas das IAs por parte dos desenvolvedores de de software de diferentes níveis de senioridade.

7. Realizar uma análise inter estudos que considere o aspecto de ferramentas e linguagens de programação específicas: encontrar diferenças de aplicação e performance de diferentes ferramentas de IA a diferentes linguagens de programação em cenários diversos,

8. Por último, em uma seção de discussão, procurar identificar, nos estudo e *surveys* levantados, *gaps* de pesquisa, isto é, dado o rápido e surgimento e estabelecimento das ferramentas de IA no cenário de desenvolvimento de software, quais questões ainda não foram abordadas/tratadas?

4 Contribuição

Para esta proposta de revisão da literatura, foram compilados 20 artigos científicos e 5 surveys primariamente de 2023 e 2024. A temática dos estudos selecionados tanto aborda centralmente o tema de uso de geração de código com LLMs quanto trata de questões tangenciais e expansivas ao tema. Conforme metodologia proposta, as seguintes seções estão centradas em fornecer uma visão compreensiva do cenário de uso de LLMs para produção, avaliação, revisão ou quaisquer outras tarefas que abordem código, por meio das frentes de benefícios/barreiras intrínsecas à IA generativa, refletidas por estatísticas bem como aquelas percebidas pelos programadores de diferentes ramos e níveis de experiência. Aliado a isso, a seção de performance discute aspectos de desempenho medido estatisticamente e também percebido pelos desenvolvedores em diferentes cenários de aplicação dos LLMs.

4.1 Entendimento de Código

4.1.1 Using an LLM to Help With Code Understanding

Introdução

Como artigo primordial acerca do tópico de entendimento de código via inteligência artificial, este paper[6] apresenta uma perspectiva empírica ao aplicar e analisar o desempenho de uma ferramenta de inteligência artificial conversacional, baseada em queries para o LLM GPT-3.5-turbo da OpenAI, como um plugin autoral de IDEs: GILT (Generation-based Information-support with LLM Technology), que prioriza interações *prompt-less*, ou seja, prompts pré-programados para determinados contextos, bem como é sensível ao contexto do código corrente trabalhado.

O estudo reforça referências que apontam para o fato de que desenvolvedores de software gastam um tempo significativo organizando e pesquisando informações, cruzando referências nem sempre completas como *websites*, documentações ou diretamente no repositório de trabalho ou estudo, com o objetivo de capturar as ideias principais de uma *codebase*. como as complexidades das APIs usadas, conceitos específicos de domínio, entre outros aspectos. Contudo, a conveniência da alta velocidade de produção de código dos LLMs de larga escala (*chatGPT* e afins) não vem de graça, uma vez que muito do gargalo mental foi transferido para o esforço das técnicas de engenharia de *prompt*, para contextualizar o *chatbot* acerca da solução desejada via código, bem como para o tempo dedicado à depuração do código produzido pelos LLMs, somado ao esforço mais repetitivo de mudança de contexto entre o ambiente conversacional e o de escrita de código.

Com isso, em um experimento exploratório da ferramenta proposta GILT, com 32 participantes, os pesquisadores compararam o ritmo de conclusão de tarefas em uma *codebase* específica bem como o nível de entendimento de código dos participantes sob duas condições: usando o GILT na IDE *versus* pesquisas na web. Por fim, uma análise qualitativa foi realizada para caracterizar o uso das ferramentas do GILT pelos participantes e os respectivos níveis de satisfação. Nesse contexto, constam as perguntas de pesquisa: **RQ1**: Até que ponto a ferramenta proposta afeta o entendimento dos desenvolvedores, tempo/ritmo de conclusão de tarefas em uma *codebase* não familiar, **RQ2**: Como desenvolvedores de diferentes senioridades interagem com o GILT e **RQ3**: Como os desenvolvedores percebem a utilidade do GILT.

Os trabalhos relacionados citados reforçam a variedade de problemas enfrentados pelos desenvolvedores ao compilar de forma mais acessível informações de uma *codebase* via documentação, mas principalmente realça artigos que investigaram de forma empírica a eficiência de ferramentas LLM em completar plenamente tarefas de desenvolvimento de software e em explicar código existente e produzido, onde os resultados mostram que tais ferramentas podem ser razoavelmente úteis para estudantes e programadores com certas ressalvas. Contudo, demais referências apontam evidências de que dificilmente há um ganho real de produtividade, tanto em tempo como em corretude, para tarefas mais complexas. Por fim, um insight importante revela que há dois modos predominantes de uso de LLMs para escrita de código (especificamente investigado com o GitHub© Copilot): acelerador, onde a ferramenta é usada para completar código em pequenos pedaços lógicos e explorador, onde o LLM é usado como assistente de planejamento da estrutura de chamadas de API e demais componentes.

As funcionalidades principais da ferramenta proposta GILT, são:

1. **Foco no entendimento**: Ausência proposital de uma função de completar diretamente o código para mirar o foco em como os desenvolvedores absorvem e extendem o código;

2. **Extensão nativa na IDE**: Facilita o fornecimento do contexto da *codebase* para o modelo de linguagem;

3. **Prompts pré-programados**: Interação *prompt-less* via botões que enviam queries automatizadas para o LLM com perguntas relacionadas a APIs, explicações conceituais ou exemplos de uso, com o objetivo de aliviar a carga cognitiva associada à formulação de prompts compreensivos pelos usuários, especialmente para programadores mais novatos,

4. **Queries textuais irrestritas**: Por fim, em uma interface conversacional, os participantes também puderam escrever seus prompts livremente, de modo que quaisquer trechos de código selecionados são transferidos automaticamente para o prompt com templates pré-definidos.

Com um grupo de tratamento e controle, onde respectivamente os participantes usaram exclusivamente o GILT para realizar as tarefas propostas e usaram quaisquer outras ferramentas de pesquisa disponíveis na internet, o experimento humano tinha como tarefa entender uma *codebase* comum em *Python* e realizar tarefas e sub-tarefas relacionadas a um domínio comum e outro menos comum: visualização de dados e renderização 3D (bibliotecas Bokeh[7] e Open3D[8], respectivamente).

Resultados

No que tange a RQ1, o mapeamento quantitativo realizado pelo estudo revelou que participantes que usaram a ferramenta proposta completaram moderadamente mais sub-tarefas que aqueles que usaram apenas *search engines* convencionais. Por outro lado, mostrou-se que não há ganho significativamente observável de compreensão ou velocidade de conclusão de tarefas entre aqueles que usaram o GILT contra os participantes que não usaram.

Ademais, dividindo os participantes em dois grupos baseados em suas ocupações declaradas: estudantes ou profissionais, e, embora a ferramenta tenha sido eficaz para ambos os grupos, houve um impacto significativamente maior no grupo dos profissionais, de modo que completaram 0,57 mais sub-tarefas com o uso do GILT comparado com profissionais que usaram apenas *search engines*, ao passo que os estudantes não viram um ganho significativo para a mesma comparação.

Acerca da RQ2, 97% dos participantes responderam que questionar diretamente o modelo de linguagem era de extrema importância, mas que a utilidade reportada das interações *prompt-less* variou bastante. De forma

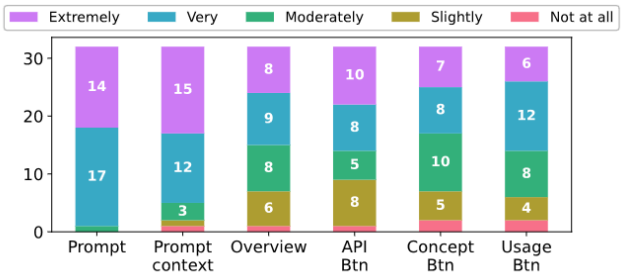


Figura 1 – Tipos de funcionalidades usadas

contraditória, 75% das sub-tarefas completadas pelo grupo GILT se originaram a partir de interações via *prompt* mas também 83% das tarefas não sucedidas também foram precedidas por interações via *prompt*, impossibilitando uma correlação mais concreta entre o uso da ferramenta e um ganho informacional.

Outro resultado interessante marca mais a diferença de interação entre o grupo de profissionais e estudantes: 86% das sub-tarefas bem-sucedidas dos profissionais originaram via *prompt* direto, comparado a 62% dos estudantes. A taxa de sucesso de interações *prompt-based* também foi maior entre os profissionais comparado aos estudantes, mas, de forma contrária, a taxa de sucesso usando a feature de overview e demais botões *prompt-less* foi menor (56%) do que a dos estudantes no mesmo cenário (85%), o que pode indicar que, mediante uma menor experiência de codificação e, por conseguinte, menor tato na procura de informações para desenvolvimento de software, os estudantes tiveram uma dificuldade maior em formular *prompts* efetivos e especializados comparado com os profissionais, de modo que confiaram mais em uma interação *prompt-less*.

Nesse contexto, por meio de uma análise do vocabulário usado durante os *prompts* para o GILT, os pesquisadores revelaram que estudantes usaram palavras-chave mais genéricas e/ou palavras que tinham um conceito marginalmente semelhante ao pedido pela tarefa, bem como tiveram mais dificuldade em revisar os próprios *prompts* quando comparados aos profissionais. Por fim, nesse cenário, outros mapeamentos indicaram que desenvolvedores mais familiarizados com ferramentas de IA para codificação tendem a usar mais *prompts* diretos

com linguagem natural, contudo, não houve diferença observacional significativa acerca da diferenciação de uso com base em diferentes estilos de aprendizado e/ou formas de busca de informações.

Sobre a percepção dos usuários, na RQ3, testes conhecidos TAM (*Technology Acceptance Model*) e TLX(*Task Load Index*), que endereçam a aceitação de usuários a novas tecnologias bem como medidas subjetivas de carga mental ao usar a ferramenta, foram aplicados aos participantes. Desse modo, revelou-se que, no que tange a utilidade e facilidade de uso percebidas, usuários do GILT reportaram esses índices de usabilidade significativamente maiores que o grupo controle. Para aspectos de esforço mental dedicado, usuários do GILT reportaram, de forma significativa, menor demanda temporal e mais performance percebida, o que, segundo os pesquisadores, indica que os desenvolvedores se sentiram menos apressados com o GILT em mãos comparado às *search engines*, bem como se sentiram mais bem-sucedidos ao completar as tarefas com o LLM.

Com a coleta de feedback dos participantes acerca da ferramenta proposta, descobriram que um dos aspectos mais benéficos considerados foi a incorporação de contexto do código na funcionalidade do LLM, o que permitiu uma melhor guia do modelo de linguagem e suas sugestões para se atentarem melhor ao contexto de programação com queries mais complexas. Por mais, diversos participantes relataram que *search engines* convencionais como *Stack Overflow* ou uma pesquisa no Google genérica exigiriam mais esforço e tempo quando comparado à procura do mesmo problema em um contexto mais especializado no LLM. Entretanto, importante destacar que ainda há uma barreira cognitiva associada a transferir a racionalização de um problema para um *prompt* de qualidade em linguagem natural, de modo que, combinado com as limitações do LLM, alguns participantes reportaram que as respostas fornecidas pelo GILT eram por vezes imprecisas, prejudicando a produtividade nas tarefas.

Discussões Relevantes

No que tange o aspecto de compreensão de uma *codebase*, um comportamento interessante foi observado entre os participantes que utilizaram a ferramenta proposta: "terceirização"da compreensão. Nesse sentido, alguns participantes transferiram a carga de entendimento do código analisado para a ferramenta, de modo que realizaram uma engenharia de *prompt* mais superficial/alto-nível em prol da produtividade nas tarefas sem leitura/compreensão completa do código a ser alterado/extendido, o que corrobora com as evidências em [9] acerca da inclinação de desenvolvedores em focar na conclusão de uma tarefa ao invés de compreender o software. Esse processo de transferência de compreensão de código para o LLM, ainda que apenas observado em laboratório, erige questões acerca da qualidade do código resultante desse processo.

4.1.2 Theory of Mind for an LLM-Based Code Understanding Assistant

Introdução

Similarmente ao estudo anterior, com o auxílio de uma LLM estabelecida (GPT-3.5), os pesquisadores desenvolveram uma ferramenta conversacional baseada em IA generativa ToMMY (Theory of Mind: Making explanations Yours), que consiste em uma cadeia de *prompts* especializados a capturar o estado mental do desenvolvedor para prover explicações de software melhor adaptadas a diferentes programadores. A premissa da ferramenta é tentar capturar nuances do que os pesquisadores chamam de estado mental, conforme a Teoria da Mente (capacidade de raciocinar acerca do que outros estão pensando), de aspectos como conhecimento prévio, familiaridade com o tópico ou experiência em programação, de modo a mitigar os efeitos de respostas incompletas dos LLMs, provenientes da dificuldade de expressar as exatas necessidades em linguagem natural via *prompts* encontrada pelos programadores, especialmente iniciantes.

Como um estudo pioneiro na aplicação e avaliação de engenharia de *prompt* relacionadas a Teoria da Mente em suporte de codificação via LLM, com um estudo humano controlado, os pesquisadores objetivaram responder as seguintes perguntas de pesquisa: **RQ1**: Como iniciantes em programação interagem com assistentes baseados em LLMs no que tange ao entendimento de código novo?; **RQ2**: Como a ferramenta proposta ToMMY impacta o entendimento de código dos iniciantes em programação?, **RQ3**: Como novatos avaliam a interação com assistentes baseados em LLMs para entender código não familiar?. Portanto, como foco na personalização da experiência conversacional com LLMs dos desenvolvedores, o estudo em questão reuniu 14 pessoas consideradas iniciantes em

programação, desde estudantes de graduação e mestrado/pós, de modo que cada um interagiu individualmente com o agente proposto ToMMY e outro agente conversacional básico (GPT-3.5) para efeitos de comparação. De forma antecipada, o estudo revela que programadores com diferentes estilos de interação com a ferramenta conversacional obtiveram resultados distintos, principalmente entre aqueles que formularam suas perguntas em forma de uma hipótese ("*Does X do Y*?") ou não. Além disso, poucos programadores explicitaram a intenção de suas interações ou escreveram perguntas em sequência em um mesmo chat, o que se mostrou bastante contributivo para o entendimento de código.

O design da ferramenta ToMMY levou em consideração aspectos importantes na literatura citada pelo artigo como: a necessidade de identificar e adaptar assistentes de programação aos pontos fortes e necessidades do usuário em questão, de modo a personalizar as respostas de acordo; a predição e reflexão (*perspective-taking*) acerca do estado mental dos desenvolvedores melhora o desempenho de LLMs de maneira geral, e a necessidade de se separar inferência de estado mental de resposta a questionamentos em diferentes *prompts*. Portanto, a ferramenta ToMMY consiste em três *prompts* principais:

1. Primeiro, por meio de questões abertas ao programador(a), o LLM é instruído a identificar aspectos do estado mental dele(a) relevantes para a produção de respostas futuras;

2. Em seguida, o LLM é instruído a tomar a perspectiva do usuário analisado para então responder as perguntas do *prompt* anterior, baseado no histórico de chat,

3. Por fim, com os dois primeiros *prompts* constituindo o estado mental do participante, o terceiro é uma instrução para a LLM responder à questão do(a) programador(a), considerando este estado mental.

Cada *prompt* da ferramenta ToMMY é acompanhado de contexto para o LLM, constituído de um trecho de código a ser analisado, o histórico de chat e o último input do usuário, ao passo que o agente controle (GPT-3.5) consiste em apenas um *prompt* com o mesmo contexto do anterior e uma instrução para responder ao usuário. Acerca dos participantes recrutados, suas experiências em programação convergiram em uma média de 5.7 (em uma escala de 1: novato a 10: experiente), com experiências variadas de uso de LLMs.

Resultados

De início, participantes não interagiram por mais tempo com nenhuma das duas ferramentas, de modo que a média e o desvio padrão do número de interações permaneceram semelhantes para ambos ToMMY e o modelo controle. Na análise qualitativa a respeito da RQ1, os pesquisadores agruparam os inputs dos participantes em 5 elementos de interação com o chatbot:

1. **Perguntas**: Em todas as 201 interações gravadas, exceto 4, houve a presença de uma pergunta, mas apensa um input continha duas perguntas, fraseadas ou de forma aberta ("O que X faz"?), como uma **Hipótese** ("X faz Y?"), como uma instrução ("Explique X") ou de forma implícita (provendo apenas uma linha de código para o modelo explicar). Outro cluster foi realizado para distinguir participantes que frasearam pelo menos uma questão em forma de hipótese e encontraram que 6/14 participantes o fizeram. Nesse contexto, participantes que frasearam suas perguntas em forma de hipótese interagiram significativamente por mais tempo com a ferramenta proposta comparado aos participantes que não usaram hipóteses. Por fim, a diferença entre experiência de programação e familiaridade com LLMs não surtiu efeito observável.

2. **Alvo das Perguntas**: O objeto de questionamento dos participantes variou desde um aspecto elementar do código (sintaxe, uma variável, uma ou várias linhas de código), todo o trecho de código, uma feature externa (função ou biblioteca, ou um conceito de programação. Com isso, participantes que formularam suas perguntas como hipóteses tenderam a fazer mais perguntas envolvendo todo o trecho de código quando comparado aos que não usaram hipóteses, que, por sua vez, objetivaram suas perguntas para aspectos mais específicos do código.

3. **Intenção das Perguntas**: A maioria das questões feitas não especificaram qual aspecto do código para explicação do agente, outrora fornecido de forma implícita. Nesse sentido, 30% das perguntas feitas para o agente controle continham intenção explicitada na pergunta comparado a 25% do agente ToMMY. Ademais, essa mesma observação foi feita para aqueles que formularam pelo menos uma hipótese: 29% contra aqueles que não formularam hipóteses: 26%.

4. **Aspectos da Conversa**: Características adicionais de estilos de conversa foram capturados pelos pesquisadores como: estender a resposta inicial com mais uma pergunta ou escrita mais educada. 15% dos participantes que usaram hipóteses aplicaram perguntas em sequência contra 10% daqueles que não usaram hipóteses.

5. **Instruções**: Em algumas instâncias, participantes proveram instruções de como a resposta deveria ser formatada: curta, passo-a-passo, resumida, etc.

Acerca da RQ2, os pesquisadores utilizaram quizzes para endereçar o entendimento do código proposto após a utilização das ferramentas e extraíram variáveis observadas em um modelo de regressão linear que revelou aspectos como: apenas experiência prévia em programação e familiaridade com LLMs impactaram de forma significativamente observável o entendimento de código dos participantes, bem como uma maior experiência em programação permitiu que alguns participantes completassem os quizzes mais rapidamente e com pontuações mais altas, ao passo que a afirmação reversa não é verdade para familiaridade com LLMs, o que indica a significância do efeito de retroalimentação positiva entre o aprofundamento do usuário e o aprofundamento os LLMs.

Nesse sentido, outra observação importante foi a de que a ferramenta ToMMY contribuiu positivamente para a pontuação nos quizzes dos participantes que não frasearam suas perguntas como hipóteses, e impactou negativamente a pontuação daqueles que o fizeram, apesar de ter um *p-value* bem próximo de 0.05.

Afinal, no que tange a RQ3, de forma similar ao estudo anterior, utilizaram métricas TLX (Task Load Index) e TAM (Technology Acceptance Model), mas não mostraram diferenças significativas entre o modelo proposto e o de controle. Destaca-se, entretanto, a média baixo e alto desvio padrão para o item esforço do TLX para o ToMMY comparado com uma média maior e um menor desvio padrão para a ferramenta controle, o que pode indicar uma redução de carga cognitiva ao usar a ferramenta proposta, mas este efeito variou bastante por participante.

Sobre o **feedback** dos usuários, foi reportado que a ferramenta proposta, marginalmente, entendeu melhor as perguntas dos participantes bem como as respondeu de forma mais clara. Alguns participantes notaram que tiveram que reescrever as perguntas ou fornecer mais detalhes para que o agente controle entendessem o objetivo de seus questionamentos, bem como outros voluntários reportaram que às vezes não entendiam as respostas providas pelo agente de controle. Outro aspecto importante foi fato de que a ferramenta ToMMY produziu respostas ligeiramente menores que a ferramenta controle, especialmente quando as perguntas eram fraseadas como hipóteses, o que, segundo os pesquisadores, sugeriu que ToMMY seria melhor para reconhecer quando usuários estão apenas buscando validar sua interpretação do código. Nesse contexto, apesar de perceber as respostas do ToMMY como menores e mais concisas, os participantes reportaram preferir as respostas longas e e formatadas do agente de controle.

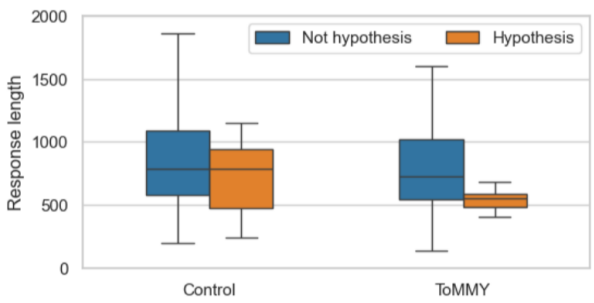


Figura 2 – Comprimento de Respostas do ToMMY Significativamente Menor para Perguntas Fraseadas como Hipóteses

Discussões Importantes

Os participantes exibiram duas formas principais de abordagem de compreensão ao usarem os modelos de linguagem: **top-down**: onde formularam hipóteses comumente associadas a features de mais baixo nível do código com várias perguntas em sequência a um mesmo chat; **bottom-up**: perguntas mais abertas, às vezes formuladas como instruções e com foco em todo o trecho de código analisado. Outros aspectos positivos da ferramenta proposta consistiram em entender melhor as entrelinhas das perguntas dos participantes que não explicitaram suas intenções ou que não fizeram perguntas em sequência, o que apontou para um melhor reconhecimento dos objetivos verdadeiros dos *prompts* dos iniciantes em programação e uma produção de respostas mais personalizadas de acordo com suas experiências. Contudo, as respostas normalmente formatadas em um único parágrafo como uma adaptação ao tipo de pergunta dos participantes pode introduzir problemas de usabilidade ao cortar detalhes que poderiam ser cruciais no processo de entendimento de código. ToMMY teve um impacto positivo nas pontuações dos questionários para os participantes com um estilo de compreensão *bottom-up* e um efeito negativo para participantes que exibem um estilo de compreensão de *top-down*. Em suma, de acordo com o observado, LLMs podem personalizar respostas de forma independente de alguns aspectos mas necessita de orientação para outros.

4.2 Aspectos do Processo e Resultados da Geração de Código com LLMs

4.2.1 A Review on Code Generation with LLMs: Application and Evaluation

Publicado em 2023, esse estudo[10] buscou fornecer um *overview* da literatura até então (similar a este próprio projeto de pesquisa) acerca dos aspectos mais importantes que consideraram sobre geração de código com LLMs, com foco também na avaliação dos LLMs em múltiplas frentes, contrastando com os estudos até então que focavam em suas aplicações, especialmente corretude dos códigos gerados. De forma semelhante a esse estudo, embora de forma mais enxuta, a ideia é fornecer uma visão geral do tópico durante um dos períodos iniciais de efervescência e evolução dos LLMs, principalmente no contexto de geração de código.

Aplicação

Com base na literatura revisada pelo estudo em questão, os pesquisadores elencaram 3 aplicações importantes a serem exploradas entre os estudos e discutidos: **Linguagem Natural para Código**, **Conclusões/Preenchimento de Código** e **Conserto Automático de Código**:

1. **Descrição Textual para Código**: Em primeiro lugar, um estudo publicado em 2022 aplicou o Codex, um modelo retreinado do GPT-3 em grandes repositórios de código aberto do GitHub focado em transformar texto em linguagem natural para código (modelo seminal para a construção do GitHub Copilot), para responder 23 perguntas introdutórias de programação em Python e revelaram que Codex performou melhor que a maioria dos estudantes que participaram nos testes, com soluções variadas e mais completas. Outro estudo com usuários aplicou uma ferramenta especializada para código chamada GenLine, baseada no então recente LaMDA[11] (Language Model for Dialogue Applications) da Google. Os resultados indicaram que, embora a síntese de código em linguagem natural possa às vezes proporcionar uma experiência mágica, os participantes ainda enfrentaram desafios. Em particular, os participantes sentiram que precisavam aprender a "sintaxe" do modelo, apesar de a sua contribuição ser a linguagem natural. Os participantes também se esforçaram para formar um modelo mental preciso dos tipos de solicitações que o modelo pode traduzir de forma confiável e desenvolveram um conjunto de estratégias para depurar o input do modelo. Por fim, um artigo desenvolveu um modelo "auto-colaborativo"com três instâncias do ChatGPT instruídos a atuarem como analista, escritor e testador de código durante um processo de desenvolvimento. Por meio de um método compreensivo, encontraram que a performance desse método colaborativo entre instâncias do GPT foi 30% maior que uma abordagem mais direta com o modelo de linguagem.

2. **Conclusões/Preenchimento de Código**: Funcionalidades de conclusão de código realizam sugestões de preenchimento de código integrado em IDEs com base no contexto do arquivo atual. O grande avanço dos

LLMs, nesse contexto, foi elevar as sugestões de um nível apenas sintático para predições mais completas a nível semântico com base em código e comentários presentes no arquivo. Nesse sentido, um estudo ("The Potential of Artificial Intelligence as a Method of Software Developer's Productivity Improvement") aplicado com a ferramenta GitHub Copilot mostrou que o modelo de linguagem pode ser consistentemente positivo em auxiliar tanto programadores experientes como iniciantes, embora, na versão analisada, encontraram alguns gaps como código de baixa qualidade não revisado, oportunidades de melhoria em segurança, confiabilidade e eficácia. Outro estudo de caso com o Copilot focou em estudar o seu impacto na produtividade dos usuários, que, por sua vez, se mostrou relativamente positivo (variando de acordo com o comportamento do desenvolvedor), com uma certa correlação positiva entre a taxa de aceitação das sugestões de código do LLM e a produtividade percebida dos programadores.

3. **Reparação Automática de Código e Correção de Bugs**: Inicialmente, em uma abordagem empírica, quatro LLMs (três versões do PolyCoder[12] e o Codex) foram testados em 80 programas contendo bugs. Os resultados indicaram que tais modelos foram mais bem sucedidos ao gerar consertos (patches), bem como tenderam a gerar reparações similares a um desenvolvedor humano. Em outros dois estudos, um benchmark de bug fixing em Python foi aplicado ao GPT-3.5 de forma mais direta e uma ferramenta conversacional completa, também em GPT-3.5, com uma abordagem de APR (Automatic Program Repair). Estas abordagens, não somente superaram em performance ferramentas tradicionais de APR como também desempenharam a nível estado-da-arte no quesito.

Avaliação

Conforme seção anterior, os pesquisadores agora abordaram três principais frentes de avaliação da geração de código via LLMs na revisão da literatura feita: **Corretude Funcional**, **Segurança** e *Outros*, discutidos em menor escala:

1. **Corretude Funcional**: Para avaliar o desempenho de uma abordagem recém-proposta de geração de código e compará-la com outras existentes, uma métrica adequada de avaliação de código (CEM - Code Evaluation Metric) é indispensável para o desenvolvimento de técnicas de geração de código. Existem, principalmente, dois tipos de CEMs predominantes: baseados em casamento (match-based) de padrão e baseados em execução (execution-based). Métricas de avaliação match-based comumente usadas em tradução automática podem ser úteis para avaliar a similaridade entre sentenças geradas e sentenças de referência, como BLEU e CodeBLEU[13]. No entanto, esses CEMs match-based apresentam deficiências em refletir a corretude funcional do código gerado, pois não conseguem determinar a equivalência funcional entre o código gerado e o código de referência. Por isso, mais CEMs baseados em execução foram propostos por pesquisadores posteriores. Atualmente, há duas CEMs baseadas em execução mais utilizadas para avaliar a correção funcional do código: AvgPassRatio (taxa média de aprovação em casos de teste) e Pass@k (para cada prompt, são gerados k snippets de código, se pelo menos um dos snippets de código estiver correto, o modelo foi bem-sucedido), ambas dependem da execução do código gerado em um conjunto de testes preparado. Em um estudo, consideraram que o cálculo de CEMs baseados em execução é oneroso, lento e inseguro, enquanto CEMs match-based são imprecisos. Por isso, propuseram um novo CEM baseado em LLM — o CodeScore, para avaliação de correção funcional. Eles definiram o PassRatio com base no AvgPassRatio e fizeram do CodeScore uma alternativa sem execução ao PassRatio. Ao avaliar com sua estrutura baseada em LLM, o CodeScore supera os CEMs match-based em precisão, quando comparado ao PassRatio, e custa muito menos tempo do que os CEMs baseados em execução.

2. **Segurança**: O grande poder dos LLMs deriva principalmente da enorme disponibilidade pública de repositórios e trechos de código em endereços como GitHub e outros para treinamento em massa dos modelos. Segue-se, claramente, que há grandes chances de que programas com vulnerabilidades ou de atuação maliciosa possa direcionar o output dos modelos de linguagem durante a geração de código. Nesses sentido, um dos estudos mencionados baseou-se nas CWEs (Common Weakness Enumerations) da MITRE[14] para

criar um conjunto de dados baseados em prompts que continha cenários relevantes a segurança de código. Desse modo, em 44% desses cenários, a ferramenta GitHub Copilot gerou código com uma vulnerabilidade significativa, com algumas introduzidas mais frequentemente que outras. Ademais, um estudo com usuários se propôs a descobrir se programadores em formação escrevem ou não códigos mais vulneráveis com a ajuda de um assistente baseado em LLM. Com uma abordagem baseada em projeto, mais encontrado no cenário real, descobriram que o número de bugs de segurança introduzido com o auxílio de LLMs foi por volta de 10% maior que o grupo controle. Estudos subsequentes foram assertivos e ratificaram o efeito de introdução de vulnerabilidades de código gerado via chatbots LLM, confirmando a insubstituibilidade de desenvolvedores experientes para detecção de vulnerabilidades.

3. **Outros**: Acerca da manutenibilidade e legibilidade dos códigos produzidos com *GenAI*, um insight interessante evidencia que o conjunto de code smells encontrados nos códigos gerados constitui um subconjunto dos code smells do código de treinamento. Já sobre a compreensibilidade do código gerado, um estudo empírico com o GitHub Copilot em 2022 avaliou códigos gerados para soluções do LeetCode, uma plataforma conhecida de resolução de problemas de programação, em termos de complexidade ciclomática e cognitiva. Assim, embora seja de um escopo reduzido, o estudo revelou que as soluções tiveram baixa complexidade independentemente da linguagem de programação, apesar das baixas taxas de corretude na solução dos problemas.

Fechamento

Em suas seções de discussões finais, o estudo destaca gaps importantes para pesquisa futura, que foram explorados em sua maioria por papers futuros no biênio 2023-2024. Dentre eles, estão: foco primordial na corretude funcional dos códigos gerados por IA, negligenciando fatores tangentes como compatibilidade, manutenibilidade, portabilidade, entre outros; a falta de um modelo de avaliação sistemática e quantitativa, futuramente provido por *surveys* acerca do tópico, e, por fim, falta de estudo e caracterização da interação humana com os modelos conversacionais, especialmente no escopo de avaliação de estratégias de *prompt* e outros comportamentos que impactam no código gerado. Este aspecto seria abordado em estudos de caso aplicados com análises qualitativas de interação com humanos.

4.2.2 Calibration and Correctness of Language Models for Code

Introdução

Publicado em agosto de 2024, este estudo[15] explora a qualidade dos código gerados via IA generativa sob o escopo da calibração do output, ou seja, se a medida de confiança associada ao modelo, uma informação normalmente oriunda do treinamento do modelo, é fortemente correlacionada com a probabilidade de corretude do código gerado, então o output produzido é considerado bem calibrado. Desse modo, o estudo aponta inicialmente que: **1)**: como os LLMs podem cometer erros, os usuários se beneficiariam de uma indicação verossímil de confiança de que o código gerado está realmente correto, de modo a tornar mais permeável a análise humana pelo código sintetizado pelo modelo de linguagem. **2)**: Tais indicações de confiança, quando bem alinhadas com a correção real, podem dar suporte a medidas de controle de qualidade de software com camadas de redundância baseadas na robustez do código gerado e, assim, melhorar a confiabilidade do ecossistema AI4SE (augmented intelligence and machine learning techniques to support systems engineering practices).

Nesse sentido, o artigo propõe um estudo empírico da calibração de modelos de geração de código usando várias técnicas anteriores e explora abordagens para melhorar ainda mais a calibração. Portanto, descreveram uma framework de avaliação para a calibração de modelos de linguagem de geração de código. Tal framework foi instanciada para diferentes tarefas, por exemplo, síntese de código, conclusão de código e reparo de programa, usando diferentes critérios de correção (correspondência exata com uma solução de referência e corretude de acordo com o teste) e aplicando o framework a diferentes LLMs. Com base nesse framework, foram empregadas

técnicas bem estabelecidas para estimar como a confiança de um modelo se alinha com a correção real. Com isso, formular as seguintes perguntas de pesquisa:

1. **RQ1**: Quão confiantes os modelos de linguagem são em seus outputs em comparação à correção empírica deste output, especificamente para tarefas generativas comuns como síntese de funções, conclusão de código de nível de linha e reparo de programa?

2. **RQ2**: O alinhamento entre a confiança do LLM no código gerado, e sua correção, pode ser melhorado pelo redimensionamento da confiança?

3. **RQ3**: A confiança do LLM é melhor alinhada com a corretude do código gerado quando obtida via reflexão?

4. **RQ4**: Técnicas de aprendizado com pouca amostragem (few-shot), com tunagem a nível de instrução e aprendizado contextual, podem ser usadas para alcançar uma confiança melhor calibrada para conclusões de código?

Metodologia e Datasets

O estudo se concentra em três tarefas amplamente utilizadas em engenharia de software onde modelos de linguagem generativa são aplicáveis. Essas tarefas são: **Síntese de função**: Gerar funções completas com base em especificações fornecidas. **Conclusão de código em nível de linha**: Prever as próximas linhas de código com base no contexto anterior e **Reparo de programa**: Identificar e corrigir bugs em código existente. O artigo utiliza vários datasets para essas tarefas, incluindo os conjuntos de dados HUMANEVAL e MBPP, que consistem em pequenos programas Python. Embora esses conjuntos de dados sejam populares para síntese de funções, o artigo também incorpora conjuntos de dados como DYPYBENCH e SSTUBS, que são mais representativos de cenários de desenvolvimento de software do mundo real, particularmente de projetos GitHub de código aberto. Nas seções seguintes, o estudo detalha as tarefas, datasets, modelos usados e a abordagem quantitativa:

1. **Coretude de Código**: O artigo discute a noção de corretude no contexto do código gerado. Ele considera inicialmente correspondências exatas com soluções *ground-truth* como uma medida de corretude, particularmente relevante em cenários de reparo de defeitos onde uma versão conhecida com bugs é comparada a uma versão corrigida. No entanto, os autores reconhecem que essa definição estrita pode não capturar as nuances da corretude do código. Portanto, eles propõem usar casos de teste como uma indicação preferencial de corretude, pois os testes são amplamente usados na prática e podem ser automatizados. O artigo também reconhece as limitações dos casos de teste, como sua potencial instabilidade ("flakiness"), onde o mesmo teste pode produzir resultados diferentes no mesmo código.

2. **Metrificação de Confiança**: A metodologia envolve o cálculo de medidas de confiança associadas ao código gerado. Nesse sentido, Duas categorias de medidas são consideradas: **Probabilidade Intrínseca**: Probabilidade atribuída pelo modelo de linguagem generativa ao código de saída, refletindo a confiança interna do modelo em suas previsões. **Probabilidade Reflexiva**: Esta medida é obtida ao invocar novamente o modelo com instruções para estimar sua confiança na correção do código que ele acabou de gerar. Esta abordagem reflexiva visa fornecer uma avaliação mais compreensiva da confiança do modelo.

3. **Framework de Avaliação de Calibração**: O artigo estabelece uma estrutura de avaliação abrangente para avaliar o alinhamento das medidas de confiança com a correção real em diferentes tarefas e modelos. Esta estrutura inclui: Técnicas para estimar o quão bem as medidas de confiança se correlacionam com noções padrão de correção de código, bem como uma análise do Erro de Calibração Esperado (ECE) para quantificar a calibração da confiança do modelo.

4. **Overview das Tarefas e Datasets**:

Desse modo, o artigo conduz um estudo empírico para validar o framework e as metodologias propostas, analisando o desempenho de diferentes modelos e sua calibração nas tarefas definidas. Os modelos alvos foram **GPT-3.5**, **CODEX** da OpenAI e o **CodeGen2-16B**, orientado a código.

Task	Dataset	Dataset Size	Correctness Measure	Confidence Measure	Calibration Metric
Function synthesis	HumanEval	164	Test-passing Correctness	Average Token Probability, Generated Sequence Probability, Verbalized Self-Evaluation, Question Answering Logit	Brier Score, *ECE*
	MBBP Func	880			
Line-level Completion	DyPyBench	1,988	Test-passing Correctness, EM		
Program Repair	Defects4J 1-line	120			
	ManySStubs4j	3,000	Exact-Match (EM)		

Figura 3 – Lista de Tarefas de Avaliação dos Modelos com as Métricas e Datasets Associados

Resultados

1. **RQ1**: Os modelos demonstraram níveis variados de correção em diferentes tarefas e conjuntos de dados. Por exemplo, o GPT-3.5 mostrou forte desempenho em tarefas de síntese de funções, alcançando uma alta porcentagem de casos de teste aprovados no conjunto de dados HumanEval, com uma taxa de correção de aproximadamente 64,60%. Isso indica que o modelo foi capaz de gerar funções que atenderam aos requisitos especificados de forma eficaz.

   a) **Conclusão de código em nível de linha**: No conjunto de dados DyPyBench, que se concentra na conclusão de código em nível de linha, o GPT-3.5 atingiu uma taxa de correção de cerca de 33,22% para aprovação em todos os casos de teste. Isso sugere que, embora o modelo seja capaz de gerar conclusões de código relevantes, ainda há um espaço significativo para melhorias para garantir que essas conclusões estejam corretas e passem em todos os testes associados

   b) **Reparo de programa**: O desempenho em tarefas de reparo de programa, avaliadas usando o conjunto de dados Defects4J, revelou que modelos como Codex e GPT-3.5 tiveram taxas de correção mais baixas, com o Codex alcançando cerca de 23,33% para aprovação em todos os casos de teste. Isso indica que reparar código com bugs é uma tarefa mais desafiadora para esses modelos, destacando as complexidades envolvidas na compreensão e correção do código existente

   c) **Calibração de Confiança**: O estudo também avaliou a calibração das medidas de confiança dos modelos, que são cruciais para entender o quão bem as probabilidades previstas refletem a correção real. As descobertas indicaram que, embora alguns modelos tenham exibido calibração aprimorada, ainda havia discrepâncias entre os níveis de confiança dos modelos e a correção real de suas saídas. Por exemplo, a pontuação Brier e as métricas de Erro de Calibração Esperado (ECE) (ambas metrificam a distância de uma calibração perfeita), sugeriram que a confiança dos modelos nem sempre se alinhava bem com seu desempenho, particularmente no contexto de reparo de programa.

   Ademais, os resultados enfatizaram a importância de usar modelos bem calibrados para a tomada de decisões em tarefas de engenharia de software. Uma medida de confiança bem calibrada permite que os usuários tomem decisões mais informadas sobre quais conclusões ou reparos de código aceitar, gerenciando assim os riscos associados ao uso de código gerado pelo LLM. Assim, embora modelos de linguagem como GPT-3.5 e Codex sejam promissores na geração de código correto, há desafios significativos em garantir que seus níveis de confiança reflitam com precisão a correção de suas saídas, particularmente em tarefas mais complexas como reparo de programa. Isso ressalta a necessidade de pesquisa contínua para melhorar tanto a correção quanto a calibração desses modelos.

2. **RQ2**: **Eficácia do redimensionamento de confiança**: O estudo descobriu que técnicas de redimensionamento de confiança, como o escalonamento de Platt, podem melhorar o alinhamento entre a confiança intrínseca fornecida pelo LLM e a correção real do código gerado. Este método envolve o ajuste das probabilidades previstas do modelo para refletir melhor a correção empírica observada nas saídas. Uma observação importante foi que o redimensionamento de confiança requer uma quantidade suficiente de dados para determinar com precisão os parâmetros da função de dimensionamento. Isso é particularmente importante para tarefas de geração de código, onde a disponibilidade de dados rotulados (ou seja, saídas corretas conhecidas) é crucial para o redimensionamento eficaz.

	All Pass@1			Exact-Match		
	CodeGen2	Codex	GPT-3.5	CodeGen2	Codex	GPT-3.5
SStubs	-	-	-	0.73%	**27.77%**	20.27%
DyPyBench	28.84%	32.96%	**33.22%**	19.68%	23.60%	**23.96%**
Defects4J	0.00%	**23.33%**	19.17%	0.00%	**19.17%**	15.00%
HumanEval	23.17%	47.24%	**64.60%**	-	-	-
MBPP	29.08%	61.79%	**72.04%**	-	-	-

Figura 4 – Comparação de Performance dos Modelos nas Tarefas em Pass@1

Os resultados indicaram também que, após a aplicação do redimensionamento de confiança, houve uma melhoria notável na calibração dos modelos. Isso se refletiu em valores mais baixos de Erro de Calibração Esperado (ECE), sugerindo que as medidas de confiança redimensionadas estavam mais alinhadas com a correção real do código gerado. Embora o redimensionamento de confiança geralmente melhorasse a calibração, a eficácia variou entre diferentes tarefas e conjuntos de dados. Algumas tarefas mostraram melhorias mais significativas do que outras, indicando que o contexto e a natureza da tarefa podem influenciar o sucesso das técnicas de redimensionamento.

As descobertas também sugeriram que o redimensionamento de confiança pode ser mais eficaz do que alguns métodos reflexivos, que foram considerados inconsistentes na melhoria do alinhamento. Isso destaca o potencial do redimensionamento como uma abordagem mais confiável para aprimorar a calibração de LLMs em tarefas de geração de código. Portanto, ao melhorar o alinhamento entre confiança e correção, os desenvolvedores podem tomar decisões mais bem informadas ao usar LLMs para geração de código, levando, em última análise, a saídas de código de maior qualidade e mais confiáveis.

3. **RQ3**: **Aplicação de Uma Estratégia Reflexiva**: O estudo descobriu que métodos reflexivos, onde o modelo é solicitado a avaliar seu próprio código gerado e fornecer uma medida de confiança, produziram resultados inconsistentes. Em alguns casos, essa abordagem melhorou o alinhamento entre a confiança do modelo e a correção real do código, enquanto em outros casos, não levou a melhorias significativas. A eficácia das medidas de confiança reflexivas pareceu ser altamente dependente da tarefa específica que está sendo executada. Certas tarefas podem se beneficiar mais da reflexão do que outras, indicando que o contexto em que o modelo opera desempenha um papel crucial na determinação do sucesso dessa abordagem.

   No geral, as medidas de confiança intrínseca fornecidas pelos LLMs foram geralmente consideradas mal alinhadas com a correção real. As medidas reflexivas, embora às vezes melhores, não superaram consistentemente as medidas intrínsecas em todas as tarefas. Isso sugere que simplesmente solicitar que o modelo reflita sobre suas saídas pode não ser suficiente para garantir avaliações de confiança confiáveis. Apesar das inconsistências, o estudo destacou o potencial das abordagens reflexivas para aprimorar a calibração em certos cenários. Isso indica que uma exploração e refinamento adicionais das técnicas reflexivas podem levar a um melhor alinhamento entre confiança e correção no futuro. As descobertas sugerem que mais pesquisas são necessárias para entender as condições sob as quais as medidas de confiança reflexiva podem ser efetivamente utilizadas, como investigar como diferentes estratégias de solicitação ou contexto adicional podem influenciar a capacidade do modelo de avaliar com precisão seus próprios resultados.

4. **RQ4**: **Técnicas Few-Shot com Modelo Ajustado para Instruções e Contexto**: O estudo descobriu que empregar técnicas de few-shot, particularmente em um ambiente reflexivo com o modelo ajustado por instrução GPT-3.5, levou a melhorias substanciais na calibração de medidas de confiança. A pontuação de habilidade melhorou significativamente de 0 para um nível muito mais alto de 0,15, indicando que as pontuações de confiança se tornaram mais alinhadas com a correção real do código gerado. Os resultados destacaram que o few-shotting aumentado pela recuperação, que envolve o uso de exemplos relevantes de um sistema de recuperação para informar as previsões do modelo, foi particularmente eficaz em melhorar

a calibração. Essa abordagem permitiu que o modelo alavancasse informações contextuais de exemplos anteriores semelhantes, levando a avaliações de confiança mais precisas.

As melhorias na calibração foram especialmente notáveis na tarefa amplamente usada de conclusão de código. Isso sugere que as técnicas de few-shot são particularmente benéficas para tarefas em que o contexto e os exemplos podem orientar a compreensão e a confiança do modelo em suas saídas. As descobertas também indicaram que as técnicas de few-shot podem fornecer um meio mais confiável de melhorar a calibração de confiança em comparação com outros métodos explorados no estudo, como somente abordagens reflexivas. Isso posiciona o aprendizado few-shot como uma estratégia promissora para aumentar a confiabilidade de LLMs em tarefas de geração de código. O estudo enfatiza as implicações práticas do uso de técnicas de poucos tiros para desenvolvedores que trabalham com LLMs em engenharia de software, ao melhorar a calibração de medidas de confiança, os desenvolvedores podem tomar decisões mais informadas sobre a aceitação e revisão do código gerado, levando, em última análise, a um melhor controle de qualidade no desenvolvimento de software.

Discussões

Em conclusão, o estudo revela que os LLMs, particularmente quando usados "out of the box"(sem ajustes), geralmente fornecem medidas de confiança mal calibradas em relação à correção do código gerado. Esse desalinhamento pode levar a avaliações não confiáveis da qualidade do código. Várias técnicas foram exploradas para melhorar a calibração, incluindo redimensionamento de confiança (por exemplo, dimensionamento de Platt), abordagens reflexivas e aprendizado de few-shot. Enquanto o redimensionamento geralmente melhorou a calibração, os métodos reflexivos mostraram resultados inconsistentes. As técnicas de few-shots, especialmente quando aumentadas com métodos de recuperação, demonstraram melhorias substanciais na calibração.

As descobertas ressaltam o potencial de técnicas de calibração aprimoradas para aumentar a confiabilidade dos LLMs no desenvolvimento de software do mundo real. Ao fornecer aos desenvolvedores melhores medidas de confiança, essas técnicas podem facilitar um controle de qualidade mais eficaz do código gerado. Os autores sugerem que mais pesquisas são necessárias para explorar métodos de calibração adicionais e entender melhor as condições sob as quais diferentes técnicas são mais eficazes. Por fim, reconheceram aspectos de ameaça à validade do trabalho: **1)**: Tamanho das Amostras Variadas e Generalizabilidade dos Datasets Usados, **2)**: Conflito entre Dados Reais e Artificiais, **3)** Seleção Limitada de LLMs e **4)**: Design Experimental Sujeito a Falhas e Vieses de Experimentação.

4.2.3 Can LLMs Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation

Introdução

Nesse estudo[16], ressalta-se, primeiro, o fato de que os estudos precedentes de avaliação de performance de LLMs para código levaram em consideração apenas corretude funcional e semântica, que, apesar de ser uma abordagem relevante, desconsidera aspectos como confiabilidade e robustez do output, especialmente no contexto de software real. Nesse sentido, o norte de avaliação dos LLMs por este estudo consiste na verificação do uso adequado/indevido de APIs em **Java**, um ponto comum de falha humana que pode gerar uma gama de erros como vazamento de recursos, comportamento inesperado, entre outros erros, o que constitui um cenário realista perante o desenvolvedor de software médio quando comparado a problemas de programação mais genéricos/amplos.

Nesse âmbito de engenharia de software, LLMs são capazes de produzir respostas mais bem ajustadas e customizadas para as dúvidas dos programadores comparado à experiência de pesquisa em fóruns de programação online, como o famoso StackOverflow. Nesse contexto, em contraste com os fóruns online de programação, os trechos de código gerados pela IA não são revisados por pares da comunidade, de modo que erros de uso de APIs (alguma condição não tratada, exceção não recuperada, por exemplo) podem escapar despercebidos de um programador desatento.

Da perspectiva da engenharia de software, a robustez e confiabilidade dos outputs de código dos LLMs ainda não foram exploradas de forma extensiva como a corretude sintática, semântica e funcional, atributos necessários, mas não suficientes para uma análise rica da qualidade do código gerado. A maioria dos benchmarks existentes de avaliação do código produzido por LLMs, usualmente baseados em datasets de plataformas de problemas de programação, são centrados no resultado da sua execução direta, de modo que o código é aceito desde que é funcional para o propósito de escopo fechado. Contudo, esse cenário não atende à realidade de muitos desenvolvedores de software: código confiável sem riscos a longo prazo.

Portanto, a proposta principal do estudo é o **RobustAPI**, um benchmark compreensivo que endereça a confiabilidade e robustez do código gerado por GenAI, o que inclui um dataset de problemas de programação e uma ferramenta de análise estática de programas (AST: árvore de sintaxe abstrata) para avaliar comparativamente se os programas gerados estão de acordo com regras de uso de API estabelecidas. Com o objetivo de representatividade do cenário real de desenvolvimento de software, 1208 questões relevantes foram coletadas no StackOverflow sobre **Java** acerca de 18 APIs, uma das linguagens mais usadas nesse contexto devido à sua alta compatibilidade e disponibilidade para aplicativos web e mobile. Os modelos analisados foram: GPT-3.5, GPT-4 (Closed-Source), Llama-2, e Vicuna-1.5. Os trabalhos relacionados mencionados reforçam a perspectiva do foco em aspectos técnicos de nível baixo/intermediário: sucesso em compilação, corretude funcional, eficiência, introdução de bugs mais sutis, pouco ou nenhum ganho no quesito segurança, entre outros.

Metodologia

1. **Coleta de Dados**: Inicialmente, o modelo proposto baseou-se no dataset ExampleCheck, que estudou o uso indevido de APIs Java em fóruns online. Na figura 5 são mostradas as 18 APIs cobertas para o estudo em questão, cobrindo 6 domínios: processamento de strings, estruturas de dados, desenvolvimento mobile, criptografia, I/O e operacionalização de bancos de dados. Em seguida, os pesquisadores procuraram por questões relevantes às essas APIs no StackOverflow, somente aquelas com respostas com ou sem uso indevido dessas APIs. Desse modo, o modelo será avaliado em cenários onde humanos podem errar com uma certa frequência.

2. **Geração de Prompts**: De início, instruem o modelo com uma introdução à tarefa e o formato de resposta requerido. Em seguida, pequenas demonstrações de código são fornecidas como forma de construção de contexto para o LLM, de modo que simulam um usuário fazendo perguntas de programação sem prover nenhuma dica adicional da documentação da API: um cenário típico na rotina SWE (Software Engineering) de desenvolvedores mais novatos junto a LLMs.

3. **Amostragem**: Dois cenários de amostragem, uma estratégia comprovadamente benéfica para a precisão dos LLMs, foram aplicados neste estudo:

   a) **One-Shot Irrelevant**: Um exemplo de uma API irrelevante para a tarefa é fornecido para o LLM em questão;

   b) **One-Shot Rrelevant**: Um exemplo da mesma API (fora do dataset principal) da questão dada é fornecido ao LLM, que contém um par de uma pergunta e reposta sem uso indevido de API;

   Nota-se que, em ambos os cenários, os exemplos foram construídos a mão e validados acerca da regra de uso da API em questão.

4. **Detecção de Uso Indevido de APIs**: Aqui está um dos elementos principais e diferenciais do paper: o RobustAPI acompanha uma etapa de análise estática dos programas sintetizados pelos LLMs. Nesse esquema, a análise extrai estruturas de controle e chamada de função do código produzido e as compara com as regras de uso das APIs em questão, de modo que, com uma AST (Árvore de Sintaxe Abstrata) do trecho de código do LLM, a análise caminha nesta árvore para determinar a sequência ordenada de chamadas de métodos e a compara com a sequência determinada pela regra de uso da API. Por fim, a análise computa a sequência comum mais longa entre a sequeência de calls do trecho de código e as regras de uso da API: se

API	Domain	Conseq*	Github*
StringTokenizer.nextToken	String	(iii)	13.3K
String.getBytes	Process	(iii)	88.1K
JsonElement.getAsString	(307)	(iii)	4.4K
List.get	Data	(iii)	2.7M
Map.get	Structure	(iii)	2.4M
Iterator.next	(404)	(iii)	918K
ProgressDialog.dismiss	Mobile	(iii)	54K
TypedArray.getString	Develop	(iv)	6.8K
ApplicationInfo.loadIcon	(75)	(v)	3.6K
Activity.setContentView		(v)	4.6K
Cipher.init	Crypto (10)	(iii)	66.3K
RandomAccessFile.write	I/O (390)	(i)	129K
BufferedReader.readLine		(iii)	74.8K
PrintWriter.write		(i)	1.1M
File.mkdirs		(ii)	73.2K
File.createNewFile		(i)	176K
FileChannel.write		(i)	5.2K
SQLiteDatabase.query	Database (22)	(iv)	4K
Total	1208		7.8M

Table 1: 18 popular Java APIs in RobustAPI. They are easily misused by developers according to the existing literature of software engineering (Zhang et al. 2018). *Consequences: (i) data loss; (ii) file system corruption; (iii) program crash; (iv) resource leak; (v) user interface bug. *Github: occurrences of this API on Github.

Figura 5 – Esquema de Dados para Teste

esta sequência não casa com as regras esperadas de uso da API, o programa em questão atesta uso indevido da API.

5. **Setup do Experimento**: Apenas 4 modelos de linguagem foram postos para teste: GPT-3.5 (OpenAI 2023), GPT-4 (OpenAI 2023), Llama2, Vicuna-1.5, sem tunagem extensiva de hiperparâmetros. Os resultados principais são em Pass@k com k = 1 *sample* nos três cenários: **One-shot relevant** e **One-shot irrelevant** como supracitado e **Zero-shot**: onde nenhum exemplo é fornecido

6. **Métricas**: Três principais métricas personalizadas foram aplicadas:

   a) **API Misuse Rate**: Taxa de uso indevido de APIs definida por $N_{misuse}/(N_{misuse} + N_{pass})$, onde N_{misuse} e N_{pass} são, respectivamente, o número de casos de uso indevido detectado pelo API Checker (análise estática) e o número de casos sem detecção de uso indevido de APIs.

   b) **Compilation Rate**: Taxa de código compilado com sucesso: $(N_{misuse} + N_{pass})/N$, onde N é o número total de questões.

   c) **Overall API Misuse Percentage**: Proporção de uso indevido de API dentre todas as questões: N_{misuse}/N

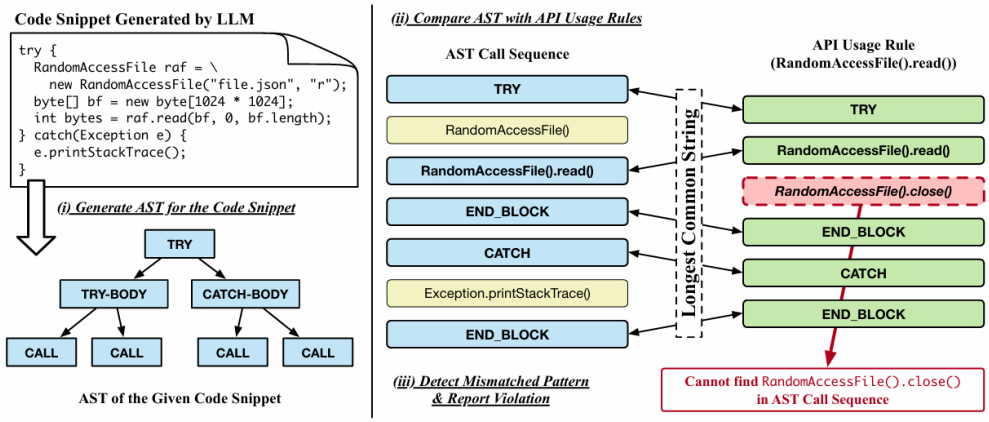


Figura 6 – Overview do API Checker

7. **Perguntas de Pesquisa**:

   a) **RQ1**: Quais são as taxas de uso indevido da API ao responder a perguntas de codificação do mundo real por esses LLMs?

   b) **RQ2**: Como as amostragens irrelevantes afetam os resultados?

   c) **RQ3**: Exemplos corretos de uso da API podem reduzir o uso indevido?

   d) **RQ4**: Por que o código gerado pelo LLM falha na verificação de uso da API?

Resultados

Model	Zero-shot			One-shot-irrelevant			One-shot-relevant		
	Misuse Rate ↓	Compilable Rate ↑	Overall Misuse ↓	Misuse Rate ↓	Compilable Rate ↑	Overall Misuse ↓	Misuse Rate ↓	Compilable Rate ↑	Overall Misuse ↓
GPT 3.5	62.97%	79.14%	49.83%	68.09%	91.06%	62.00%	38.56%	80.71%	31.13%
GPT 4	68.81%	90.23%	62.09%	70.38%	91.39%	64.32%	54.40%	90.40%	49.17%
Llama 2*	7.34%*	9.02%*	0.66%*	61.36%	80.13%	49.17%	64.47%	72.93%	47.02%
Vicuna 1.5	45.66%	37.17%	16.97%	57.85%	83.86%	48.51%	42.53%	64.24%	27.32%
ds-coder-6.7b-base	41.55%	40.65%	16.89%	75.60%	95.90%	72.43%	64.12%	67.14%	43.05%
ds-coder-6.7b-instruct	47.52%	50.00%	23.76%	59.04%	96.61%	57.04%	38.40%	86.01%	33.03%

Figura 7 – Performance por LLM no RobustAPI: ↑: quanto maior melhor, ↓: quanto menor melhor.

A partir da figura 7, podemos construir diversos raciocínios e insights. De início, todos os modelos avaliados sofreram com problemas de uso indevido de API, até mesmo os modelos comerciais de última geração como GPT-3.5 e GPT 4. Em configurações de **zero-shot**, Llama tem a menor taxa de uso indevido de API. No entanto, isso se deve em parte ao fato de que a maioria das respostas do Llama não inclui nenhum código. Uma descoberta contraintuitiva é que GPT-4 na verdade tem uma taxa de uso indevido de API maior do que GPT-3.5, embora a capacidade de codificação de GPT-4 seja comprovada 40% mais avançada do que seu antecessor, GPT-3.5 (OpenAI 2023). Também avaliou-se um LLM especializado em código, **DeekSeekCoder**, que é treinado em uma variedade de linguagens de programação incluindo Java, e supera muitos LLMs de código existentes. Observou-se que o LLM especializado em código pode gerar mais amostras compiláveis. No entanto, a taxa de uso indevido de API não é significativamente melhor do que outros modelos. Isso indica que com a capacidade de geração de código de grandes modelos de linguagem sendo amplamente melhorada hoje em dia, a confiabilidade e a robustez do código de produção do mundo real aumentam como um problema despercebido.

Acerca do cenário **One-shot Irrelevant**, para a maioria dos modelos, os exemplos irrelevantes não reduziram significativamente as taxas de uso indevido de APIs, de modo que a aumentou levemente em alguns casos. O estudo aponta como uma plausível razão para isso o fato de que esse cenário ”encoraja” o LLM a fornecer um trecho de código mais comprido, o que aumenta as chances de gerar um mal uso da API. Nesse contexto, oe

estudo evidencia os achados **1)**: Dentre todsas as respostas contendo código compilável, 57-70% das respostas dos LLMs contêm uso indevido de APIs, **2**: Amostras irrelevantes não ajudam a diminuir as taxas de uso indevido de APIs, mas engatilha mais respostas de código válidas, tornando mais efetivo o processo de *benchmarking*.

Já sobre o cenário de **One-shot Relevant**, após adicionar um uso correto da API como contexto, as taxas de uso incorreto da API dos LLMs GPT-3.5, GPT-4 e Vicuna caem significativamente. Isso indica uma melhoria efetiva sob essa configuração de experimento. Quanto ao Llama, o exemplo relevante não melhora o desempenho. Este experimento mostra que alguns LLMs podem efetivamente "aprender" o uso correto da API e seguir o uso. No entanto, como os modelos de linguagem existentes são treinados com dados de repositórios de código, se os conjuntos de dados de treinamento contiverem um grande número de violações de API, os modelos de linguagem são propensos a gerar código com uso incorreto da API, o que explica a alta taxa de uso incorreto da API em avaliações **zero-shot irrelevant** e **one-shot irrelevant**.

Pass@k	Misuse Rate	Compilation Rate	Overall Misuse
Pass@1	39.06%	76.08%	29.72%
Pass@5	21.98%	93.79%	20.61%
Pass@10	16.51%	96.27%	15.89%

Figura 8 – Taxa de uso indevido cai com o número sucessivo de tentativas de prompt para o GPT-3.5

Análise de Robustez: O benchmark foi aplicado em diferentes parametrizações de temperatura do modelo (quanto mais "quente", mais aberto à criatividade), mas essa abordagem não surtiu efeito significativo nas taxas de uso indevido e/ou compilação. Para estudar o efeito de diferentes *prompts*, o estudo avaliou como a taxa de uso indevido da API muda quando substitui-se os exemplos one-shot pelas regras de uso da API. As regras simbolizadas foram fornecidas para o ChatGPT e as regras em linguagem natural foram geradas como output. Portanto, adicionaram as regras de uso como parte dos prompts e avaliaram o GPT-3.5 com RobustAPI. Os resultados indicam que as regras de uso da API podem não ajudar a reduzir a taxa de uso indevido de APIs em comparação com exemplos one-shot relevantes. Sobre as respostas dos LLMs que não passaram na avaliação de uso de APIs, descobriram duas categorias de falha: código não compilável e código compilável mas que contém uso indevido de APIs. O LLM GPT-4 performa melhor dentre todos os outros no que tange à compilabilidade dos códigos, mas experimentos com mais tentativas e amostragens ajudam a reduzir a taxa de falha de compilação. Dentre todas as APIs utilizadas, a API de desenvolvimento Android *Activity.setContentView* teve a menor taxa de uso indevido para todos os LLMs avaliados.

4.2.4 Towards more realistic evaluation of LLM-based code generation: an experimental study and beyond

Introdução

Publicado em junho de 2024, este artigo[17], inicialmente, aponta para o fato de que os esforços de avaliação do código produzido por LLMs, ao normalmente utilizarem códigos nas versões mais recentes de projetos de software, falham ao considerar aspectos dinâmico-evolutivos que os softwares possuem com o tempo, aspecto o qual o estudo define como situação "evolving-ignored". Nesse sentido, essas avaliações podem incorrer em vazamento de contexto futuro de código e contexto útil faltante, o que, em consequeência, contribui para que essas avaliações de performance de LLMs tenham precisão comprometida. Com isso, um estudo empírico é conduzido para entender profundamente a performance de geração de código de LLMs com configurações que refletem a natureza evolutiva do desenvolvimento de software. Primeiro, sintetizaram um dataset de códigos de teste a nível de repositório e orientado a evolução: **HumanEvo**, nas linguagens Python e Java, equipado com uma ferramenta de avaliação automatizada baseada em execução. Ademais, os pesquisadores categorizaram o dataset **HumanEvo** de acordo com níveis de dependência de software para analizar de modo mais compreensivo a performance dos LLMs

para geração de funções com diferentes níveis de dependência em código. Por último, conduziram experimentos extensivos com o **HumanEvo** com sete LLMs para verficação da efetividade do *benchmark* proposto.

Mais detalhadamente, muitos dos benchmarks de avaliação a nível de repositório apontados na literatura atual (em relação ao tempo do artigo) amostram funções de projetos reais de software como formas de perguntas para os LLMs gerarem essas funções como resposta e ferramenta de avaliação. Embora seja uma estratégia válida, esse tipo de ferramental de avaliação desconsidera a natureza evolutiva destes projetos de software ao passo que consideram somente a versão mais atual para geração de perguntas, o que incorre em uma análise imprecisa do cenário real de desenvolvimento de software, onde múltiplas funcionalidades passam por diversas etapas evolutivas e construcionais. Nesse sentido, os pesquisadores identificaram dois problemas principais: **1) future context leakage issue**: o contexto fornecido ao LLM durante a avaliação inclui código ou informações que não estariam disponíveis para um desenvolvedor no momento em que ele estava escrevendo a função alvo. **2) useful context missing issue**: falha em fornecer ao LLM código ou informações relevantes que estariam disponíveis para um desenvolvedor no momento em que ele estava escrevendo uma função alvo (poderia ter sido removida antes da construção do dataset). Como resultado, o modelo pode não ter acesso a todo o contexto necessário para gerar código preciso e eficaz.

Construção do Benchmark HumanEvo

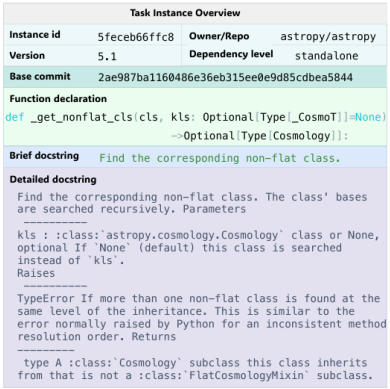


Figura 9 – Overview do Formato de Tarefa de Avaliação do Benchmark

HumanEvo consiste em um total de 400 instâncias de tarefa, compreendendo 200 perguntas de programação em Python e 200 em Java. Cada instância de tarefa é associada a um número de identificação exclusivo. Um campo Proprietário/Repositório ajuda a identificar o projeto GitHub do qual a função selecionada se origina, enquanto um campo Versão indica a versão do projeto durante a implementação desta função. O commit base da tarefa se refere ao último commit no branch antes da função alvo ser confirmada, usado para restaurar o repositório do projeto para o estado antes da função alvo ser implementada. A declaração Function da tarefa da função alvo será fornecida ao LLM como parte do prompt, junto com sua docstring correspondente. Como os estilos de docstring de funções em projetos pragmáticos variam, com alguns adotando comentários linha por linha e outros até mesmo sem comentários correspondentes, foi necessário reescrever a docstring para cada função selecionada: O campo docstring Brief da tarefa fornece um resumo conciso da funcionalidade da função alvo, enquanto o campo docstring Detailed fornece uma descrição completa da funcionalidade, entradas e saídas da função.

1. **Seleção de Projetos de Alta Qualidade**

   - **Critérios do Projeto:** Os pesquisadores buscaram selecionar projetos reais de alta qualidade para garantir a relevância do benchmark. Para Python, eles se concentraram nas 5.000 bibliotecas mais baixadas do Python Package Index (PyPI) em 2023. Para Java, eles miraram nos 200 principais projetos estrelados no GitHub, pois esses projetos são normalmente bem documentados e seguem práticas de desenvolvimento estruturadas.

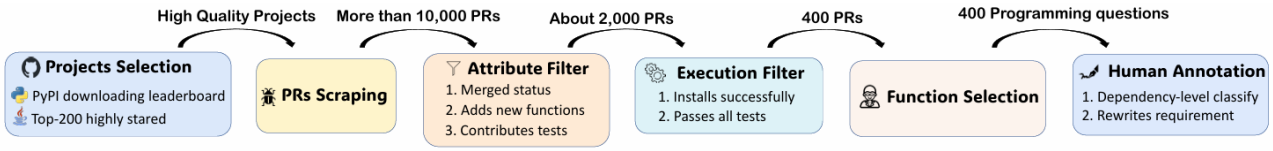


Figura 10 – Pipeline de Construção do Benchmark

- **Implantação e Teste:** Cada projeto selecionado foi implantado, e sua estrutura de teste foi executada para confirmar que o projeto poderia ser executado corretamente e de forma estável. Esta etapa garantiu que apenas projetos funcionais fossem incluídos no benchmark.

2. **Coleta de Pull Requests (PRs)**

- **Rastreamento de PR:** Os pesquisadores utilizaram a API do desenvolvedor do GitHub para rastrear um grande número de PRs iniciais dos repositórios selecionados. Eles coletaram um total de 12.039 PRs, registrando meticulosamente metadados relacionados a esses PRs para filtragem subsequente.
- **Foco em PRs mesclados:** A ênfase foi colocada em PRs que tinham um status "mesclado", indicando que as mudanças foram aceitas e incorporadas à base de código principal. Esse foco garantiu que as funções selecionadas para avaliação tivessem passado por um processo de revisão, melhorando sua qualidade.

3. **Filtragem Inicial de PRs**

- **Filtragem baseada em atributos:** Os PRs rastreados foram filtrados com base em três atributos principais:
  - **Status mesclado:** Apenas PRs que foram mesclados foram considerados, garantindo que o código tenha sido revisado e aceito.
  - **Introdução de novas funções:** Os PRs tiveram que introduzir novas funções, que são essenciais para o desenvolvimento do projeto de software e podem ser enquadradas como questões de programação.
  - **Cobertura de teste:** Cada PR teve que introduzir pelo menos um novo teste que cobrisse a função recém-adicionada, garantindo que as funções fossem testadas adequadamente.

4. **Verificação da Cobertura de Teste**

- **Configuração do ambiente de execução:** Para cada PR, um ambiente de execução foi estabelecido para executar a estrutura de teste do projeto correspondente. Esta etapa foi crucial para verificar se as funções recém-adicionadas foram cobertas por testes, o que é vital para avaliar sua correção e confiabilidade.

5. **Criação de Perguntas de Programação**

- **Seleção de funções:** Após filtrar e verificar a qualidade das funções, os pesquisadores selecionaram um total de 400 perguntas de programação — 200 para Python e 200 para Java. Essas perguntas foram projetadas para refletir tarefas de codificação realistas que os desenvolvedores podem encontrar em projetos reais.
- **Diversidade de funções:** As funções selecionadas abrangeram uma gama diversificada de cenários de programação, incluindo vários níveis de complexidade e tipos de dependência (funções autônomas, intraclasse e interclasse).

6. **Anotação Humana**

- **Criação de docstring:** Para aprimorar a diversidade e o realismo do benchmark, os pesquisadores criaram manualmente dois tipos de docstrings para cada função: detalhado e breve. Essa abordagem teve como objetivo refletir os diferentes estilos de documentação encontrados em projetos do mundo real e avaliar como os vários níveis de detalhes em docstrings afetam o desempenho do LLM.
- **Classificação de Nível de Dependência:** As funções foram categorizadas com base em seus níveis de dependência, que incluíam funções autônomas (independentes de outras funções), funções intraclasse (dependentes de outras funções dentro da mesma classe) e funções interclasse (dependentes de funções em outros arquivos). Essa classificação auxilia na análise do desempenho do LLM em diferentes contextos.

7. **Avaliação de LLMs**

- **Configuração Experimental:** O benchmark foi usado para avaliar vários LLMs tradicionais, incluindo a série CodeLlama, série DeepSeekCoder, GPT-4 e GPT-3.5-Turbo. A avaliação teve como objetivo revelar o desempenho desses modelos em cenários de desenvolvimento prático e destacar a importância da conscientização evolutiva em tarefas de geração de código.
- **Comparação de Desempenho:** Os pesquisadores compararam o desempenho de LLMs em configurações de conscientização evolutiva ignoradas e de conscientização evolutiva, demonstrando como os LLMs podem exibir desempenho inflado quando o contexto futuro é incluído incorretamente.
- **Métrica de Avaliação:** A métrica usada foi a **Pass@1**, calculada em uma média simples de três experimentos seguidos para suavizar efeitos de aleatoriedade da geração dos modelos.
- **Perguntas de Pesquisa**
  - **RQ1:** O quão efetivo é **HumanEvo** para avaliação sistemática da performance de LLMs na geração de código a nível de repositório? Para isso, conduziram experimentos em dois cenários: **evolving-aware** (ciente do desenvolvimento evolutivo do software) e **evolving-ignored**, de modo a explorar as diferenças em cenários mais realistas
  - **RQ2:** Como a performance de geração de código muda com a evolução do código dos repositórios?
  - **RQ3:** Como diferentes descrições de código (docstrings) no dataset afetam a performance de geração de código?

Resultados - RQ1

Uma análise inicial dos resultados no que tange ao vazamento de contexto futuro de código mostra uma significativa redução de performance sob a condição **evolving-aware** quando comparado ao cenário **evolving-ignored**. De modo que sugere que, no cenário em que a evolução do software é desconsiderada, a fonte de contexto provida para os LLMs continha muitas informações privilegiadas, apontando para uma performance não realista e inflada dos modelos de linguagem. Ademais, sob a configuração **evolving-aware**, a precisão dos LLMs na geração de funções intraclasse e interclasse diminuiu visivelmente. O declínio é mais proeminente para funções interclasse, pois essas funções normalmente exigem contexto entre arquivos, que geralmente é mais disperso. À medida que o projeto evolui, suas várias partes tendem a se tornar mais abrangentes. Portanto, sob a configuração de **evolving-aware**, os modelos podem fazer referência a uma quantidade crescente de contexto. Consequentemente, quando a configuração de **evolving-aware** perde o acesso a esses contextos, o desempenho do modelo se deteriora significativamente. Portanto, os experimentos mostram uma tendência inflada no desempenho do LLM sob a configuração **evolving-ignored**. Após mais investigação, descobriu-se que tanto a presença de muito contexto estranho quanto o desaparecimento do contexto que deveria existir afetaram significativamente o desempenho do modelo, resultando no desempenho do modelo se desviando da realidade.

Resultados - RQ2

Acerca do impacto da configuração **evolving-ignored** na performance dos LLMs com a evolução dos projetos, o desempenho dos LLMs desvia da realidade conforme o projeto evolui. No entanto, a melhoria artificial no

desempenho do modelo induzida pela configuração **evolving-ignored** não exibe um aumento linear, mas sim uma curva indicando uma tendência ascendente. Atualizações no projeto às vezes introduzem contextos de projeto irrelevantes, que podem atuar como ruído e impactar o desempenho dos LLMs. Além disso, atualizações de projeto podem resultar na remoção ou modificação de contextos originalmente úteis, levando à perda direta de contexto que auxilia na geração objetiva do modelo, causando, assim, um declínio no desempenho do modelo. Esses fatores podem ser os principais motivos para a ligeira diminuição ocasional no desempenho do modelo ao usar diferentes versões do projeto como fontes de contexto. Portanto, na configuração **evolving-ignored**, a medida que o projeto evolui, a performance do modelo desvia gradualmente da realidade, e a superestimação não aumenta de forma linear, mas segue uma curva flutuante.

Resultados - RQ3

Durante o processo de coleta de dados para a construção do benchmark, os pesquisadores perceberam que ora os projetos preferiram escrever docstrings detalhadas, ora preferiram descrever a funcionalidade em afirmações breves. Nesse sentido, escreveram manualmente descrições tanto breves quanto detalhadas para cada função no **HumanEvo**. Com isso, observaram que, comparado a docstrings mais reduzidas, docstrings mais detalhadas forneceram mais informações benéficas para os LLMs, resultando em uma melhor performance. No entanto, devido ao extenso contexto do projeto necessário para concluir a tarefa de geração de código a nível de repositório, a atenção do modelo é dispersa. Consequentemente, a melhoria de desempenho em docstrings detalhadas não é substancial.

4.3 Segurança do Uso de IA Generativa Para Código

4.3.1 Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers

Como o título sugere, o estudo em questão [18] adota uma abordagem empírica para avaliar o desempenho do amplo modelo de linguagem GPT. Nesse sentido, os pesquisadores compilaram trechos de códigos em Java provenientes de perguntas relacionadas à segurança no fórum conhecido StackOverflow (**SO**) e compararam as respostas dos fóruns com as respostas geradas pela *GenAI* com as correspondentes perguntas do **SO** enviadas nos prompts. Um dos principais fatores motivadores para a pesquisa em segurança no uso de *GenAI* para codificação em segurança foi o fato apontado pelo paper de que, em um report, 97% dos desenvolvedores e líderes de segurança integram Inteligência Artificial generativa (IA), particularmente LLMs, em seu processo de desenvolvimento. Nesse sentido, os códigos foram comparados e analisados conforme as CWEs (Common Weakness Enumeration)[14].

Em contexto introdutório, o estudo também aponta referências que indicaram que quase um terço dos projetos de código aberto com estrelas no GitHub tem um mantenedor usando o GitHub Copilot, bem como o fato de que, apesar da comprovada preocupação dos desenvolvedores consultados acerca dos riscos de segurança do uso de *GenAI* no desenvolvimento de software, 74% deles se sentem pressionados a usar essas ferramentas no ambiente de trabalho. Este cenário levanta questionamentos e preocupações acerca das potenciais vulnerabilidades que estão vazando no código gerado via *GenAI* para ambientes de produção críticos.

Neste cenário, o estudo aponta como objetivo: "O objetivo deste trabalho é conscientizar os desenvolvedores de software sobre as implicações de segurança ao selecionar trechos de código comparando empiricamente as vulnerabilidades do ChatGPT e do StackOverflow.". Para atingir tal objetivo, as seguintes questões de pesquisa foram propostas:

- **RQ1**: Quais são as diferenças de vulnerabilidades entre trechos de código do ChatGPT e do SO?
- **RQ2**: Quais tipos de vulnerabilidades em termos de tipos de CWEs estão presentes para código gerado pelo ChatGPT versus respondido pelo SO?

A metodologia seguiu 5 passos principais:

1. **Seleção de Plataformas**: **ChatGPT** e **StackOverflow**, por serem plataformas centrais e agregadores de, respectivamente, solução em geração automatizada não controlada de código e discussões de problemas e soluções no ecossistema de desenvolvimento de software

2. **Seleção de Perguntas e Respostas**: A partir do **SO** e da API do *StackExchange*, mineraram 1216 perguntas e 1377 respostas associadas em fóruns de perguntas relacionadas a segurança em Java, uma linguagem ainda proeminente na construção de sistemas de código aberto em larga escala.

3. **Filtragem de Código**: Grande parte dos trechos de código foram filtrados de acordo com sua composição nos fóruns do **SO**: códigos em apenas uma linha ou com apenas um nome de classe (não compiláveis em Java *by default*). Com os *snippets* remanescentes, anotaram aqueles os quais erros de compilação persistiram por falta de importação de módulos, de modo que alteraram os trechos importando esses módulos. Ao todo, sobraram 189 perguntas e 204 respostas.

4. **Geração de Respostas com o ChatGPT**: A partir das questões filtradas, o prompt "For all the following questions, generate a compilable code snippet in Java. [Questão]" foi usado para sintetizar as soluções via ChatGPT-3.5 Turbo. Ademais, adicionaram restrições de que o código gerado deve ser compilável, de modo que, adicionado a etapas de filtragem processuais, o conjunto resultante de avaliação contou com 87 questões e 90 respostas.

5. **Detecção de Vulnerabilidades**: Por fim, usaram uma ferramenta de análise estática de programas para detecção das CWEs. Mais especificamente, a ferramenta CodeQL, uma *engine* de análise de código desenvolvida pelo GitHub, usada para avaliação de código gerado por LLMs em pesquisas anteriores e possui performance exemplar em análise estática de programas Java.

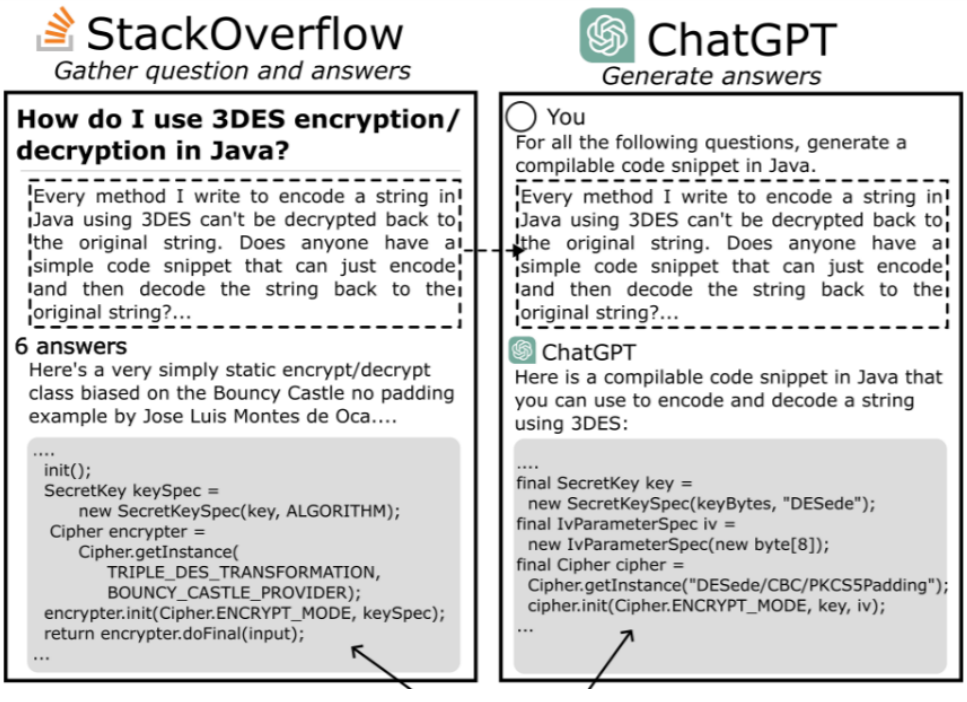


Figura 11 – Esquema Comparativo de Trechos de Código do ChatGPT e StackOverflow

Resultados

Inicialmente, de forma interessante, encontraram vulnerabilidades em 79 resultados de perguntas para o ChatGPT *versus* 77 para trechos de código provenientes do **SO**, com uma sobreposição de 81%, ou seja, uma diferença estatisticamente insignificante para efeitos de pesquisa. Similarmente, em termos de trecho de código, ChatGPT gerou 87 trechos com vulnerabilidade comparado 83 trechos do **SO**.

Contudo, uma diferença significativa foi observado ao se comparar o número total de vulnerabilidades, de modo que 248 foram encontradas nos códigos gerados pelo ChatGPT em comparação com um número 20% maior para a plataforma **SO** (302 vulnerabilidades)

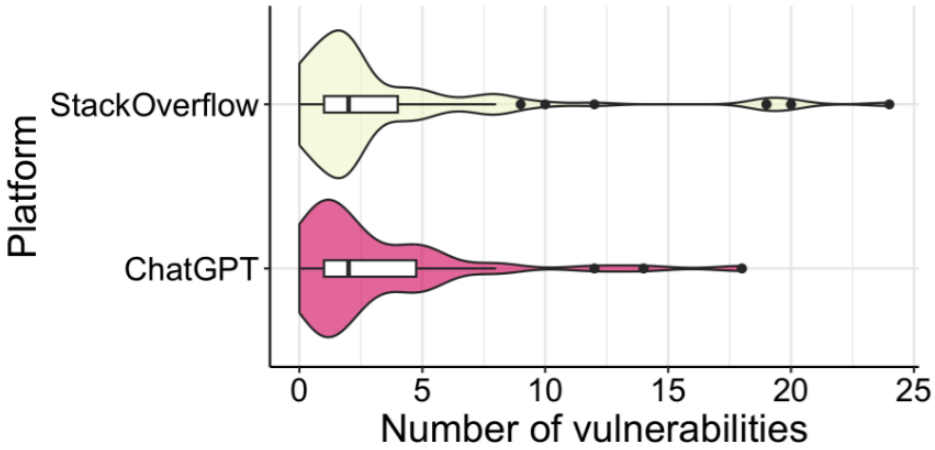


Figura 12 – Violin Chart do Número de Vulnerabilidades por Code Snippet da Plataforma

No que tange a figura 12, métricas relevantes foram extraídas, como: **1)** o número médio e mediano de vulnerabilidades por snippet para ChatGPT foram 3,04 e 2, respectivamente. Em contraste, o número médio de vulnerabilidades para snippets de código **SO** foi 3,62, enquanto 2 foi a mediana, reforçando a noção de maior número de vulnerabilidades em códigos da plataforma de fóruns. **2)** O número de vulnerabilidades únicas nos trechos de código é 158 para o ChatGPT e 183 para **SO**. As vulnerabilidades geradas pelo ChatGPT e presentes nos trechos de código do **SO** são apenas as mesmas em 25% dos trechos, indicando um ponto de preocupação e pesquisa para vulnerabilidades específicas para código gerado por IA. Notavelmente, há uma diferença de pelo menos 54% entre a interseção de vulnerabilidades únicas em trechos e as outras sobreposições.

Acerca dos tipos de vulnerabilidades CWEs encontradas no ChatGPT, os resultados indicaram que:

- 19 CWEs foram indentificadas para os resultados do LLM. O tipo mais frequente foi o **CWE-327**: Uso de um Algoritmo Criptográfico Quebrado ou Arriscado (captura a insegurança de algoritmos criptográficos) e **CWE-328**: Uso de Hash Fraco (Funções de Hash não atendem expectativas de segurança), ambos empatados em primeiro lugar com 95 snippets associados;

- Empatado em segundo lugar está **CWE-404**: Liberação imprópria de recursos e **CWE-772**: Falta de liberação de recursos após a vida útil efetiva, encontrado em 37 snippets. Essas CWEs, por sua vez, capturam noções de segurança ao lidar com recursos voláteis em código;

- Por fim, o terceiro mais presente é o **CWE-798**: Uso de credenciais codificadas. A vulnerabilidade descreve quando credenciais como senhas ou chaves criptográficas são codificadas no código (”hardcoded”) e podem ser aproveitadas por invasores para ignorar a autenticação. Além disso, o **CWE-798** está em 18º lugar no Top 25 do MITRE[14].

Para o **SO**, encontraram 22 tipos de **CWEs**, ou seja, o ChatGPT produziu 14,63% CWEs a menos que a plataforma de fóruns. Nesse cenário, de forma interessante, a ordem dos CWEs mais comuns é a mesma para o ChatGPT e **SO**: **CWE-327** e **CWE-328** empatados em primeiro lugar com 122 snippets, **CWE-404** e **CWE-722** em segundo lugar com 29 snippets e **CWE-798** em terceiro com 20 snippets. No entanto, o número de vulnerabilidades para os CWEs mais frequentes encontrados em cada plataforma diferiu com um desvio de 8 a 27. Para os CWEs mais comuns, ChatGPT criou menos vulnerabilidades para **CWE-327** e **CWE-328**. Enquanto isso, **SO** produziu menos vulnerabilidades para **CWE 404**, **CWE-772** e **CWE-798**. Os snippets do ChatGPT foram melhores que os do **SO** para 15 tipos de CWEs. Em contraste, o **SO** foi melhor para 8 tipos de CWEs.

Uma análise de relevância estatística focada nos CWEs revelou que 5 deles atingiram tal significância Em ordem decrescente nesta análise, seguem-se estes CWEs:

1. **CWE-335**: Uso Incorreto de Sementes em Geração Pseudoaleatória de Números
2. **CWE-835**: Loop com Condição de Saída Inatingível (Loop Infinito)
3. **CWE-798**: Uso de Credenciais ”hardcoded”
4. **CWE-570**: Expressão Sempre Falsa
5. **CWE-571**: Expressão Sempre Verdadeira

Por fim, a seção de discussões aponta que o uso de LLMs no desenvolvimento de software apresenta oportunidades e riscos. Os desenvolvedores geralmente são mal informados sobre o potencial de propagação de código inseguro dessas plataformas, que podem conter vulnerabilidades, os quais os pesquisadores identificaram 274 em 25 tipos de CWE nessas plataformas, destacando os riscos de incorporar fontes externas nos projetos de software. Nesse sentido, as principais recomendações dos pesquisadores incluem: **1)**: Não confiar cegamente em código — seja gerado por IA ou criado por humanos — de fontes externas, pois ele pode se tornar um vetor de ataque. **2)**: Usar LLMs apesar dos riscos, mas aplicando práticas robustas de segurança de software, como ferramentas de análise estática e testes de software, para identificar vulnerabilidades em código copiado. Os desenvolvedores devem priorizar o tratamento de vulnerabilidades como **CWE-335**, **CWE-570** e **CWE-571** em código gerado por LLM, e **CWE-798** e **CWE-835** em respostas do Stack Overflow (SO). Adotar práticas de segurança de alto impacto é essencial para mitigar riscos ao aproveitar LLMs.

4.4 LLMs em Depuração de Código

4.4.1 DebugBench: Evaluating Debugging Capability of Large Language Models

Introdução

Este estudo[19], publicado em 2024, constrói e aplica um benchmark de problemas a serem depurados por modelos de linguagem, com o objetivo de avaliar a capacidade dos LLMs, de código aberto e fechado, de depuração de código. De início, o estudo atesta que demais avaliações feitas acerca da capacidade de *debugging* dos LLMs estavam sujeitas a vazamento de dados, de modo que as próprias instâncias de testes poderiam estar presentes nos dados de pré-treino dos modelos de linguagem, bem como eram limitadas em escala do *dataset* de testes e pouco variadas em tipos de *bugs* e modelos testados.

Durante o processo de desenvolvimento de software, a escrita do código em si age como uma janela entre duas etapas de grande demanda temporal e carga mental dos programadores: construção lógica da solução e depuração do código, de modo que, de acordo com o estudo, consome de 35% a 50% do tempo de desenvolvimento e 50-75% do orçamento de um projeto de software, o que, com o desenvolvimento rampante dos LLMs, abre oportunidades para melhorias na pesquisa de ferramentas de depuração automatizada de código. Nesse cenário, os estudos apontados de avaliação da capacidade de depuração de LLMs, além de não consolidarem um *benchmark* de avaliação para tal, apresentam *gaps* de pesquisa como:

1. **Vazamento de Dados**: Como mencionado, devido a natureza de treinamento dos modelos de linguagem, as próprias massas de teste usadas em avaliações anteriores podem fazer parte do corpo de treinamento dos LLMs, recuperado via *web-scraping*. Por exemplo, foi verificado que o ChatGPT (2023), pode enumerar todos os projetos do Defects4J[20], uma coleção de aproximadamente 850 (open-source) bugs reproduzíveis e uma infraestrutura de suporte.
2. **Diversidade de Bugs Apllicados**: Estudos anteriores focavam mais em reportar taxas de conserto gerais ao invés de explorar cenários de bug variados
3. **Outros**: como variabilidade de LLMs aplicados e escalas reduzidas de testes.

Este estudo, portanto, propõe a criação de um *benchmark* de avaliação da capacidade de depuração de LLMs que sobrepõe essas limitações observadas.

Construção do DebugBench

De início, para a construção de um corpo de testes que mitiga ao máximo o efeito de vazamento de dados discutido, 3206 trechos de código (em **C++**, **Java** e **Pyhton**) de soluções submetidas por usuários para a plataforma LeetCode, um site popular de desafios de programação de múltiplos tópicos e dificuldades, de modo que cada trecho (*snippet*) continha o enunciado do problema, a solução correta submetida, exemplos e data de lançamento do problema na plataforma. Para minimizar o efeito de vazamento de dados para os modelos testados, todas as instâncias coletadas foram após junho de 2022, com uma data média de lançamento em abril de 2023, uma vez que a data limite dos dados de pré-treino dos modelos *closed-source* usados (GPT-3.5-Turbo-0163 e GPT-4-0163) foi, oficialmente, no mês de setembro de 2021. Além da prevenção do vazamento de dados, o LeetCode foi escolhido devido a outras razões como: dificuldade dos problemas suficientemente difícil até para LLMs mais poderosos/estado-da-arte, bem como a plataforma provém test suites robustas que facilitam avaliações automatizadas.

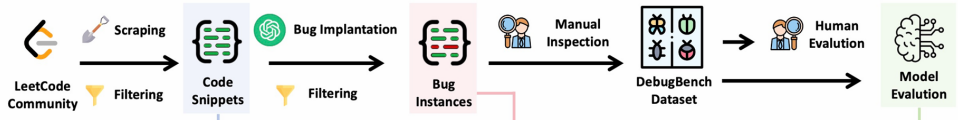


Figura 13 – Overview dos Passos de Construção do DebugBench

Em seguida, a partir das instâncias coletadas, usaram o GPT-4 para introduzir bugs às soluções dos problemas. Para cada trecho de código (solução), foi implantado um ou vários bugs em sequência por meio de um prompt para o LLM em questão que continha, além do trecho em si, instruções para introduzir o(s) tipo(s) de bug(s) desejado(s) e para gerar algumas frases que explicam o(s) bug(s) inserido(s). Nesse sentido, os bugs em questão foram categorizados em 4 tipos principais: **Sintaxe**, **Referência** (métodos e objetos indefinidos, por exemplo), **Lógica** e **Múltiplos Bugs** (de 2 a 4 no mesmo trecho), dentre os quais estão contidos 18 subtipos. Para efeitos de controle de qualidade, seguiram-se etapas de filtragem automática e manual:

1. : As soluções recuperadas do LeetCode devem estar corretas, ou seja, passam 100% na test suite da plataforma, as instâncias devem conter informações necessárias como linguagem de programação, id da questão e data de lançamento na plataforma (após, no mínimo, Julho de 2022);

2. : Após a implantação de bugs nos códigos filtrados, filtraram novamente aqueles trechos que falham alguns casos de testes para serem considerados "bugados", de modo que o código não pode conter comentários reveladores do bug no código bem como a explicação sobre o bug, gerada pelo LLM, deve ser completa e relevante para o tipo de bug introduzido;

3. : Em uma etapa de avaliação manual, os bugs são validados na medida em que devem produzir as falhas esperadas, de modo que casos de teste específicos não devem passar para alinhar com o bug assinalado e descrito,

4. : Por fim, informações sensíveis, como dados pessoais, foram apagados e demais ajustes manuais foram feitos para alinhar os bugs o máximo possível com cenários reais, como, por exemplo, remoção de comentários não filtrados que revelem a localização ou o tipo do bug.

Ademais, os pesquisadores se atentaram para a possibilidade de que os bugs gerados sejam artificiais demais quando comparados a bugs vistos em codebases do mundo real. Para endereçar esse problema, conduziram uma análise comparativa entre as características dos bugs gerados e as características de bugs encontrados em um dataset de bugs reais, mais especificamente, o QuixBugs[21]. Os resultados dessa análise mostraram que: **1)**: os bugs sintetizados até então mapeam, em tipos, para a maioria dos bugs registrados no QuixBugs (por exemplo,

"Incorrect Variable"do QuixBugs mapeia para o tipo definido "variable error"do DebugBench), e **2)**: ao solicitar o GPT-4 e o LLM Claude 3 Sonnet para identificar bugs do mesmo tipo em ambos os datasets, foi observado que, na maioria dos casos, os bugs sintetizados para o DebugBench são mais prováveis de serem encontrados no mundo real.

Avaliação dos Modelos de Linguagem

Os modelos avaliados foram dois de código fechado: **gpt-3.5-turbo-0613** e **gpt-4-0613**, e quatro modelos de código aberto: **CodeLlama-7b-Instruct**, **Llama-3-8B-Instruct**, **DeepSeek-Coder-33B-Instruct** e **Mixtral-8x7B-Instruct**. A métrica principal de avaliação nos modelos foi baseada na taxa de sucesso dos casos de teste da plataforma LeetCode. Mais especificamente, o **PassRate** dos modelos de linguagem é definido por

$$PR = \sum_{i=0}^{n} \frac{\bigwedge_{j=0}^{m} \left[a_{\theta_i^*} \left(x_i^j \right) = y_i^j \right]}{n} \times 100\%$$

, de modo que, n é o número de instâncias de bug, m é o número de casos de teste de cada bug, (a_{θ_i} representa uma solução de código com bug e $a_{\theta_i^*}$) a sua versão consertada e o par (x_i^j, y_i^j) representa, respectivamente, o input e o output correto para o caso de teste j do bug i. Portanto, a fórmula computa a porcentagem de códigos consertados (passam em todos os casos de teste de um problema) de cada modelo de linguagem testado. Para efeitos comparativos de performance, três programadores com quatro anos de experiência cada depuraram 72 bugs de forma independente, com acesso a IDEs mas sem o uso de modelos de linguagem ou outras ferramentas de aprendizado profundo.

Resultados

Major Category	Minor Type	CodeLlama	Llama-3	DeepSeek	Mixtral	gpt-3.5	gpt-4	human
Syntax	misused ==/=	18.2	58.4	68.6	12.4	70.5	87.9	11/12
	missing colons	23.3	44.2	62.8	25.6	80.9	93.6	12/12
	unclosed parentheses	27.1	51.9	86.5	14.3	81.2	89.6	12/12
	illegal separation	7.4	61.8	77.9	17.6	78.1	89.0	12/12
	illegal indentation	4.4	42.2	77.8	28.9	79.6	87.8	12/12
	unclosed string	28.8	48.0	94.4	9.6	82.0	91.4	12/12
	illegal comment	31.5	41.1	45.2	12.9	67.4	78.0	11/12
Reference	faulty indexing	27.2	53.4	67.5	11.7	72.9	77.1	10/12
	undefined objects	21.9	54.5	68.4	4.3	70.6	81.7	12/12
	undefined methods	15.0	46.7	43.7	6.6	59.3	78.5	11/12
	illegal keywords	58.1	13.5	57.3	18.5	76.1	83.6	11/12
Logic	condition error	13.5	46.5	47.7	22.3	58.5	73.1	10/12
	operation error	8.3	28.3	27.8	3.3	49.5	68.6	10/12
	variable error	10.0	29.0	38.0	10.0	52.3	63.1	9/12
	other error	8.0	40.0	44.0	2.0	61.1	72.2	10/12
Multiple	double bugs	3.3	43.2	46.1	8.4	56.4	70.7	11/12
	triple bugs	6.7	29.3	54.5	5.6	45.5	58.9	9/12
	quadraple bugs	5.0	31.2	49.2	4.5	38.7	55.9	8/12

Figura 14 – Resultados do DebugBench

1. **Efeito dos Tipos de Bug**: Inicialmente, o desafio da depuração varia significativamente com o tipo de bug para humanos e modelos. Erros de sintaxe e referência são comparativamente mais simples de detectar e retificar. Em contraste, bugs de lógica representam um desafio maior, exigindo uma compreensão dos mecanismos subjacentes do código. Além disso, a complexidade da depuração aumenta com o aumento do número de bugs dentro de um trecho de código;

2. **Modelos de Linguagem Closed-Source**: Acerca dos modelos comerciais **GPT-3.5 Turbo** e **GPT-4**, alcançaram, respectivamente, 62.1% e 75% em **PassRate**, ambas pontuações pouco abaixo da performance humana, cujo maior valor pode ser atribuído a casos de teste mais robustos e interatividade com o programa via *breakpoints* e ambientes de desenvolvimento. Apesar da eficácia limitada na correção de bugs, apresentaram a vantagem clara de eficiência temporal, de modo que completaram os processos de inferência

para um bug em menos de 10 segundos em média, ao passo que um humano levaria em torno de 20 minutos para o mesmo exercício, ou seja, as versões dos LLMs analisadas neste estudo são parcialmente capazes de atingir o objetivo de depuração automatizada;

a) **Efeito da Amostragem Múltipla**: Especificamente para os modelos comerciais, os pesquisadores analisaram o cenário onde os LLMs foram permitidos a gerar várias "tentativas"(máximo de resposta para uma única query de depuração, de modo que uma instância de bug é considerada consertada se pelo menos uma dessas tentativas de conserto passe em todos os casos de teste. De modo geral, os resultados apresentam um *trade-off* entre um aumento consistente de 20% de PassRate da primeira até a nona tentativa, e o custo financeiro e computacional do processamento extra de tokens;

b) **Efeito do Feedback Informacional em Tempo de Execução**: Ainda no contexto dos modelos comerciais, os pesquisadores recuperaram informações em tempo de execução fornecidas pelo ambiente de testagem do LeetCode, de modo a fornecer esse conteúdo textual como input auxiliar para os LLMs. Nesse sentido, para os bugs de sintaxe e referência, informações textuais como traceback de erros foi de grande serventia para melhorar o PassRate dos LLMs, de modo entre 1 e 6 pontos percentuais foram ganhos. Em contrapartida, essa estratégia prejudicou a métrica de PassRate para bugs de lógica, uma vez que os detalhes fornecidos em mensagens de traceback são frequentemente muito baixo nível para facilitar a depuração e podem até causar interrupções,

c) **Comparação entre Geração e Depuração de Código**: Em última análise, os pesquisadores aplicaram os mesmos problemas cujas instâncias de bugs foram sintetizadas para solucionar via os LLMs comerciais. Nesse quesito, descobriram que corrigir erros de sintaxe e referência geralmente apresenta menos dificuldade do que gerar código completo para uma query específica, enquanto lidar com erros lógicos ou problemas múltiplos pode ser tão desafiador quanto a geração de código em si, de modo que esse padrão implica que a tarefa de depuração é relativamente mais fácil que a de geração de código. Além disso, por meio de cálculos de correlação, o estudo investigou se um problema de programação é tão fácil/difícil de gerar quanto de depurar, de modo que há uma correlação positiva de dificuldade entre as duas tarefas, especialmente para bugs de lógica.

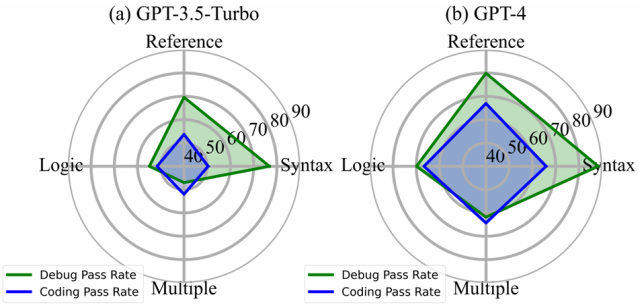


Figura 15 – Comparação de PassRate de Códigos Gerados e Depurados

Conclusões e Limitações

O estudo conclui que, embora os LLMs demonstrem capacidades significativas na geração de código, seu desempenho de depuração permanece abaixo do dos programadores humanos. Os modelos avaliados alcançaram taxas de sucesso variadas, com modelos comerciais superando os de código aberto. Apesar de suas limitações na depuração eficaz de bugs lógicos complexos, os LLMs mostram-se promissores em termos de eficiência de tempo, concluindo tarefas de depuração muito mais rápido do que os humanos. As descobertas sugerem que aprimorar as capacidades de depuração dos LLMs pode contribuir para avanços em ferramentas de depuração automatizadas, particularmente ao melhorar sua capacidade de escrever casos de teste confiáveis e interagir com IDEs.

O estudo reconhece várias limitações. Primeiro, os bugs usados nos experimentos foram criados sinteticamente, o que pode não representar totalmente as complexidades dos cenários de codificação do mundo real. A avaliação

foi limitada a uma pequena seleção de quatro modelos de código aberto e dois de código fechado, o que não abrange toda a gama de LLMs existentes. Além disso, a métrica de avaliação do benchmark dependia de casos de teste da plataforma LeetCode, que, embora de alta qualidade, podem não ser tão acessíveis quanto os casos de teste locais.

4.4.2 DebugEval: Enhancing the Code Debugging Ability of LLMs via Communicative Agent Based Data Refinement

Introdução

Na mesma temática de aplicação e avaliação de LLMs para depuração de código, este estudo em questão[22] introduz o **DebugEval**, um benchmark compreensivo para testar as capacidades de depuração de código de modelos de linguagem a partir de datasets robustos e avaliação em quatro frentes principais: localização do bug, identificação do bug, review de código e reparação do código. Além disso, para fins de fine-tuning, foi proposto um framework baseado em um agente comunicativo LLM para refinamento de dados de depuração (**MASTER**) para prover insights contextuais importantes para o modelo de linguagem usado diretamente para depuração do código, como descrições em linguagem natural de soluções alto nível a serem codificadas pelo LLM alvo.

Em referência à proposta do estudo anterior, os pesquisadores deste estudo, de início, apontaram as principais limitações construtivas dos benchmarks de avaliação de LLMs para depuração de código: **1)**: o design é centralizado na reparação de código, o que é considerado insuficiente para uma avaliação compreensiva das capacidades de depuração e **2)**: construir bugs com LLMs como o GPT-4 falha ao capturar nuances e especificidades de falhas de código em cenários do mundo real. Nesse sentido, então, o benchmark proposto **DebugEval**, para cada tarefa cujo código deve ser depurado, incorpora código em **Python**, **C** e **Java**, bem como mescla as abordagens de introdução de bugs artificialmente com o GPT-4 e recuperação de bugs reais em datasets públicos, atentando-se à possibilidade de vazamento desses datasets para o treinamento dos LLMs avaliados.

Além disso, o framework **MASTER** supracitado define quatro agentes para produção de dados para SFTs (Fine-Tuning Supervisionado) de modo a potencializar a qualidade de depuração dos LLMs. Esses quatro agentes definem uma pipeline que, em suma, geram problemas de depuração de código, que, por sua vez, são avaliados de acordo com suas características educacionais, revisados e por fim solucionados e explicados em detalhe para servirem como dados SFT para o modelo de depuração autoral **NeuDebugger**.

Cosntrução do DebugEval

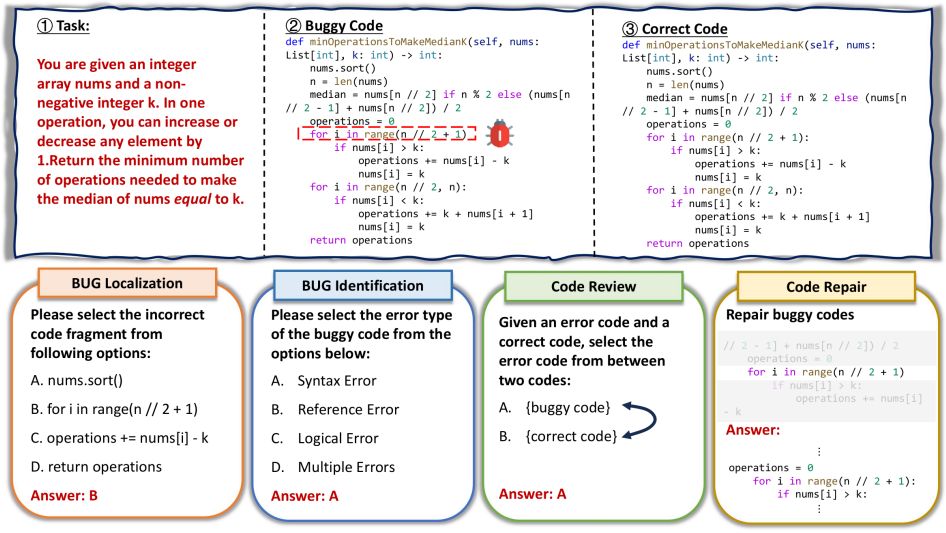


Figura 16 – Esquema Geral do DebugEval

O BenchMark proposto subdivide-se em quatro principais passos, emulando o processo intrínseco de depuração que um programador humano faria:

1. **Localização**: O LLM é fornecido com 4 trechos específicos de um código que contém um bug e então solicitado para identificar qual o trecho chave que gera o bug nesse contexto. A construção de dados para esta etapa consistiu em amostrar instâncias de teste do DebugBench[19]. Mais especificamente, o conjunto de dados foi construído por amostragem de até 20 instâncias de cada um dos 15 tipos de erro de código único para cada linguagem de programação, de modo que o código com bugs foi comparado com o código correto para encontrar o snippet que contém erros como a escolha certa. Para construir as outras escolhas confusas, o snippet de código com bug foi retirado e realizou-se outra amostragem aleatória, dessa vez de uma linha de código como uma das escolhas confusas.

2. **Identificação**: Encontrado o trecho do bug, o LLM deve identificá-lo de acordo com quatro categorias (similares ao do estudo DebugBench): Erro de sintaxe, referência, lógica ou múltiplos bugs. De forma similar ao passo anterior, novas amostras de teste foram retiradas do DebugBench, aproveitando o mapeamento direto dos tipos de bugs entre os dois estudos. Para cada linguagem de programação, comparou-se os tamanhos dos conjuntos de amostras de teste para diferentes tipos de erros de código e então foi selecionado o menor tamanho de conjunto como o número de amostragem. Finalmente, instâncias de teste foram amostradas dos diferentes conjuntos de teste de erros de código de acordo com este número de amostragem.

3. **Revisão**: Nesta etapa, o LLM é instruído a escolher dentre duas opções, um código com o tipo do bug em questão e outro implementado corretamente, qual é o código com falhas. Nesse conjunto de dados, foram misturadas instâncias de teste de ambos DebugBench e LiveCodeBench[23], um conjunto de dados de geração de código onde cada instância de teste contém uma descrição da tarefa, código correto e casos de teste para avaliar a correção dos códigos gerados. Então, coletou-se pares de teste do conjunto de dados mistos por amostragem aleatória de 800 instâncias de teste de cada linguagem de programação.

4. **Reparação**: Por fim, com o código com falhas escolhido em mãos, o LLM é instruído a depurar o código e consertá-lo de modo que passe em todos os casos de teste propostos para o problema e questão. Nesta etapa, para testagem definitiva do poder de depuração de código dos LLMs, 138 problemas de programação foram retirados da plataforma de programação competitiva AtCoder[1], lançados entre setembro de 2023 até abril de 2024 com o fito de evitar vazamento de dados para os prórpios modelos de linguagem. Para cada um desses problemas, uma submissão com bugs foi selecionada para cada linguagem (**C++**, **Python** e **Java**) com uma solução correta associada, isto é, que passa em todos os casos de teste da plataforma.

Construção do MASTER

O estudo mostra que emprego de fine-tuning supervisionado têm sido amplo e bem sucedido para melhorar a performance dos LLMs em domínios específicos. Nesse sentido, o "coMmunicative Agent baSed daTa rEfinement fRamework"é proposto pelo artigo como forma de enriquecer em contexto e dados a capacidade de depuração dos modelos de linguagem. Em linhas gerais, a abordagem multi-agente (baseados em prompts instrutivos para LLMs mais potentes) consiste na interação de três agentes que geram, revisam e refinam dados importantes de depuração para usufruto das LLMs avaliadas. Esses três agentes consistem em:

1. **Code Quizzer**: Por meio do prompt "You are a code debugging expert, skilled in generating code debugging problems to challenge programmers", este agente é responsável por gerar problemas de depuração personalizados e de alta qualidade.

2. **Code Learner**: Com o prompt “You are a student, please provide an answer to the following code debugging question using your own knowledge”, este agente serve como um crítico que avalia o valor educacional dos problemas gerados pelo agente anterior, determinado pela capacidade de solução do problema deste agente ao passo que foi solicitado que resolvesse o problema do Code Quizzer, melhorando o processo de fine-tuning.

[1] https://atcoder.jp/

3. **Code Teacher**: Inspirado por [24], o último agente, com o prompt: "You are an experienced and insightful debugger", é direcionado a gerar soluções em forma de cadeia de pensamentos para guiar os LLMs no processo de SFT, ao passo que gera feedback para o Code Learner. Portanto, as respostas finais geradas pelo Code Teacher são os outputs finais que servirão de dados de fine-tuning para o modelo depurador **NeuDebugger** proposto.

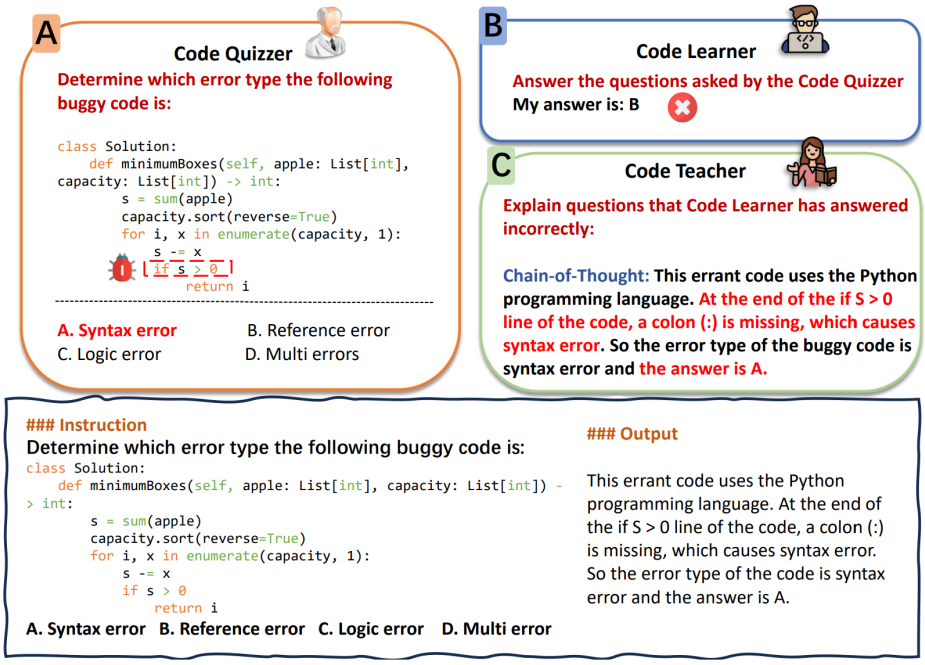


Figura 17 – Overview do MASTER

Em seguida, o estudo descreve que a avaliação dos modelos foi calculada com base na taxa de acerto das questões propostas nas três primeiras etapas do DebugEval (Localização, Identificação e Revisão), juntamente com uma métrica de Pass@k com base em estudos anteriores:

$$Pass@k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

, onde n é o número total de amostras, C é o número de amostras corretas e k é o número de amostras superiores. Foi definido $k = 1$ para este estudo. Dentre modelos de código aberto e fechado, 13 LLMs compuseram o corpo de avaliações: **GPT-4o-mini-0718** e **GPT-3.5-Turbo-0125**, dois LLMs populares e poderosos ao tempo desta pesquisa; modelos de linguagem **Llama** (Meta) mais recentes, incluindo um modelo tunado para tarefas de código e demais modelos poderosos como selecionados da linha *Qwen* e da *DeepSeek*, além do modelo tunado **NeuDebugger**.

Resultados

Em primeira análise, modelos de larga escala como ambos GPTs e modelos da *DeepSeek* com maior quantidade de parâmetros performaram melhor em geral. Em suma, modelos que excedem os 70 bilhões de parâmetros demonstraram performance consistente nas tarefas de depuração propostas nas 4 etapas do benchmark. Em contrapartida, a maioria dos modelos de linguagem de 7 bilhões de parâmetros em escala atingiram menos de 30% de precisão nas tarefas de múltipla escolha (localização e identificação).

Ainda nesse contexto, os pesquisadores escolheram os modelos **DSCoder-6.7B-Ins** e **Llama3-8B-Ins** como base para compor os modelos propostos **NeuDebugger-DS-6.7B** and **NeuDebuggerLlama3-8B**, construídos a partir de fine-tuning com os dados sintetizados pelo **MASTER**. Nesse quesito, o modelo **NeuDebugger** apresenta performance comparada aos modelos de 70B de parâmetros, bem como ultrapassa em performance os

LLMs que serviram de base para sua constituição, o que evidencia a importância da construção de dados SFT de alta qualidade para potencializar modelos de menor escala/open-source.

Model	BUG Localization				BUG Identification				Code Review				Code Repair				Avg.
	PY	C++	JAVA	Avg.	PY	C++	JAVA	Avg.	PY	C++	JAVA	Avg.	PY	C++	JAVA	Avg.	
GPT-4o-mini-0718 [65]	84.8	81.0	81.5	82.4	53.3	48.5	48.9	50.2	85.4	90.9	91.0	89.1	65.2	67.2	67.4	66.6	72.1
GPT-3.5-Turbo-0125 [7]	40.4	47.2	52.2	46.9	35.5	33.3	34.1	34.3	79.4	82.4	84.0	81.9	57.2	52.9	61.6	57.2	55.1
DeepSeek-V2-0628 [66]	82.0	81.0	85.9	83.0	62.0	61.0	61.3	61.4	77.4	83.9	80.5	80.6	65.2	63.0	63.5	63.9	72.2
DeepSeek-Coder-V2-0724 [67]	88.8	83.1	89.8	87.2	58.7	58.9	60.8	59.4	87.9	94.9	93.3	92.0	66.7	63.1	62.3	64.0	75.7
Llama3-70B-Ins [68]	74.2	75.9	82.0	77.5	42.8	42.3	44.9	43.3	73.9	61.6	63.3	66.3	44.9	44.2	45.7	44.9	58.0
Qwen2-72B-Ins [69]	79.8	69.2	74.6	74.4	45.8	45.0	41.3	44.1	61.5	75.8	70.4	69.2	43.5	42.0	42.8	42.8	57.6
DSCoder-33B-Ins [64]	52.2	50.3	51.7	51.4	24.9	26.0	30.9	27.2	24.8	27.0	30.5	27.4	46.4	50.7	54.3	50.5	39.1
Llama2-7B-Ins [8]	18.0	20.0	22.4	20.2	24.9	27.0	25.8	25.9	2.3	0.6	2.0	1.6	4.3	11.7	19.6	11.9	14.9
CodeLlama-7B-Ins [37]	27.0	20.0	23.9	23.5	26.1	23.0	23.8	24.3	48.1	60.5	65.6	58.1	18.8	23.2	23.2	21.7	31.9
CodeQwen1.5-7B-Ins [36]	29.2	30.8	38.0	32.9	27.6	25.9	28.8	27.4	26.9	34.4	37.1	32.8	39.1	49.3	52.9	47.1	35.1
DeepSeek-LLM-7B-Ins [70]	27.0	19.0	25.9	23.9	30.5	28.5	30.9	29.9	35.9	36.6	46.0	39.5	21.0	24.1	14.5	19.9	28.3
DSCoder-6.7B-Ins [64]	22.5	25.6	33.7	27.5	26.6	26.0	25.9	26.2	15.5	17.8	27.8	20.3	31.9	43.5	46.4	40.6	28.7
Llama3-8B-Ins [68]	55.6	55.9	61.0	57.6	36.8	38.1	34.6	36.6	69.1	77.4	78.1	74.9	26.1	34.3	28.3	29.6	49.7
NeuDebugger-DS-6.7B	62.4	55.4	59.0	58.8	42.6	46.9	47.8	45.8	71.0	71.4	71.4	71.3	43.5	48.6	56.5	49.5	56.4
NeuDebugger-Llama3-8B	64.6	57.9	61.0	61.1	38.6	29.9	33.3	33.8	75.3	78.0	82.4	78.5	38.4	41.3	45.7	41.8	53.8

Figura 18 – Resultados Principais do DebugEval

Dentre as quatro tarefas definidas para o DebugEval, LLMs normalmente produzem melhores resultados na localização de bugs e tarefas de revisão de código. Por exemplo, o **GPT-4o-mini-0718** alcança Pontuações de precisão de 82,4 e 89,1 nessas tarefas, respectivamente. Isso indica que esses LLMs têm um forte entendimento de código com o apoio do fine-tuning com tarefas de geração de código, permitindo eles identificarem efetivamente os trechos de código com bugs e exibirem melhores habilidades de execução de código. Entretanto, todos os LLMs demonstraram menor eficácia na identificação de bugs e tarefas de reparo de código, que se concentram mais em avaliar a capacidade de depuração de código dos LLMS. A eficácia reduzida dos LLMs nesta tarefa ilustra a dificuldade que os LLMs atuais têm em derivar as raizes de um bug. A tarefa de reparo de código é ainda mais complexa, exigindo que os LLMs localize trechos de código com falha, determine o tipo de bug e depois corrija o código. O desempenho abaixo do ideal desses LLMs de 70B de parâmetros ainda indica os desafios que eles enfrentam em "autodebugging".

Em uma análise mais específica, o estudo discute o impacto da quantidade de dados SFT nas habilidades de depuração dos modelos. Nesse aspecto, para os dois modelos de base para o **NeuDebugger**, também avaliado com o **DebugEval**, comparado ao **Llama3-8B-Ins**, o **DSCoder-6.7B-Ins** mostra uma redução significativa em sua performance de depuração quando mais dados de fine-tuning são aplicados. Isso indica que os LLMs orientados a código (série Llama) são melhores em aprender com os dados de depuração, enquanto um modelo de linguagem padrão tem dificuldade para aprimorar seus recursos de depuração sem um entendimento compreensivo essencial do código. Já acerca das tarefas propostas pelo benchmark, o modelo da *DeepSeek* apresentou melhorias significativas na localização, identificação de bugs e revisão de código, com poucas melhorias na tarefa final de reparação de código, o que sugere essas tarefas de depuração do benchmark realmente contribuíram para o melhor capacidade de reparo de código do LLMS, embora a depuração completa permaneça bastante desafiadora.

Por fim, os pesquisadores relataram diferenças comparativas em termos das diferentes etapas do becnchmark. Assim, na tarefa de identificação de bugs, LLMs atuais ainda apresentam performance subótima. Os modelos base para o **NeuDebugger** alcançam, no entanto, 97% de precisão nos erros lógicos, mas performam abaixo do esperado para outros tipos de erros de programação. Além disso, uma análise da distribuição de respostas dos LLMs revelou que o modelo **Dscoder-6.7b-ins** não tem a capacidade de identificar bugs em seus tipos variados e coloca como default erro lógico, resultando em alta precisão para este tipo de erro específico. **NeuDebugger** mostra sua eficácia ao exibir uma distribuição de escolha de resposta mais equilibrada e alcançar melhorias significativas na identificação de bugs.

Em última instância, **NeuDebugger** atinge uma performance satisfatória para todo tipo de bug, especialmente falhas de sintaxe, pela sua simplicidade de identificação e aprendizado para os modelos de linguagem quando comparado com os demais tipos. **NeuDebugger** também encontra dificuldades ao reparar códigos com erros lógicos ou com múltiplos erros, em consonância com os resultados observados no artigo anterior.

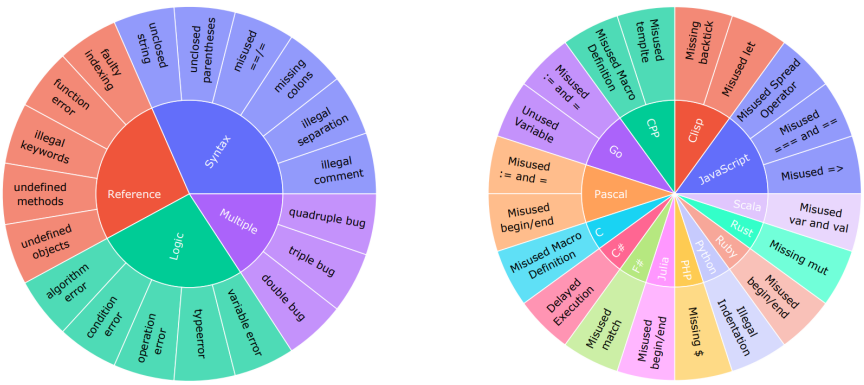
4.4.3 MDEVAL: Massively Multilingual Code Debugging

Introdução

Dando seguimento evolutivo à pesquisa de depuração de código via LLMs, este estudo[25], em uma visão similar, menciona os benchmarks anteriores (DebugBench e DebugEval) e atesta para o fato de que foram poucos diversos em termos de linguagens de programação usadas com cenários específicos para determinadas linguagens ("Misused Macros"em C, "Missing Mut"em Rust, entre outros). Nesse sentido, propuseram um novo benchmark com o mesmo intuito, dessa vez com 3.6K amostras curadas manualmente de 18 linguagens de programação diferentes com tarefas de APR (Automated Program Repair), code review e identificação de bugs. Similar ao **NeuDebugger** visto anteriormente, este estudo propõe o **xDebugCoder**, uma ferramenta LLM orientada a depuração de código de múltiplas linguagens de programação, treinado no **MdEval-Instruct**, um dataset também autoral de instruções de depuração que serviu de auxílio para o trabalho de debugging dos 15+ LLMs avaliados. Mais detalhadamente, esse conjunto de dados foi formado pela ferramenta **xDebugGen**, responsável por criar os pares (programa correto, programa defeituoso) para tunagem de instruções de depuração.

Construção do MdEval

De início, os pesquisadores evidenciam o diferencial da proposta do MdEval ao passo que o benchmark considera tanto erros genéricos adotados nos estudos anteriores (erros de sintaxe, referência, de lógica ou múltiplos), bem como erros específicos da linguagem de programação analisada:



(a) Generic Error Types (b) Language-Specific Error Types

Em uma comparação antecipada, o estudo mostra que, em comparação com demais benchmarks de mesmo propósito (incluindo o **DebugBench** e o **DebugEval**), o modelo proposto **MdEval** excede os predecessores em linguagens de programação abordadas (18 em comparação com uma média de 2-3 linguagens), tipos de erros analisados (39, mais do que o dobro do número máximo comparado) e com a vantagem de que os bugs usados foram anotados manualmente. Nesse aspecto de anotação dos bugs, 13 participantes graduados em ciência da computação foram recrutados para definir problemas, soluções correspondentes e código defeituoso para eles, de modo que seguiram as seguintes instruções **1)**: Escreva um enunciado claro de um problema e defina casos de teste para garantir que esses bugs possam ser efetivamente identificados e **2)**: Categorize esses bugs em múltiplos níveis de dificuldade (fácil, médio, difícil), baseado na complexidade em consertá-los.

Para o corpo principal do dataset, combinando os bugs gerados pelos voluntários, foram coletados e filtrados trechos de código do GitHub. Antes da fase de anotação, agruparam os tipos de erros em uma classe de genéricos e os tipos de erros específicos da linguagem de programação. Os anotadores prosseguiram anotando o código de acordo com os tipos de erros identificados e os métodos de anotação especificados e, para garantir a qualidade da anotação, eles avaliaram o código anotado com base em quatro critérios: dificuldade do problema, ambiguidade, tipo de erro e se ele é solucionável. Além disso, após concluir suas anotações, cada anotador trocou dados com outro anotador para refinamento cruzado, visando minimizar o viés subjetivo e os erros. Quaisquer discrepâncias

entre anotadores foram resolvidas por consenso ou com a contribuição de anotadores experientes. Por fim, três voluntários avaliaram a precisão do benchmark (visando $> 90\%$) e corrigiram erros.

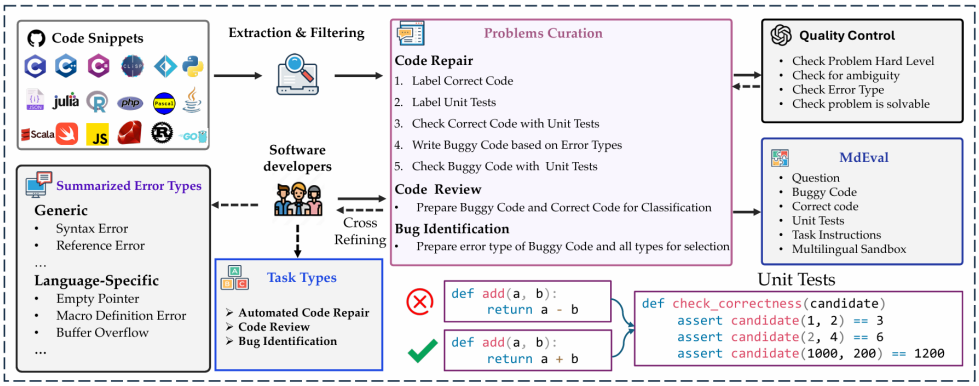


Figura 19 – Overview da Construção do Mdeval

Em seguida, o estudo detalha a construção de um corpo de instruções de depuração **xDebugGen** a ser usado para fine-tuning de um modelo LLM proposto: **xDebugCoder**. Inicialmente, para cada uma entre 18 linguagens diferentes, uma LLM orientada a código é instruída a gerar um par (q^{L_k}, c^{L_k}) de, respectivamente, um problema de programação e um código correto (filtrado em pós-processamento da LLM) para o tal problema na linguagem L_k. Em seguida, o corpo de instruções é formado pelos seguintes 3 passos:

1. **Injeção de Bugs na Query**: Usando a LLM, nesse contexto, a instruíram de modo a modificar o enunciado origina para outra questão similar com diferenças pequenas. Este enunciado, q'^{L_k}, é aplicado em um LLM de menor capacidade M_w (em quantidade de parâmetros) para gerar a resposta $M_w(q'^{L_k})$. Portanto, dada as diferenças entre a questão original e a modificada pela LLM mais poderosa, o par $(M_w(q'^{L_k}), c^{L_k})$ pode ser usado, respectivamente, como predição de código fonte e código alvo na tunagem de instruções;

2. **Injeção de Bugs na Solução**: Nessa abordagem, código defeituoso é diretamente introduzido na solução correta c^{L_k}. Assim, o LLM alvo é instruído a gerar a resposta bugada $M(c^{L_k})$, de modo que o par $(M(c^{L_k}), c^{L_k})$ pode ser usado para tunagem de instruções de depuração,

3. **Injeção de Bugs com Tradução Bidirecional**: Por fim, usaram o LLM alvo para traduzir a resposta correta de uma linguagem k para outra linguagem j e, em seguida, usaram uma LLM de menor capacidade para traduzir o trecho novamente para linguagem k, de modo que o trecho resultante foi considerado defeituoso e então usado juntamente com a resposta correta original para tunagem de instruções. Desse modo, o corpo de instruções de depuração **MdEval-Instruct** é formado.

Em conclusão, as tarefas avaliativas do benchmark construído serão três:

1. **Reparação Automatizada de Código**: Dado o cenário de 18 linguagens de programação, problemas, exemplos de casos de teste e input de código bugado, o LLM é instruído a consertar tal código defeituoso, isto é, fazê-lo passar em todos os casos de teste do problema em questão. Para cada problema, o LLM alvo em questão poderá resolvê-o mediante três cenários: enunciado do problema e código defeituoso como input, código com bugs e casos de teste como input ou somente o código com bugs como input. O LLM é avaliado com a métrica Pass@1, nesse sentido, é pontuado se conserta o código com apenas uma geração.

2. **Code Review**: Neste caso, duas versões do código são apresentadas para o LLM alvo: o correto e o com erros, com pequenas diferenças entre os dois. Listados em ordem aleatória, esses pares de código serão input para o LLM alvo distinguir o código que está com erros.

3. **Identificação de Bugs**: Finalmente, o LLM alvo é requerido a classificar o tipo de erro presente no trecho de código com apenas um erro. A escolha correta deve ser feita entre 39 tipos diferentes, incluindo tipos

genéricos e específicos a linguagens de programação. Para esta tarefa e a anterior, a avaliação é feita usando uma medida simples de precisão de acerto dentre as opções dadas.

16 LLMs foram avaliados, tanto de código fechado quanto de código aberto (tamanhos variando de 1.3B a 236B parâmetros). Para modelos gerais, avaliamos a séria GPT (**GPT4-o**,**GPT4-o-mini**), **Claude-3.5** (Anthropic, 2023). Para modelos orientados a código, **Qwen2.5-Coder**, **DeepSeekCoder** (DS-Coder), **CodeLlama** e **Codegemma** (Gemma Team). Além disso, fine-tuning foi aplicado ao **Qwen2.5-Coder-7B** para fornecer um modelo de linha de base **xDebugCoder** para referência. Para modelos de código fechado, as respostas são geradas pela API oficial, já para os modelos de código aberto, foi realizada inferência em todos os modelos usando a estrutura vLLM.

Resultados

Acerca da tarefa de reparação de código, os resultados indicam que houva uma disparidade marcante entre modelos de última geração de código fechado e a maioria dos modelos de código aberto em quase todas as linguagens de programação. Notavelmente, **GPT-4o**, **Claude-3.5-sonnet** e **DS-Coder-V2-236B** se destacam nessa tarefa e demonstram vantagens significativas de desempenho sobre outros modelos. Além disso, o modelo proposto **xDebugCoder** foi ajustado usando apenas 16K dados relacionados a bugs do dataset **MdEval-Instruct**. Apesar do tamanho limitado desse conjunto de dados, o modelo demonstrou desempenho competitivo em comparação a outros de escala semelhante, destacando a eficácia do **MdEval-Instruct** em aprimorar as capacidades de depuração dos modelos.

Sobre a tarefa de revisão de código, os resultados mostram que os modelos de código fechado ainda superam significativamente os modelos de código aberto na tarefa de revisão de código. Esta tarefa requer que o modelo distinga entre código correto e com bugs. Os modelos de código fechado demonstram uma forte capacidade de entender lógica de código complexa, alcançando uma taxa de precisão de aproximadamente 90%. Em contraste, o modelo menor de código aberto exibe desafios significativos, com uma precisão de cerca de 50%. Esta disparidade ressalta as limitações das opções atuais de código aberto na interpretação eficaz de padrões de codificação intrincados.

Além disso, na tarefa de identificação de bugs, evidenciou-se que, notavelmente, os LLMs de código fechado, como **GPT-4o** e os da série **Claude**, têm vantagens expressivas, superando os LLMs de código aberto em quase +10 pontos. A identificação de bugs com 39 tipos representa um grande desafio para os LLMs, exigindo capacidade de alinhamento dos LLMs entre o snippet de código fornecido e seu tipo de bug correspondente.

Análises mais profundas dos resultados mostraram que, no quesito de performance em diferentes tipos de erros, os modelos demonstram robustez na reparação de erros de sintaxe, referência e lógica, que tendem a serem mais diretos e com escopo mais reduzido, permitindo uma maior precisão dos modelos. Em contraste, os modelos exibem seu pior desempenho ao lidar com erros específicos de linguagem, que podem surgir de regras de sintaxe únicas, expressões idiomáticas ou mesmo práticas de programação cultural que não são universalmente aplicáveis. Como resultado, lidar com esses tipos de erros apresenta um desafio significativo e evidencia a necessidade de mais melhorias nos processos e metodologias de treinamento dos modelos.

Por fim, a análise de performance dos modelos em diferentes cenários de input, onde que recebiam somente o código defeituoso para conserto ou recebiam tanto o código com bugs como casos de teste correspondentes, revelou que o ganho de performance com a informação extra dos casos de teste foi apenas marginal para alguns modelos (GPT-4o-240806, Claude-3.5-sonnet-241022, CodeQwen-1.5-7B-Chat, Qwen2.5-Coder-7B-Chat, DS-Coder-V2-Lite-Instruct, Yi-Coder-1.5B-Chat) exceto para o modelo proposto **XDebugGen**, o qual apresentou uma queda de performance média com o contexto extra.

Novamente, os resultados indicaram que há um gap significativo de performance interpretativa e corretiva de bugs entre modelos com número de parâmetros diferentes, especialmente os modelos comerciais, que, ao tempo do estudo, lideraram os benchmarks analisados com bastante folga em relação aos modelos de código aberto. Nesse sentido, a performance desse modelos parece ser determinada fortemente não só pela quantidade de parâmetro que os constituem, mas pelos tipos de dados usados para treino e fine-tuning, de modo que há uma curta janela

Model	Size	Avg_{all}	C	C#	CLISP	CPP	F#	Go	JS	Java
Closed-Source Models & 200B+ Models										
GPT-4o-240806	🔒	72.1	62.5	65.7	80.0	50.0	81.5	42.2	64.3	86.7
GPT-4o-mini-240718	🔒	68.7	57.8	65.7	66.7	53.0	76.9	37.3	60.0	83.3
Claude-3.5-sonnet-240620	🔒	73.1	51.6	87.1	85.0	57.6	84.6	39.8	62.9	80.0
Claude-3.5-sonnet-241022	🔒	74.0	59.4	82.9	83.3	68.2	87.7	32.5	65.7	78.3
DS-Coder-V2-Instruct	236B	72.7	60.9	87.1	70.0	63.6	80.0	44.6	61.4	78.3
1B+ Models										
DS-Coder-Instrcut	1.3B	33.5	43.8	21.4	16.7	37.9	13.8	22.9	45.7	31.7
Yi-Coder-Chat	1.5B	33.0	35.9	38.6	8.3	39.4	9.2	26.5	45.7	36.7
Qwen2.5-Coder-Instruct	1.5B	36.2	29.7	2.9	13.3	33.3	16.9	10.8	58.6	35.0
6B+ Models										
DS-Coder-Instruct	6.7B	52.9	59.4	7.1	53.3	63.6	23.1	37.3	67.1	60.0
CodeLlama-Instruct	7B	31.9	32.8	2.9	23.3	33.3	12.3	15.7	62.9	20
CodeGemma-Instruct	7B	48.9	45.3	2.9	3.3	36.4	20	41	71.4	66.7
CodeQwen1.5-Chat	7B	50.3	37.5	31.4	41.7	43.9	41.5	33.7	55.7	58.3
Qwen2.5-Coder-Instruct	7B	63.4	59.4	57.1	60.0	62.1	29.2	44.6	60.0	85.0
Yi-Coder-Chat	9B	48.3	54.7	7.1	16.7	47.0	16.9	4.8	68.6	80.0
DS-Coder-V2-Lite-Instruct	16B	59.5	60.9	12.9	40.0	57.6	32.3	32.5	65.7	76.7
DS-Coder-Instruct	33B	55.6	70.3	10	56.7	59.1	27.7	14.5	70	76.7
xDebugGen (Our Method)	7B	46.8	53.1	2.9	36.7	54.5	35.4	3.6	54.3	48.3

Method	Json	Julia	PHP	Pascal	Python	R	Ruby	Rust	Scala	Swift
Closed-Source Models & 200B+ Models										
GPT-4o-240806	20.3	82.1	87.5	72.3	90.2	86.7	84.1	80.0	92.2	78.3
GPT-4o-mini-240718-	39.0	85.1	85.9	64.6	88.5	80.0	84.1	71.4	76.6	70.0
Claude-3.5-sonnet-240620	44.1	82.1	78.1	58.5	90.2	85.0	79.4	91.4	92.2	75.0
Claude-3.5-sonnet-241022	40.7	82.1	85.9	60.0	93.4	86.7	82.5	85.7	89.1	78.3
DS-Coder-V2-Instruct	44.1	83.6	71.9	67.7	85.2	75.0	81.0	85.7	87.5	86.7
1B+ Models										
DS-Coder-Instrcut	8.5	46.3	31.2	40.0	49.2	20.0	57.1	32.9	31.2	53.3
Yi-Coder-Chat	16.9	40.3	28.1	24.6	54.1	20.0	52.4	42.9	34.4	36.7
Qwen2.5-Coder-Instruct	28.8	61.2	43.8	46.2	57.4	26.7	57.1	61.4	23.4	50.0
6B+ Models										
DS-Coder-Instruct	23.7	67.2	57.8	58.5	78.7	48.3	82.5	71.4	35.9	63.3
CodeLlama-Instruct	11.9	43.3	34.4	38.5	39.3	21.7	63.5	58.6	23.4	36.7
CodeGemma-Instruct	28.9	68.7	48.4	50.8	73.8	46.7	74.6	75.7	64.1	61.7
CodeQwen1.5-Chat	23.7	64.2	53.1	52.3	68.9	45.0	71.4	77.1	37.5	71.7
Qwen2.5-Coder-Instruct	35.6	74.6	75.0	66.2	78.7	65.0	69.8	77.1	76.6	71.7
Yi-Coder-Chat	25.4	65.7	54.7	52.3	67.2	36.7	74.6	68.6	70.3	70.0
DS-Coder-V2-Lite-Instruct	30.5	70.1	68.8	64.6	73.8	71.7	77.8	82.9	79.7	83.3
DS-Coder-Instruct	28.9	70.1	54.7	50.8	73.8	60	85.7	67.1	76.6	63.3
xDebugGen (Our Method)	23.7	61.2	64.1	46.2	63.9	36.7	65.1	74.3	62.5	68.3

Figura 20 – Resultados % da pontuação do Pass@1 dos modelos na tarefa de reparação de código no **MdEval**

de performance a ser expandida com técnicas de especialização "pós-treinamento"dos modelos ou baseadas em agentes.

4.5 Comparativo entre Linguagens de Programação

4.5.1 A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages

Publicado em 2023, este estudo[26] apresentou uma abordagem pouco explorada pela literatura científica correspondente à avaliação e aplicação de LLMs para geração de código: um estudo comparativo compreensivo

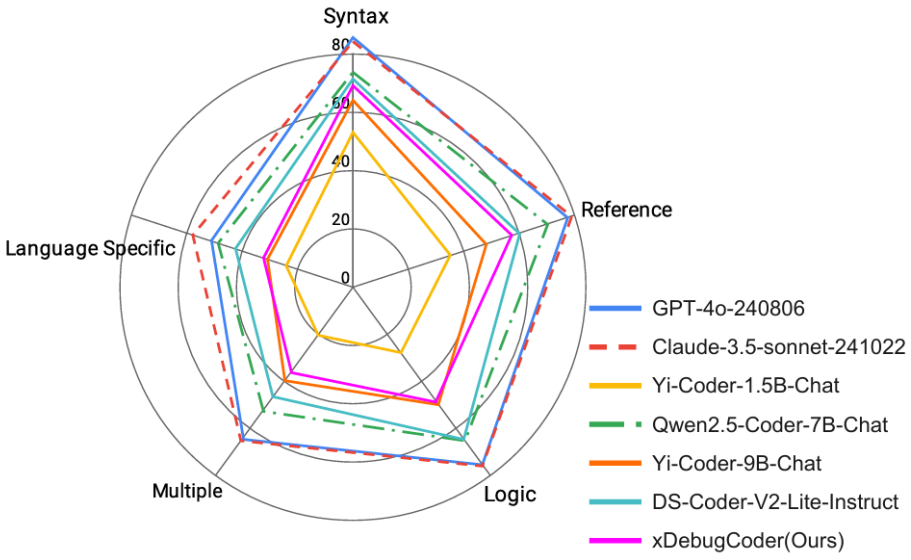


Figura 21 – Performance dos Modelos de Acordo com o Tipo de Erro na Tarefa de Reparação de Código

de atuação de um LLM, nesse caso, o GPT-3.5, entre diferentes linguagens de programação. Em perspectiva, o intuito do autor foi identificar nuances, particularidades e aplicações específicas de 10 linguagens de programação a partir de código gerado com o LLM em questão, dado um conjunto de tarefas de temas variados a serem resolvidas com código, compiladas (se aplicável) executadas e verificadas.

Nesse sentido, este estudo comparativo levou em conta três atributos principais de avaliação das linguagens: performance em tempo de geração do código, comprimento do código e executabilidade do código (compreensão do processo de geração, compilação (se aplicável), execução e verificação), de modo a identificar, dentre as 10 linguagens analisadas, quais são as características fundamentais que contribuem para que uma determinada linguagem seja mais adequada para geração de código via LLMs ou não.

Metodologia

1. **Linguagens Selecionadas**: Com 10 linguagens distintas, o autor evidenciou o detalhe de variar entre diferentes paradigmas (imperativo, OO e funcional), bem como em gerenciamento de memória, características de performance e capacidades especializadas de domínio. Nesse sentido, mostra-se a seleção: **C**, **C++**, **Go**, **JavaScript**, **Julia**, **Perl**, **Python**, **R**, **Ruby** e **SmallTalk**.

2. **Parâmetros e Setup**: As queries para o modelo de linguagem foram automatizadas via API pública da OpenAI para o ChatGPT-3.5 versão Turbo, de modo que o Python 3.11.2 foi usado para enviar as requisições e processar os outputs. Como definição de papel do modelo de linguagem, parâmetro que ajusta o comportamento do modelo durante a conversa, foi preparado como desenvolvedor de software via a string "You are a software developer". Demais parâmetros seguiram o default da API.

3. **Tarefas e Temas**: Com fontes em sites de universidades e plataformas que disponibilizam desafios de programação para entrevistas, um corpus de 40 tarefas foi definido que abrange os seguintes temas: **Data Science** (especialmente algoritmos em processamento de dados e tarefas de classificação); **Games**, de modo que o modelo é instruído a construir 2 versões (uma simples e uma complexa) de jogos conhecidos; **Segurança**, tal que o LLM é desafiado com tarefas que objetivam melhorar a segurança ou simular comportamento de um adversário; **Algoritmos Simples**, exemplos sortidos englobam, por exemplo, computar fatores primos, verificar se um inteiro é um palíndromo, fibonacci, entre outros.

4. **Implementação**: Para uma tarefa T, papel R e linguagem L, seguem-se os passos:

a) Gerar a query "First, write L code to T without importing libraries. Second, write exactly one test called Test for the generated code. Do not include comments within the code.". A query em questão, escrita com comandos sequenciais e assertivos, foi resultado de otimização do pesquisador para produzir respostas estruturadas de modo consistente, bem como para suprimir o uso de bibliotecas externas, normalmente usadas pelo modelo de linguagem em testes preliminares como atalhos convenientes para resolver algumas das tarefas, especialmente aquelas relacionadas a aprendizado de máquina;

b) A partir da query formada Q, um output O é gerado após uma chamada bem sucedida da API do GPT-3.5,

c) Por fim, o output passa por uma etapa de pós-processamento, diferente para cada linguagem, para atingir 2 objetivos: destacamento das seções que contenham apenas código, bem como identificação das etapas de implementação e testagem.

5. **Setup de Testes**: O ChatGPT foi testado 10 vezes para cada uma das 40 tarefas em 10 linguagens de programação diferentes, totalizando 4000 testes distintos. Solicitar o modelo de linguagem a realizar a mesma tarefa múltiplas vezes contribuiu para a produção de resultados estatisticamente significativos e para a percepção de insights acerca do comportamento não determinístico do modelo bem como alguns potenciais e limitações. Todos os testes foram executados via scripts bash, em um mesmo hardware.

Resultados

1. **Resultados Principais**: Antes da análise por linguagem, os outputs produzidos puderam ser categorizados de acordo com 6 status diferentes:

   a) **Nenhum Código Produzido - Questões Éticas**: O modelo de linguagem afirmou não poder gerar o código solicitado devido à diretrizes éticas da OpenAI e/ou poderia ser ilegal. Encontrado em algumas das tarefas relacionadas à segurança;

   b) **Nenhum Código Produzido - Outras Razões**: Tipicamente associado à incapacidade do modelo em gerar o código solicitado;

   c) **Erro de Compilação**: Para linguagens compiláveis (**C**, **C++** e **Go**), o output apresenta código com erro de compilação;

   d) **Falha de Execução**: Similar ao passo anterior, mas com erros de execução;

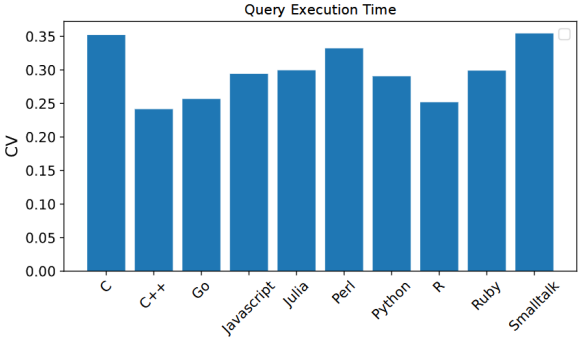
   e) **Execução Indeterminada**: Código gerado executa, mas com resultados inacessíveis dada as condições de máximo de 30 segundos para a execução ou input humano requerido pelo programa, de modo que o programa é interrompido,

   f) **Execução Bem Sucedida**: Código gerado, compilado (se aplicável) e sua execução foi exitosa.

   Com isso, 1833 (45.8%) das 4000 rodadas de teste resultaram em código executável, variando bastante quando analisado dentre as linguagens de programação. Nesse sentido, o ChatGPT performou melhor em **Julia**, com 81.5% de código executável, e pior em **C++**, com apenas 7.3% de código executável produzido. Em geral, houve melhor desempenho em linguagens tipadas dinamicamente de alto nível (**JavaScript**, **Julia**, **Perl**, **Python**, **R**, **Ruby**, **Smalltalk**) em vez de linguagens tipadas estaticamente de mais baixo nível (**C**, **C++**, **Go**). Dentre as linguagens de alto nível, logicamente, o modelo, em média, performa melhor dentre aquelas as quais ele foi treinado com mais recursos, **JS**, **Pyhton** e **Ruby** segundo a própria OpenAI, com 62.8% de executabilidade em média.

   Acerca das categorias dos problemas, a pior performance encontrou-se na categoria de games, para todas as linguagens. As demais categorias apresentam clara variabilidade de performance dentre as linguagens: problemas de segurança viram melhor performance em **C**, **C++** e **SmallTalk**; problemas relacionados a algoritmos simples performaram melhor em **Go**, **JS**, **Perl**, **Pyhton** e **Ruby** e, por fim, o modelo foi exitoso na categoria de Data Science com **Julia** e **R**.

2. **Performance de Tempo**: Neste aspecto, foi avaliado o tempo gasto desde a chamada da API do modelo até um trecho completo de código ser produzido para a solução instruída, o que excluiu instâncias de outputs da categoria **Nenhum Código Produzido**. De modo a evitar produção de estatísticas enviesadas por linguagem de programação com uma média geral ou uma média agrupada por categorias, avaliou-se o desempenho temporal do modelo em relação a uma linguagem em relação ao tempo médio empregado para cada tarefa em todas as linguagens, explicado pela fórmula de performance por linguagem: $P_l = \mu_{t \in T} \frac{G_{l,t}}{\mu_{l \in L,t}}$, para um conjunto L de linguagens, um conjunto T de tarefas, de modo que $G_{l,t}$ expressa o tempo empregado pelo modelo para gerar o código solicitado para a tarefa t na linguagem l e $\mu_{l,t}$ o tempo médio de geração de código para a tarefa t para todas as linguagens em L. A figura a seguir mostra os resultados dessa métrica, aliado a uma métrica **CV** (Coeficiente de Variação), que quantifica a dispersão relativa de uma distribuição de frequência.



Nesse contexto, o LLM demorou 60% mais tempo para gerar código em **C** comparado à média das outras linguagens. Por outro lado, os tempos de resposta para geração de código em **C++** foram aproximadamente metade da média das demais linguagens.

3. **Tamanho do Código**: Apesar da limitação de não prover informações acerca da legibilidade e/ou manutenibilidade do código produzido, bem como não levar em conta a verbosidade de algumas linguagens, essa métrica foi selecionada com a premissa de que, ao observar a extensão do código gerado pelo LLM, certos insights possam ser extraídos acerca da habilidade do modelo de produzir soluções eficientes e concisas. Caso contrário, o modelo pode estar apresentando dificuldades para entender o problema ou para otimizar as soluções propostas.

   Nesse contexto, duas métricas foram analisadas: Linhas de código, excluindo linhas vazias, e número de caracteres, excluindo espaços, de modo que ambas foram calculadas de forma semelhante à P_l. Os resultados mostraram que: **1)** O comprimento do código gerado parece não estar correlacionado com o tempo de resposta do modelo em relação a linguagens de programação diferentes. **2)** Há uma variabilidade maior no comprimento do código do que no tempo de resposta. Isso sugere que o tempo gasto para entender a tarefa, bem como avaliar possíveis estratégias para sua solução, é menos oneroso do que projetar a solução e gerar o código.

4. **Limitações do GPT-3.5**: Como supracitado, mais da metade do código produzido pelo ChatGPT-3.5 apresentou bugs após as solicitações. Além disso, o estudo reportou uma série de comportamentos inconsistentes do modelo:

   a) A compreensão dos requisitos de uma tarefa parece ser parcialmente dependente da escolha de uma linguagem. Por exemplo, quando desafiado na tarefa simplePong em **Python**, o modelo produz 10 vezes em 10 códigos que exigem interação entre dois usuários via linha de comando. Em contraste, em todas as 6 execuções bem-sucedidas de simplePong em **R**, o código permite uma correspondência entre um usuário humano e uma "raquete"controlada por IA.

   b) Em alguns outputs, parte das instruções da query são desconsideradas, tendo como exemplo mais significativo a presença de comentários em alguns códigos.

c) Uma tarefa é considerada antiética em certas instâncias, resultando na ausência de código gerado, mas em outras instâncias o código é produzido. Por exemplo, a tarefa simpleSniffer é considerada antiética 4 vezes em **Go**, enquanto nas outras 6 instâncias o modelo aceita produzir código. Isso mostra que o não-determinismo do modelo não se relaciona apenas à produção de código, mas também à compreensão e avaliação da tarefa.

d) A avaliação das ramificações éticas de uma tarefa parece ser contingente à seleção da linguagem utilizada. Por exemplo, a tarefa bruteForce100 é consistentemente vista como antiética em todos os testes quando executada no **JavaScript**. Por outro lado, ela é classificada como eticamente aceitável em 9 testes de 10 quando implementada em **Go**, e em 8 vezes de 10 quando executada em **Julia**. Em um dos dois casos na linguagem **Julia** em que o modelo considera a tarefa antiética, o código ainda é fornecido com base em propósitos educacionais, acompanhado por um aviso alertando contra seu uso para atividades maliciosas.

Discussões

Por fim, a seção de discussão aborda as implicações de LLMs na geração automatizada de código e seu impacto potencial em linguagens de programação e na indústria de tecnologia. Os pontos principais incluem:

1. **Futuro das Linguagens de Programação**: A proficiência de LLMs em várias linguagens de programação pode influenciar as escolhas das empresas sobre quais linguagens adotar para projetos. Linguagens com maior corpus de treinamento e suporte dos LLMs podem levar a uma geração de código mais rápida e precisa, permitindo que os desenvolvedores se concentrem em tarefas mais críticas. O estudo sugere que a popularidade das linguagens de programação pode se correlacionar cada vez mais com o desempenho do LLM, potencialmente determinando quais linguagens prosperam ou declinam em uso.

2. **Fatores que afetam a competência da linguagem**: A pesquisa indica que o desempenho do ChatGPT é influenciado pelo nível de abstração da linguagem de programação e sua popularidade, o que afeta os dados de treinamento disponíveis para o modelo. Linguagens de nível superior tendem a produzir melhores resultados devido às suas estruturas explícitas e expressivas, o que pode reduzir a complexidade e os erros.

3. **Emprego e considerações éticas**: A integração de LLMs na codificação levanta preocupações sobre o emprego na indústria de tecnologia. Os autores enfatizam a necessidade de programas de requalificação e atualização para ajudar os trabalhadores a se adaptarem às mudanças trazidas pela IA generativa. Além disso, considerações éticas sobre o uso responsável de LLMs são destacadas, enfatizando a importância da transparência, justiça e responsabilidade em seu desenvolvimento e implantação.

5 Fechamento

5.1 Discussão

5.1.1 Fronteiras da Literatura

Necessidade de Estudos em Maior Escala

Papers como [6, 27], conduziram experimentos que produziram insights relevantes acerca de como desenvolvedores abordam desafios e tarefas de codificação com ferramentas de IA generativa, com foco em metrificar e quantizar o entendimento do código dos participantes. Contudo, pesquisadores da área atentam para a necessidade da realização de experimentos similares, mas em maior escala, de modo a capturar nuances e aspectos endereçáveis apenas em cenários de mundo real. Muitos dos estudos nesse sentido, por conta de limitações de recursos/escopo, são direcionados a refinarem e especializarem cenários controlados de uso de LLMs para domínios específicos com

um grupo mais restrito de participantes, o que traz uma abordagem aprofundada acerca das potencialidades em cenários específicos (*prompts* específicos) em contrapartida a uma análise em largura das potencialidades dos modelos de linguagem na execução de tarefas de programadores de diferentes áreas.

Pouca pesquisa para linguagens baixo-nível e design de hardware

Em um escopo mais reduzido e experimental, papers como [28, 29, 30] aplicam *GenAI* em abordagens múltiplas no que tange ao design de hardware, desde a criatividade no design de circuitos em alto nível como desempenho em tarefas de Verilog. As descobertas principais revelaram que, mesmo com os níveis mais caros dos LLMs comerciais, a dificuldade e o custo escalam bastante com o escopo dos problemas propostos, ainda que possuam certa margem a ser explorada na solução de problemas. Nesse sentido, urge estudo investigativo acerca das possibilidades de diferentes LLMs em múltiplas abordagens de construção e design de hardware.

Pesquisas necessárias para entender o impacto do uso contínuo de LLMs para codificação

Nesse sentido, seria importante a aplicação de estudos interdisciplinares e de escopo mais aberto para caracterizar os novos e adaptados perfis de desenvolvedores de software com o advento da IA generativa. De modo a endereçar questionamentos acerca da obsolescência do programador tradicional ou que papel exercem as ferramentas de IA no ciclo de desenvolvimento de um projeto.

Pesquisas necessárias para investigar a capacidade de sintetizar provas de conceito para projetos de software

Sendo uma poderosa e massiva interface linguagem natural para código, poderia-se investigar o potencial dos LLMs na prototipação de projetos e ideias de software de modo ubíquo, tanto para não desenvolvedores validarem ideias de forma mais eficiente, como uma ferramenta potencial para os estágios iniciais de um software.

Pesquisas necessárias para se avaliar o impacto de diferentes estratégias de prompt na performance dos LLMs

Na maioria dos designs de avaliação, benchmarking e aplicações específicas de LLMs, pouca ou nenhum detalhamento e exploração técnica foi aplicada no quesito de *prompt engineering*. Nesse sentido, apesar dos prompts usados para teste serem bem esclarecidos, pouco se estudou o impacto de diferentes estilos de prompt encontrado no cenário real de uso de GenAI para programação.

Aspectos qualitativos negligenciados: compatibilidade, manutenibilidade, portabilidade

Com os saltos rampantes da pesquisa e desenvolvimento de LLMs, grande parte dos estudos focaram esforços em fornecerem a abordagem mais compreensiva possível de avaliação de performance dos modelos de linguagem para geração de código. Contudo, os aspectos centrais, como mencionado neste estudo, são focados na corretude funcional, execução de tarefas específicas, taxas de acerto e similares, com pouco caráter qualitativo do código em si.

5.1.2 Limitações

O conteúdo produzido neste projeto orientado a computação não apresenta um *overview* exaustivo das principais frentes de pesquisa acerca do uso de *GenAI* para codificação, de modo que a seleção de frentes de pesquisa, evidenciada pelas subseções deste trabalho, foi resultado de uma priorização de tópicos considerados mais relevantes para o tema. Nesse sentido, outros tópicos de fronteira de pesquisa constam na literatura, como aplicação de *GenAI* para geração de testes automatizados[31, 32, 33], refatoração de código[34], code review[35, 36, 37], entre outros.

5.2 Conclusão

Este estudo teve como proposta principal representar uma revisão de literatura em menor escala com o objetivo de fornecer um *overview* compreensivo dos esforços e fronteiras da literatura científica no que tange ao uso e aplicações de IA generativa para codificação. Os estudos e suas principais ideias, metodologias, resultados e discussões foram explanados e agrupados em 5 frentes principais: **Entendimento de Código**, **Aspectos do Processo e Resultados da Geração de Código com LLMs**, **Segurança do Uso de IA Generativa Para Código**, **LLMs em Depuração de Código** e **Comparativo entre Linguagens de Programação**. Em suma, tanto leigos quanto experientes em programação devem se atentar à natureza e as capacidades intrísecas dos LLMs para codificação, especialmente para que não superestimem a sua viabilidade, aplicando-os como oráculos ou sem fornecer contextos significativos, mas que também não subestimem suas capacidades de, por exemplo, aceleração de produtividade. Nesse sentido, múltiplos estudos capturaram características semelhantes que capturam o essencial do uso de IA para codificação: a profundidade e especificidade dos outputs depende fortemente da profundidade e especificidade dos prompts fornecidos; existe um *sweetspot* de quanto contexto e liberdade criativa fornecer para IA para produzir resultados mais coerentes, mas, via de regra, informações contextuais são vitais para a geração de respostas mais específicas; existe uma barreira de atuação em tarefas de complexidade maior em todos os ramos, desde geração de código para problemas de programação até depuração de código, reforçando o papel de copilota da inteligência artificial. Os estudos analisados são consoantes no fato de que modelos de linguagem de código fechado, os quais os números de parâmetros exatos são desconhecidos, apresentam performance líder em toda sorte de tarefa aplicada, principalmente a linha **GPT** da **OpenAI**, o que, ao mesmo tempo, soa um alerta acerca dos riscos da falta de transparência dos dados e métodos de treinamento desses modelos.

Referências

[1] ZHU, K. *Ranked: The Most Popular Generative AI Tools in 2024*. 2024. https://www.visualcapitalist.com/ranked-the-most-popular-generative-ai-tools-in-2024/. Accessed: 2024-11-09.

[2] MCKINSEY. *The state of AI in early 2024: Gen AI adoption spikes and starts to generate value*. 2024. https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai. Accessed: 2024-11-09.

[3] OVERFLOW, S. *2024 Developer Survey*. 2024. https://survey.stackoverflow.co/2024/ai/. Accessed: 2024-11-09.

[4] ZIEGLER, A. et al. Measuring github copilot's impact on productivity. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 67, n. 3, p. 54–63, feb 2024. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/3633453>.

[5] JIANG, J. et al. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[6] NAM, D. et al. Using an llm to help with code understanding. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. [S.l.: s.n.], 2024. p. 1–13.

[7] BOKEH. 2024. https://bokeh.org/.

[8] OPEN3D. 2024. https://www.open3d.org/.

[9] MAALEJ, W. et al. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 23, n. 4, p. 1–37, 2014.

[10] WANG, J.; CHEN, Y. A review on code generation with llms: Application and evaluation. In: IEEE. *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. [S.l.], 2023. p. 284–289.

[11] GOOGLE. *LaMDA: our breakthrough conversation technology*. 2021. https://blog.google/technology/ai/lamda/.

[12] XU, F. F. et al. A systematic evaluation of large language models of code. In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. [S.l.: s.n.], 2022. p. 1–10.

[13] REN, S. et al. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[14] MITRE.ORG. *Common Weakness Enumeration*. https://cwe.mitre.org/.

[15] SPIESS, C. et al. Calibration and correctness of language models for code. *arXiv preprint arXiv:2402.02047*, Aug, 2024.

[16] ZHONG, L.; WANG, Z. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. [S.l.: s.n.], 2024. v. 38, n. 19, p. 21841–21849.

[17] ZHENG, D. et al. Towards more realistic evaluation of llm-based code generation: an experimental study and beyond. *arXiv preprint arXiv:2406.06918*, 2024.

[18] HAMER, S.; D'AMORIM, M.; WILLIAMS, L. Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers. In: *2024 IEEE Security and Privacy Workshops (SPW)*. [S.l.: s.n.], 2024. p. 87–94.

[19] TIAN, R. et al. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*, 2024.

[20] DEFECTS4J Github Repository. `https://github.com/rjust/defects4j`.

[21] LIN, D. et al. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. [S.l.: s.n.], 2017. p. 55–56.

[22] YANG, W. et al. Enhancing the code debugging ability of llms via communicative agent based data refinement. *arXiv preprint arXiv:2408.05006*, 2024.

[23] JAIN, N. et al. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[24] WANG, H. et al. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing. *arXiv preprint arXiv:2311.09868*, 2023.

[25] LIU, S. et al. Mdeval: Massively multilingual code debugging. *arXiv preprint arXiv:2411.02310*, 2024.

[26] BUSCEMI, A. A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *arXiv preprint arXiv:2308.04477*, 2023.

[27] RICHARDS, J.; WESSEL, M. What you need is what you get: Theory of mind for an llm-based code understanding assistant. In: IEEE. *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2024. p. 666–671.

[28] DELORENZO, M.; GOHIL, V.; RAJENDRAN, J. Creativeval: Evaluating creativity of llm-based hardware code generation. *arXiv preprint arXiv:2404.08806*, 2024.

[29] MA, R. et al. Verilogreader: Llm-aided hardware test generation. *arXiv preprint arXiv:2406.04373*, 2024.

[30] THAKUR, S. et al. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, ACM New York, NY, v. 29, n. 3, p. 1–31, 2024.

[31] WANG, J. et al. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, IEEE, 2024.

[32] HAJI, K. E.; BRANDT, C.; ZAIDMAN, A. Using github copilot for test generation in python: An empirical study. In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*. [S.l.: s.n.], 2024. p. 45–55.

[33] SCHÄFER, M. et al. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, IEEE, 2023.

[34] LIU, B. et al. An empirical study on the potential of llms in automated software refactoring. *arXiv preprint arXiv:2411.04444*, 2024.

[35] RASHEED, Z. et al. Ai-powered code review with llms: Early results. *arXiv preprint arXiv:2404.18496*, 2024.

[36] YU, J. et al. Security code review by llms: A deep dive into responses. *arXiv preprint arXiv:2401.16310*, 2024.

[37] WANG, L. et al. Unity is strength: Collaborative llm-based agents for code reviewer recommendation. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2024. p. 2235–2239.

[38] MASTROPAOLO, A. et al. On the robustness of code generation techniques: An empirical study on github copilot. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2023. p. 2149–2160.

[39] WERMELINGER, M. Using github copilot to solve simple programming problems. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. New York, NY, USA: Association for Computing Machinery, 2023. (SIGCSE 2023), p. 172–178. ISBN 9781450394314. Disponível em: <https://doi.org/10.1145/3545945.3569830>.

[40] PURYEAR, B.; SPRINT, G. Github copilot in the classroom: learning to code with ai assistance. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, Evansville, IN, USA, v. 38, n. 1, p. 37–47, nov 2022. ISSN 1937-4771.

[41] JOEL, S.; WU, J. J.; FARD, F. H. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*, 2024.

[42] BUSCEMI, A. A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *arXiv preprint arXiv:2308.04477*, 2023.

[43] DU, X. et al. Evaluating large language models in class-level code generation. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. [S.l.: s.n.], 2024. p. 1–13.

[44] YU, J. et al. *An Insight into Security Code Review with LLMs: Capabilities, Obstacles and Influential Factors*. 2024. Disponível em: <https://arxiv.org/abs/2401.16310>.