UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

PEDRO OTÁVIO PROBST ZAMPIER

COMPARATIVO DE MODELOS DE LINGUAGEM AMPLA NO DIAGNÓSTICO AUTOMÁTICO DE ERROS EM FUNDAMENTOS DE PROGRAMAÇÃO

GUARAPUAVA

2025

PEDRO OTÁVIO PROBST ZAMPIER

COMPARATIVO DE MODELOS DE LINGUAGEM AMPLA NO DIAGNÓSTICO AUTOMÁTICO DE ERROS EM FUNDAMENTOS DE PROGRAMAÇÃO

Comparison of Wide Language Models in Programming Fundamentals Automatic Error Diagnosis

Projeto de Trabalho de Conclusão de Curso de Graduação apresentado como requisito parcial para obtenção do título de Tecnólogo em Sistemas para Internet pela Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Eleandro Maschio

GUARAPUAVA

2025

RESUMO

O ensino de programação em cursos superiores enfrenta desafios significativos relacionados às elevadas taxas de reprovação e evasão, parcialmente atribuídas às dificuldades no diagnóstico de erros conceituais em códigos de iniciantes. Modelos de linguagem ampla têm demonstrado capacidades promissoras na análise automatizada de código, porém estudos concentram-se em modelos individuais sem análises comparativas abrangentes que considerem especificamente o contexto educacional. Este trabalho propõe estabelecer um comparativo sobre a capacidade de diagnóstico automatizado de erros em fundamentos de programação por quatro modelos de linguagem ampla proeminentes (GPT-4, Claude, Gemini e DeepSeek), utilizando TypeScript como linguagem-alvo. A metodologia baseia-se na classificação de problemas de compreensão em códigos corretos, adaptando um subconjunto de erros frequentes identificados na literatura para as especificidades da linguagem TypeScript e construindo um conjunto padronizado de códigos de teste que contenham estrategicamente os erros selecionados. Serão realizadas avaliações quantitativas mediante percentual de acerto na identificação e diagnóstico dos erros, estabelecendo métricas de desempenho, e avaliações qualitativas considerando clareza e relevância pedagógica do *feedback* fornecido por cada modelo. Os resultados serão correlacionados com os custos operacionais de cada modelo, detalhando custos por análise e projeções para diferentes volumes de uso. Como contribuição prática, pretende-se implementar um protótipo de mínimo produto viável de interface de programação de aplicação utilizando Laravel, integrado ao modelo de melhor desempenho geral identificado no comparativo. Os resultados parciais incluem a seleção e configuração de três modelos (GPT-4, Claude e Gemini) via suas respectivas interfaces de programação de aplicação, com testes de requisição e validação de carga realizados, definição do subconjunto de erros baseado em critérios de gravidade e frequência, e andamento na adaptação desses erros para TypeScript, considerando diferenças sintáticas e semânticas em relação à linguagem original da classificação. Busca-se fornecer evidências empíricas que orientem escolhas tecnológicas no desenvolvimento de ferramentas educacionais baseadas em inteligência artificial para o ensino de programação, contribuindo para melhor integração entre conhecimento humano e potencialidades da inteligência artificial no contexto educacional.

Palavras-chave: modelos de linguagem ampla; ensino de programação; diagnóstico automatizado de erros; typescript; inteligência artificial na educação.

ABSTRACT

Programming education in higher education faces significant challenges related to high failure and dropout rates, partially attributed to difficulties in diagnosing conceptual errors in beginner code. Large language models have demonstrated promising capabilities in automated code analysis, but studies focus on individual models without comprehensive comparative analyses that specifically consider the educational context. This work proposes to establish a comparison of the automated diagnostic capacity for programming fundamentals errors by four prominent large language models (GPT-4, Claude, Gemini, and DeepSeek), using TypeScript as the target language. The methodology is based on the classification of comprehension problems in correct codes, adapting a subset of frequent errors identified in the literature to the specificities of the TypeScript language and building a standardized set of test codes that strategically contain the selected errors. Quantitative evaluations will be conducted through accuracy percentage in error identification and diagnosis, establishing performance metrics, and qualitative assessments considering clarity and pedagogical relevance of the feedback provided by each model. Results will be correlated with the operational costs of each model, detailing costs per analysis and projections for different usage volumes. As a practical contribution, a minimum viable product prototype of an application programming interface will be implemented using Laravel, integrated with the best overall performing model identified in the comparison. Partial results include the selection and configuration of three models (GPT-4, Claude, and Gemini) via their respective application programming interfaces, with request tests and payload validation performed, definition of the error subset based on severity and frequency criteria, and progress in adapting these errors to TypeScript, considering syntactic and semantic differences in relation to the original classification language. This work seeks to provide empirical evidence to guide technological choices in developing artificial intelligence-based educational tools for programming education, contributing to better integration between human knowledge and artificial intelligence potentialities in the educational context.
Keywords: large language models; programming education; automated error diagnosis; typescript; artificial intelligence in education.

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização do Projeto

O ensino de programação representa um dos pilares na formação dos cursos de Computação. No contexto das universidades brasileiras, as disciplinas de fundamentos de programação apresentam elevadas taxas de reprovação e consequente evasão. Nesse sentido, pesquisas indicam que as dificuldades no processo de aprendizagem estão associadas, principalmente, ao "alto grau de abstração, além do tempo e esforço exigidos pela disciplina" (ARIMOTO; OLIVEIRA, 2019).

As dificuldades específicas em disciplinas de programação constituem um fator relevante que pode ser mitigado por meio de ferramentas de apoio educacional (ALVIM; BITTENCOURT; DURAN, 2024). Esse cenário evidencia a necessidade de soluções tecnológicas que facilitem o processo de aprendizagem e reduzam as barreiras iniciais no aprendizado de programação.

A análise automatizada de código representa um tema de crescente interesse na Educação em Computação, especialmente com a disseminação de tecnologias baseadas em Inteligência Artificial (IA). Modelos de linguagem ampla (LLMs)[1] têm apresentado capacidades promissoras na análise de código, trazendo novas possibilidades para o desenvolvimento de ferramentas educacionais relevantes (LEINONEN *et al.*, 2023a).

No ambiente educacional, professores enfrentam o desafio de fornecer *feedback* individualizado e de qualidade para turmas numerosas. Tal limitação compromete o processo de aprendizagem, uma vez que erros mais relevantes educacionalmente permanecem sem diagnóstico e tratamento adequados, conforme evidenciado em estudos sobre identificação automatizada desses erros (WATSON; LI; GODWIN, 2025).

1.2 Definição do Problema

O problema central abordado encontra lugar na ausência de estudos comparativos sobre a capacidade de resposta de LLMs no diagnóstico automático de erros em programação. Até então, as pesquisas disponíveis concentram-se em LLMs individuais, não trazendo uma análise comparativa que permita identificar qual apresenta melhor desempenho nesse contexto de diagnóstico de erros (SUN *et al.*, 2024; RAIHAN *et al.*, 2025; PRATHER *et al.*, 2023a). Existe potencial de pesquisa em um comparativo que considere fatores como assertividade na identificação de erros, qualidade do *feedback* fornecido (em português), bem como custos operacionais dos LLMs no diagnóstico de erros em fundamentos de programação.

[1] Do inglês, *Large Language Models* (LLMs).

1.3 Relevância do Problema

A relevância da pesquisa é justificada pelos avanços surpreendentes da área de IA nos últimos anos. Faz-se necessário entender as capacidades de diagnóstico dos LLMs (ainda que em constante evolução), para que sejam propostas soluções educacionais factíveis, que integrem harmônica e eficientemente o conhecimento humano (professores) às ferramentas computacionais (IA). Além disso, uma pesquisa recente da Associação Brasileira de Mantenedoras do Ensino Superior (2024) constatou que 71% dos universitários brasileiros usam IA frequentemente nos estudos, sugerindo receptividade e crescente demanda por ferramentas educacionais baseadas nessas tecnologias.

1.4 Justificativa

A escolha do tema é justificada na convergência de fatores tecnológicos, educacionais e sociais que tornam o momento propício para compreender as potencialidades e limitações dos LLMs. O avanço significativo de cada um dos diversos LLMs (e.g., GPT, DeepSeek e Claude), bem como a concentração de estudos que exploram apenas características individuais de cada modelo (SUN *et al.*, 2024; RAIHAN *et al.*, 2025; PRATHER *et al.*, 2023a), sugerem a relevância de análises comparativas que considerem capacidades específicas – como o diagnóstico de erros em programação. Do ponto de vista das políticas públicas nacionais, a pesquisa alinha-se com as diretrizes do Plano Brasileiro de Inteligência Artificial 2024-2028, que estabelece como prioritária a “integração de soluções de IA em ambientes educacionais para personalização do aprendizado e melhoria dos resultados educacionais” (BRASIL. Ministério da Educação, 2024).

Justifica-se a motivação pessoal pela oportunidade de aprofundamento em um tema atual e não contemplado pela matriz curricular do curso. Além disso, foi valorizada a chance de realizar pesquisa ainda durante a graduação. Entende-se que há benefício tanto técnico (para o mercado de trabalho) quanto científico (considerando a possibilidade de pós-graduação).

1.5 Desafios do Projeto

O primeiro desafio reside em definir um subconjunto representativo de erros comuns em fundamentos de programação, na linguagem TypeScript, a serem diagnosticados. Esses erros precisam ir além do escopo léxico e sintático, mas se entende que nem todo erro do subconjunto deva ser semântico. Existem, inclusive, questões terminológicas sobre como designar esses erros. O termo, em inglês, *misconception* tem se mostrado uma alternativa, no sentido de “equívoco conceitual” (SILVA, 2024). Toda essa distinção será oportunamente apresentada na evolução da pesquisa.

Desafios relacionados à avaliação incluem o estabelecimento de métricas comparativas, bem como a configuração dos LLMs considerados, que possuem diferentes interfaces e limitações. Nisso, será necessário estudar sobre engenharia de *prompts* e assumir que as respostas podem apresentar componentes probabilísticos (em que, ao se fazer duas vezes uma mesma pergunta, nem sempre se tem a mesma resposta).

Outro desafio reside no próprio processo de formalização e escrita científica, que não é abordado nos quatro primeiros semestres do curso. Também há preocupação em adequar o projeto à correspondência de carga-horária das disciplinas de TCC1 e TCC2, visto que o acadêmico trabalha.

1.6 Contribuição

A contribuição da pesquisa se dá por estabelecer um comparativo entre as LLMs, considerando a capacidade de diagnóstico automático de erros em programação, em um momento no qual estudos se concentram nas características individuais de cada modelo. Embora a pesquisa seja realizada em nível de graduação, acredita-se que o comparativo possa auxiliar em escolhas no desenvolvimento de ferramentas educacionais para o ensino de programação, como também em entender sobre quais aspectos a IA pode contribuir na avaliação de códigos feita por professores (humanos).

1.7 Objetivos

1.7.1 Objetivo Geral

Estabelecer um comparativo, sobre a capacidade de resposta, de três modelos de linguagem ampla no diagnóstico automático de um subconjunto de erros em fundamentos de programação, considerando a linguagem TypeScript.

1.7.2 Objetivos Específicos

No momento da proposta, consideram-se os seguintes objetivos específicos que serão novamente ponderados para o projeto:

1. Selecionar e configurar três LLMs para uma análise comparativa no escopo da pesquisa proposta;
2. Definir, a partir da literatura, um subconjunto representativo de erros frequentes em fundamentos de programação;
3. Adaptar o subconjunto de erros definido para a linguagem TypeScript;

4. Compor um conjunto padronizado (*corpus*) de códigos de teste que contenha, estrategicamente, os erros identificados no subconjunto;
5. Avaliar o percentual de acerto de cada LLM selecionado na identificação e diagnóstico dos erros, estabelecendo métricas quantitativas de desempenho;
6. Analisar qualitativamente a natureza do *feedback* fornecido por cada LLM, considerando clareza e relevância pedagógica;
7. Correlacionar as avaliações qualitativas e quantitativas com os custos operacionais de cada modelo avaliado;
8. Implementar um protótipo de mínimo produto viável (MVP)[2] de API REST, utilizando Laravel, integrado ao modelo de melhor desempenho geral identificado.

[2] *Minimum Viable Product*.

2 FUNDAMENTAÇÃO TEÓRICA

A presente pesquisa fundamenta-se em dois temas: Modelos de Linguagem Ampla (Seção 2.1) e erros em Programação de Computadores (Seção 3.1). Ambos são discorridos na sequência.

2.1 Modelos de Linguagem Ampla (LLMs)

Esta seção aborda os Modelos de Linguagem Ampla (LLMs) diante de: histórico e definição (Seção 2.1.1), conceitos e terminologia (Seção 2.1.2), como também modelos proeminentes na atualidade (Seção 2.1.3).

2.1.1 Histórico e Definição

A evolução dos modelos de linguagem modernos teve significativo avanço a partir de 2013, quando o Word2Vec (MIKOLOV *et al.*, 2013) introduziu a representação de palavras como vetores densos que capturam relações semânticas. Esse foi um passo importante, mas o verdadeiro marco transformador veio em 2017, quando Vaswani *et al.* (2017) publicaram o artigo intitulado *Attention is All You Need*, apresentando a arquitetura Transformer. Essa arquitetura inovou o campo ao usar apenas mecanismos de atenção, dispensando redes recorrentes e permitindo o processamento paralelo eficiente. A partir disso, a evolução acelerou: em 2018 surgiram o GPT e o BERT (DEVLIN *et al.*, 2018); em 2019, o GPT-2; e, em 2020, o GPT-3 (BROWN *et al.*, 2020), que, com impressionantes 175 bilhões de parâmetros, demonstrou capacidades emergentes notáveis. A popularização massiva aconteceu em novembro de 2022, com o lançamento do ChatGPT (OPENAI, 2023), que tornou essas tecnologias acessíveis a milhões de pessoas e estabeleceu um novo paradigma de interação com a IA.

Modelos de Linguagem Ampla (LLMs, do inglês *Large Language Models*) são sistemas de inteligência artificial, baseados em redes neurais profundas, que são treinados em enormes quantidades de texto através de aprendizado autossupervisionado (ZHAO *et al.*, 2023). Segundo Brown *et al.* (2020), esses modelos são capazes de aprender em contexto (*in-context learning*), executando tarefas apenas com instruções e exemplos fornecidos no momento, sem precisar de treinamento extra. A base dessas LLMs é a arquitetura Transformer (VASWANI *et al.*, 2017), que usa mecanismos de autoatenção para processar sequências de *tokens*[1] em paralelo, capturando dependências complexas na linguagem. Diferente de sistemas tradicionais baseados em regras, as LLMs aprendem padrões linguísticos complexos ao serem expostas a bilhões de palavras durante o treinamento, desenvolvendo compreensão de sintaxe, semântica e conhecimento factual (MINAEE *et al.*, 2024). Essa capacidade de generalização permite que

[1] *Token* é a menor unidade de texto processada por um modelo de IA, podendo representar palavras completas, partes de palavras ou caracteres individuais.

realizem tarefas diversas, como geração de texto, tradução, análise de código e resolução de problemas em diferentes domínios.

2.1.2 Conceitos e Terminologia

Sendo um campo de conhecimento específico, ao se trabalhar com as LLMs, utilizam-se conceitos e termos próprios. Os principais conceitos e termos são abordados na sequência:

Engenharia de *Prompt*

A engenharia de *prompt* refere-se à prática de formular instruções, contextos e exemplos para orientar modelos de linguagem na geração de respostas adequadas a tarefas específicas (WHITE *et al.*, 2023). Conforme argumentado por Wei *et al.* (2022), a forma como as instruções são estruturadas impacta significativamente a qualidade e assertividade das respostas geradas. As principais técnicas incluem *zero-shot prompting*, na qual o modelo executa tarefas sem exemplos prévios, confiando apenas no conhecimento adquirido durante o treinamento; *few-shot prompting*, que fornece alguns exemplos demonstrativos para guiar o modelo no padrão de resposta desejado; e *chain-of-thought prompting* (WEI *et al.*, 2022), que instrui o modelo a expor seu raciocínio passo a passo antes de apresentar a resposta final, melhorando significativamente o desempenho em tarefas de raciocínio complexo.

Janela de Contexto

A janela de contexto, ou *context window*, representa a quantidade máxima de *tokens* que um modelo de linguagem pode processar simultaneamente durante uma única interação (ANTHROPIC, 2024). Historicamente, modelos como o GPT-2 eram limitados a aproximadamente 2.048 tokens, restringindo significativamente sua capacidade de processar documentos extensos ou manter contexto em longas conversas. A evolução dos modelos trouxe expansões: GPT-3 suportava até 4.096 *tokens*, GPT-4 expandiu para 8.192 *tokens* na versão base e 32.768 *tokens* na versão estendida, enquanto modelos recentes como Claude 3 da Anthropic e Gemini 1.5 do Google alcançaram janelas de contexto de 200.000 *tokens* ou mais (GOOGLE, 2023). Janelas maiores implicam maior custo computacional e latência, estabelecendo um compromisso entre capacidade analítica e eficiência operacional que deve ser considerado na implementação de sistemas práticos.

Engenharia de Contexto

Engenharia de contexto envolve técnicas para otimizar a utilização eficiente da janela de contexto disponível, maximizando a relevância das informações fornecidas ao modelo (LEWIS *et al.*, 2020). Uma abordagem proeminente é a *Retrieval-Augmented Generation* (RAG), proposta por Lewis *et al.* (2020), que combina recuperação de infor-

mações com geração de texto: antes de produzir uma resposta, o sistema busca documentos relevantes em uma base de conhecimento externa e os inclui no contexto, fundamentando as respostas em informações atualizadas e específicas do domínio. Outras técnicas incluem *chunking*, que divide documentos longos em segmentos gerenciáveis processados sequencialmente; *summarization*, que condensa informações extensas preservando o conteúdo essencial; e estratégias de priorização que ordenam informações por relevância.

Alucinação

O fenômeno de alucinação em LLMs refere-se à geração de informações falsas, incoerentes ou que não são factualmente corretas, inconsistentes com o contexto fornecido ou completamente inventadas pelo modelo (JI *et al.*, 2023). Segundo a taxonomia apresentada por Huang *et al.* (2023), alucinações podem ser classificadas em intrínsecas, quando o modelo contradiz informações presentes no contexto de entrada, e extrínsecas, quando gera informações não verificáveis ou falsas sobre o mundo real. As causas fundamentais incluem lacunas no conhecimento adquirido durante o treinamento, vieses nos dados de treinamento, e a própria natureza probabilística do processo de geração, na qual o modelo seleciona *tokens* baseado em distribuições de probabilidade que nem sempre priorizam precisão factual sobre plausibilidade linguística (LI *et al.*, 2024).

2.1.3 Modelos Proeminentes na Atualidade

O cenário atual de LLMs é caracterizado por diversos modelos desenvolvidos por diferentes organizações, cada um com características técnicas distintivas, filosofias de desenvolvimento particulares e focos de aplicação específicos. Essa diversidade reflete tanto a intensa competição no campo quanto diferentes abordagens para resolver os desafios técnicos e éticos associados a esses sistemas.

ChatGPT

O ChatGPT, desenvolvido pela OpenAI e lançado em novembro de 2022, foi o precursor da popularização das LLMs, alcançando 100 milhões de usuários em apenas dois meses (OPENAI, 2023). Baseado inicialmente no GPT-3.5 e evoluindo para GPT-4, o modelo utiliza *Reinforcement Learning from Human Feedback* (RLHF) para melhorar suas respostas. O modelo destaca-se por sua versatilidade em tarefas diversas, desde programação e análise de dados até geração de conteúdo criativo e assistência em pesquisa acadêmica, consolidando-se como referência em aplicações conversacionais de IA.

DeepSeek

O DeepSeek representa o primeiro grande concorrente chinês no mercado de LLMs, desenvolvido pela *startup* DeepSeek AI em 2023 como resposta aos modelos ocidentais. O DeepSeek-Coder, especializado em programação, apresenta desempenho competitivo em *benchmarks* como HumanEval (GUO *et al.*, 2024). Seus diferenciais incluem arquitetura otimizada para eficiência computacional, reduzindo custos, e técnicas de *fill-in-the-middle* para completação de código em diferentes contextos.

Claude

Claude, desenvolvido pela Anthropic em 2021, diferencia-se pelo foco em segurança através da metodologia *Constitutional AI*, que treina o modelo para seguir princípios éticos explícitos (ANTHROPIC, 2024). O Claude 3 possui três versões (Haiku, Sonnet e Opus) e oferece janela de contexto de 200.000 *tokens*, uma das maiores disponíveis.

Gemini

Gemini, desenvolvido pelo Google DeepMind e lançado em dezembro de 2023, foi projetado como modelo multimodal nativo, processando texto, imagens, áudio e vídeo através de uma arquitetura unificada (GOOGLE, 2023). O Gemini possui três versões: Nano (para dispositivos móveis), Pro (uso geral) e Ultra (mais poderosa).

2.2 Erros em Programação

De acordo com Silva, Caceffo e Azevedo (2023), o ato de programar está sujeito a diferentes tipos de erros. Erros de sintaxe violam as regras gramaticais da linguagem e são detectados pelo compilador, como parênteses não balanceados ou palavras-reservadas incorretas. Erros de lógica ocorrem quando o código executa mas produz resultados incorretos, como usar operador de comparação errado em laços. Erros de *runtime* aparecem durante a execução, como divisão por zero ou acesso a índices inválidos. Erros semânticos acontecem quando o código funciona mas não atende às expectativas do domínio, como calcular média aritmética ao invés de ponderada. Nesse sentido, Altadmri e Brown (2015) afirma que erros sintáticos são mais frequentes, porém erros lógicos consomem mais tempo de depuração.

O diagnóstico de erros é desafiador, especialmente para iniciantes. Mensagens de compiladores são frequentemente ininteligíveis (BECKER, 2016) e iniciantes têm dificuldade em conectar os efeitos com as causas no código. O tempo de depuração consome entre 35% e 50% do tempo de programadores profissionais (MCCAULEY *et al.*, 2008), proporção que aumenta para iniciantes. Funcionalidades de depuração, como o estabelecimento de *breakpoints*, exigem saber onde investigar – o que pressupõe hipóteses que iniciantes não conseguem formular.

LLMs têm surgido como ferramentas promissoras para auxiliar no diagnóstico de erros em programção. Leinonen *et al.* (2023b) mostraram que GPT-3.5 e GPT-4 geram explicações mais compreensíveis do que as de compiladores, traduzindo e acessibilizando termos técnicos.

Finnie-Ansley *et al.* (2022) encontraram taxas de sucesso acima de 50% em problemas introdutórios. Porém, limitações persistem: alucinações podem sugerir elementos (como funções e métodos) inexistentes ou sintaxe inválida (PRATHER *et al.*, 2023b), a falta de contexto completo dificulta diagnósticos precisos, e há riscos de que uma dependência excessiva prejudique o aprendizado (dívida cognitiva).

3 TRABALHOS RELACIONADOS

Em se tratando do processo de ensino-aprendizagem em programação, o presente capítulo primeiramente aborda os erros na aprendizagem (Seção 3.1) e, depois, o uso de LLMs no ensino (Seção 3.2). Essas pesquisas foram cruciais para trazer uma compreensão mais aprofundada sobre o tema. Por fim, faz-se uma comparação com a presente pesquisa (Seção 3.3).

3.1 Erros no Processo de Aprendizagem de Programação

O trabalho de Castro e Tedesco (2020) discute como os erros fazem parte do processo de aprendizagem em programação. Foi feito um experimento em que os alunos precisavam refletir sobre os erros que cometiam e, para isso, usavam portfólios. O estudo mostrou que os alunos, ao pararem para pensar sobre os erros cometidos, conseguiam aprender melhor e diminuíam a sensação de incapacidade diante de erros futuros. Tal constatação é relevante, pois muitos abandonam o estudo de programação justamente por se frustrarem ao não conseguirem progredir diante dessas situações.

Outro trabalho interessante é o de Gomes *et al.* (2015), que fizeram uma pesquisa para identificar quais são os erros mais comuns que iniciantes cometem. Foi criado um banco de dados com os erros de compilação que os alunos tiveram durante as aulas práticas de programação em C. Com isso, conseguiram fazer uma lista dos erros mais frequentes, o que ajudou os professores a entenderem em quais pontos os alunos tinham mais dificuldades.

Kutzke e Direne (2016) desenvolveram uma ferramenta chamada FARMA-ALG que ajuda no ensino de algoritmos. A ideia foi usar os erros como uma oportunidade de aprendizagem, ao invés de só acusar que o aluno errou. A ferramenta analisa os erros e traz dicas personalizadas para cada estudante. Com isso, torna-se o aprendizado mais interessante.

3.2 LLMs no Ensino de Programação

Com o surgimento do ChatGPT e outras LLMs, vários pesquisadores começaram a estudar como essas ferramentas podem ajudar (ou atrapalhar) no ensino de programação. Trata-se de um tópico recente de pesquisa e isso pode ser constatado pelo ano das referências.

O artigo de Rosa *et al.* (2025) discute justamente no sentido do impasse mencionado: as LLMs são boas ou ruins para quem está aprendendo a programar? Foi feita uma revisão da literatura e percebido que existem pontos positivos (como a facilidade de obter explicações e correções rápidas), mas também pontos negativos (como o risco dos alunos ficarem muito dependentes, apenas consultando ferramentas, e não aprenderem de verdade).

Filho, Souza e Paula (2023) analisaram especificamente as respostas do ChatGPT para conteúdos de programação para iniciantes. Foram testadas várias perguntas básicas de programação e avaliado se as respostas estavam corretas e fáceis de entender. Os resultados mostraram que, em alguns casos, o ChatGPT acertava; mas, em outros casos, fornecia respostas confusas ou até erradas. Essa instabilidade pode prejudicar quem está começando.

Yepes e Fiorin (2023) usaram o ChatGPT como assistente de ensino na disciplina de Estruturas de Dados. A experiência foi positiva, porque os alunos conseguiram sanar dúvidas rapidamente e o professor teve mais tempo para focar em questões mais importantes. Entretanto, os autores também alertaram para o perigo dos alunos só copiarem o código sem entender.

Como estudo mais recente, tem-se a pesquisa de Pimentel *et al.* (2025), recém publicada no Congresso Brasileiro de Informática na Educação (CBIE 2025), que ocorreu entre 24 e 28 de novembro de 2025 em Curitiba, Paraná. O artigo descreve uma uma abordagem com LLMs com o propósito de gerar *feedback* mais detalhado e pedagógico diante das soluções fornecidas pelos alunos. O *feedback* foi avaliado em termos de cinco critérios e considerou se: (1) a solução foi adequadamente identificada como correta ou incorreta; (2) a solução identificou ao menos um erro presente no código corretamente; (3) a solução identificou todos os erros presentes no código corretamente; (4) a solução adicionou erros que não existem; e (5) o *feedback* gerado forneceu o código da solução ao aluno. Os *prompts* foram executados nos seguintes LLMs: Llama 3.1 (8b), Mistral-neMo (12b), Zephyr (7b), DeepSeek-Coder-v2 (16b), CodeLlama (7b) – tendo o Chat GPT-4o sido usado como controle. Experimentos foram realizados e evidenciam o potencial dos LLMs no apoio ao processo de ensino-aprendizagem. Além disso, o artigo traz características e aspectos metodológicos que podem auxiliar no presente projeto.

3.3 Comparação com o Presente Trabalho

As pesquisas indicam que já se investigou a respeito de falhas em códigos e da utilização de LLMs no aprendizado. Contudo, nota-se que são raros os estudos que confrontam diferentes modelos de LLM (como ChatGPT, Claude, DeepSeek e Gemini) com o propósito específico de identificar erros em códigos de iniciantes. Grande parte das pesquisas se concentra em um único modelo ou não realiza uma análise comparativa mais aprofundada.

O presente trabalho de conclusão de curso busca resultados nessa lacuna. Serão usados códigos com erros comuns de iniciantes em programação e, na sequência, será testado o desempenho de cada um desses modelos em detectar e explicar tais erros. Assim, será possível analisar e comparar qual modelo oferece o melhor suporte, a iniciantes, nessas situações.

4 DETALHAMENTO DA SOLUÇÃO

O presente projeto concentra-se na análise comparativa sobre a capacidade de diagnóstico automatizado de erros em fundamentos de programação, na linguagem TypeScript, considerando diferentes modelos de linguagem ampla (LLMs). Pretende-se, ao final, implementar um protótipo de mínimo produto viável (MVP) que utilize o modelo com melhor desempenho identificado.

4.1 Descrição do Trabalho

Conforme estudos sobre o uso de LLMs para explicação de códigos e melhoria de mensagem de erro (LEINONEN *et al.*, 2023a), torna-se oportuna uma análise comparativa de LLMs proeminentes, que avalie a respectiva capacidade de diagnóstico automatizado de erros em fundamentos de programação. O intuito é fornecer evidências empíricas (alcançadas por meio de experimento e/ou observação) que embasem escolhas para a criação de ferramentas educacionais nesse sentido.

Para maior clareza, seguem exemplos que serão considerados na composição do subconjunto de erros em fundamentos de programação:

1. Não usar variáveis declaradas previamente;
2. Não usar nomes significativos para identificadores;
3. Fazer atribuição sem efeito;
4. Usar o operador de atribuição (=) ao invés do operador de comparação (==);
5. Retestar, em uma estrutura condicional `if-else`, condições já verificadas;
6. Usar um laço `for` somente com a expressão condicional, sem os outros parâmetros, de modo que funcione como um `while`.

A pesquisa está estruturada em duas fases principais: (1) avaliação comparativa de três LLMs usando um conjunto padronizado de testes; e (2) implementação de um protótipo de MVP funcional de uma API REST, usando o LLM com melhor desempenho.

Haverá esforço para que a abordagem metodológica (definição do subconjunto de erros, composição do conjunto de teste, avaliação qualitativa e quantitativa dos modelos, bem como a comparação e correlação de custo operacional) aproxime-se da ideia de um *benchmark* que possa ser replicável e também atualizável. Isso é importante em função de novos LLMs que possam surgir, como também de novas versões daqueles que serão considerados na pesquisa.

4.2 Público-alvo

Como público-alvo direto, primeiramente, tem-se os pesquisadores e desenvolvedores de ferramentas educacionais, interessados em implementar soluções baseadas em LLMs para o ensino de programação. Adicionalmente, professores de disciplinas de fundamentos em programação podem se beneficiar ao melhor entender o diagnóstico de erros por LLMs.

De maneira indireta, estudantes de programação podem ter benefício diante dos dois aspectos recém-mencionados. Primeiro, pela implementação de ferramentas educacionais que proporcionem *feedback* mais assertivo e pedagogicamente alinhado às necessidades apresentadas. Depois, porque, assim como os estudantes já usam LLMs para aprender idiomas (conversação), nada impede que também utilizem de maneira parecida com o uso dos professores de programação, a fim de que tenham diagnóstico e detalhamento de erros.

4.3 Resultados Esperados

Os principais resultados pretendidos pela pesquisa são:

- **Comparação quantitativa:** avaliando-se o percentual de acerto dos LLMs, dentre outras métricas de comparação estabelecidas;
- **Comparação qualitativa:** mediante análise da clareza e da relevância pedagógica do *feedback* fornecido por cada LLM;
- **Comparação de custos operacionais:** considerando-se os dois comparativos anteriores e detalhando-se os custos por análise de cada LLM, incluindo também projeções de custos para diferentes volumes de uso;
- **Protótipo de MVP funcional**: MVP de uma API REST implementada com o LLM de melhor desempenho geral, com código-fonte disponível para adaptação e extensão, tendo o intuito de mostrar a viabilidade de ferramentas no escopo abrangido.

5 MATERIAIS E MÉTODOS

Na sequência, são apresentados os materiais (Seção 5.1), que correspondem aos recursos tecnológicos utilizados, e os passos metodológicos previstos para o desenvolvimento da pesquisa (Seção 5.2).

5.1 Materiais

Os principais recursos tecnológicos que se pretende utilizar na pesquisa são detalhados a seguir, em termos das características pertinentes ao projeto:

Laravel Framework

Possui recursos nativos para desenvolvimento de APIs REST, ORM Eloquent para interação eficiente com MariaDB, sistema robusto de validação e tratamento de erros, e facilidades integradas para autenticação, *cache* e *logging* (Laravel Team, 2025).

MariaDB

Oferece performance para operações de leitura intensiva, compatibilidade com MySQL e suporte nativo no Laravel, recursos avançados de indexação, transações ACID para integridade de dados, facilidade de *backup* e recuperação, bem como custos reduzidos de licenciamento (MariaDB Foundation, 2025).

GPT-4 (OpenAI)

Apresenta amplo conhecimento geral e capacidades de raciocínio complexo, excelente compreensão contextual e geração de explicações detalhadas, suporte robusto para múltiplas linguagens de programação (incluindo TypeScript), e larga adoção no mercado com documentação extensiva (Open AI, 2025).

DeepSeek

Destaca-se pela especialização em análise e geração de código, arquitetura otimizada para compreensão de estruturas de programação, custo operacional competitivo para aplicações em larga escala e crescente adoção em ferramentas de desenvolvimento (DeepSeek, 2025).

Claude (Anthropic)

Fornece uma abordagem baseada em Constitutional AI, promovendo respostas mais seguras e educativas (ANTHROPIC, 2022), desempenho superior em *benchmarks* de programação como SWE-bench (ANTHROPIC, 2025), como também capacidade de fornecer *feedback* estruturado e pedagogicamente orientado.

Gemini (Google)

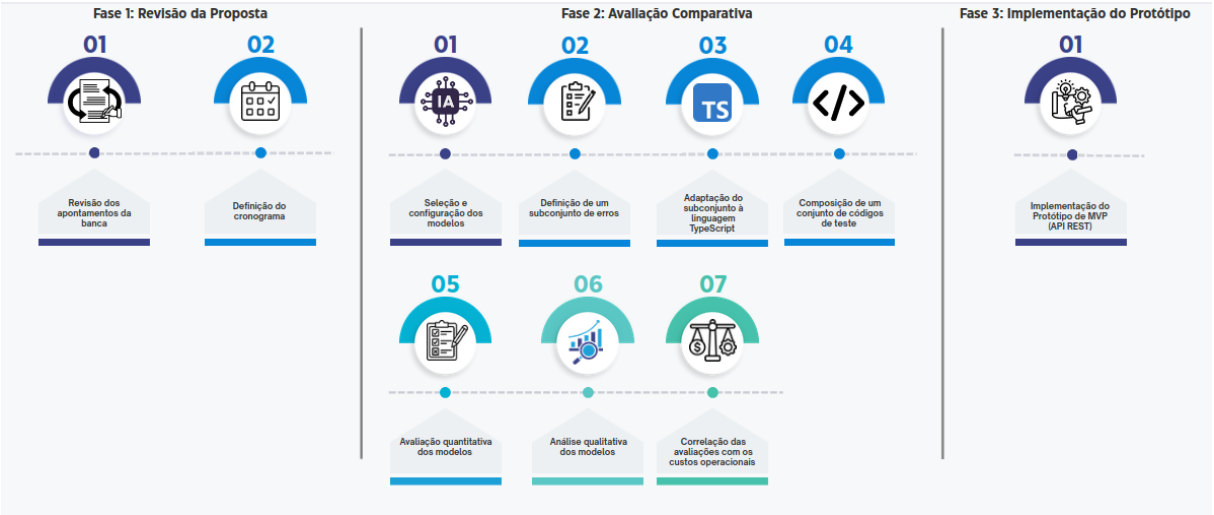
Dispõe de uma arquitetura multimodal nativa que permite análise integrada de código

e documentação, desempenho competitivo em tarefas de raciocínio e programação, integração otimizada com ecossistema Google facilitando *deployment* em ambientes educacionais, além da capacidade avançada de processamento de contextos extensos para análise de projetos completos (TEAM, 2023).

5.2 Passos Metodológicos

Foram previstos os seguintes passos metodológicos para o desenvolvimento da pesquisa, divididos em três fases, sendo a primeira preliminar: (1) revisão da proposta; (2) avaliação comparativa e (3) implementação do protótipo de MVP. Serão posteriormente consideradas, no cronograma (Capítulo 6), as atividades de elaboração e defesa tanto do projeto quanto do Trabalho de Conclusão de Curso propriamente dito. As três fases, bem como os passos relacionados, são detalhados na sequência e também pela Figura 1.

Figura 1 – Fases previstas para o desenvolvimento da pesquisa



Fonte: Autoria própria.

5.2.1 Fase 1: Revisão da Proposta (Concluída)

Passo 1: Revisão dos apontamentos da banca (Concluído)

A proposta foi adequada levando-se em consideração o que foi observado e sugerido pela banca;

Passo 2: Definição de um cronograma (Concluído)

Na sequência, foi definido um cronograma com atividades, entregas fracionadas e reuniões periódicas, até a conclusão da pesquisa. Esse cronograma será revisado no início da disciplina de Trabalho de Conclusão de Curso 2.

5.2.2 Fase 2: Avaliação Comparativa (Em Andamento)

Passo 1: Seleção e configuração dos modelos (Concluído)

Foram selecionados três LLMs para a análise comparativa. Consideraram-se: GPT, DeepSeek, Claude e Gemini. Depois disso, cada modelo foi configurado para receber o subconjunto de erros e o conjunto de códigos de teste;

Passo 2: Definição do subconjunto de erros (Concluído)

A partir da literatura técnica da área, foi definido um subconjunto representativo de erros frequentes em fundamentos de programação. Mediante critérios estabelecidos, sabe-se que alguns erros precisarão ser desconsiderados para os testes. Pretende-se documentar e categorizar esses erros;

Passo 3: Adaptação do subconjunto de erros à linguagem TypeScript (Em andamento)

O subconjunto de erros, então documentado e categorizado, precisará ser adaptado às especifidades da linguagem TypeScript, a fim de fornecer informações relevantes para que o modelo proceda com o diagnóstico automático;

Passo 4: Composição de um conjunto padronizado (*corpus*) de códigos de teste

Esses códigos devem conter erros específicos de forma controlada, mantendo realismo pedagógico e representatividade de erros reais. Entende-se que deverá haver distribuição equilibrada dos erros por entre os códigos, como também repetição (mais de uma ocorrência de um mesmo erro no conjunto de testes);

Passo 5: Avaliação quantitativa dos modelos

Diante do conjunto de códigos de teste, será avaliado o percentual de acerto de cada modelo selecionado. Serão consideradas métricas quantitativas adicionais de desempenho;

Passo 6: Análise qualitativa dos modelos

Atendo-se ao mesmo conjunto de código de testes, será analisada qualitativamente a natureza do *feedback* fornecido por cada modelo, em relação tanto à clareza quanto à relevância pedagógica;

Passo 7: Correlação das avaliações com os custos operacionais dos modelos

Dados os comparativos dos passos 5 e 6, pretende-se detalhar os custos por análise de cada modelo, estendendo isso a projeções de custos para diferentes volumes de uso.

5.2.3 Fase 3: Implementação do Protótipo de MVP

Está sendo estudada a viabilidade, em termos de cronograma, para que a pesquisa avance até uma terceira fase. A intenção é validar o comparativo por meio da implementação de um protótipo de MVP de API REST que integre o modelo de melhor desempenho geral identificado. Deseja-se chegar a um protótipo funcional que realize o diagnóstico de erros de fundamentos de programação em TypeScript. O protótipo deve fornecer uma resposta estruturada diante de um código submetido, de maneira que mostre, na prática, como um LLM pode ser usado em uma ferramenta desse sentido.

A previsão é que o protótipo seja desenvolvido com as seguintes tecnologias: Laravel 10; banco de dados MariaDB; e com serviços fornecidos por rotas HTTP (*endpoints*) RESTful padronizadas. Contudo, tratam-se de ideias bastante preliminares, que serão amadurecidas ao longo do desenvolvimento do projeto.

6 CRONOGRAMA

O desenvolvimento deste trabalho de conclusão de curso está planejado para ser executado ao longo de 12 meses, compreendendo o período de agosto de 2025 a julho de 2026. O cronograma foi estruturado de forma a distribuir as atividades de forma sequencial e paralela desde a revisão dos apontamentos da banca, referentes à proposta, até a defesa final.

As atividades foram organizadas considerando a interdependência entre as tarefas, priorizando inicialmente a revisão bibliográfica e o planejamento metodológico, seguidos pela implementação do protótipo e pelos experimentos comparativos entre os modelos de LLM. A fase final concentra-se na análise dos resultados, elaboração da documentação e preparação para a defesa.

A Tabela 1 apresenta a distribuição, ao longo dos meses, das atividades planejadas, com a estimativa de horas dedicadas a cada uma delas. O cronograma estima 120 horas de trabalho, distribuídas entre as atividades para equilibrar as demandas acadêmicas e permitir ajustes, conforme necessário, durante o desenvolvimento do projeto.

Tabela 1 – Cronograma de atividades do TCC

Atividades	**2025**					**2026**							**Horas**
	08	**09**	**10**	**11**	**12**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	
Revisão dos apontamentos da banca		X											8
Definição do cronograma	X	X											2
Escrita do capítulo de fundamentação teórica		X	X										5
Escrita do capítulo de trabalhos relacionados		X	X										5
Seleção e configuração dos modelos			X										8
Definição de um subconjunto de erros			X	X									12
Escrita do capítulo de resultados parciais			X	X									2
Adaptação do subconjunto à linguagem TypeScript			X	X	X								4
Composição de um conjunto de códigos de teste				X	X								8
Avaliação quantitativa dos modelos					X	X							9
Avaliação qualitativa dos modelos						X							6
Correlação das avaliações com os custos operacionais						X	X						10
Implementação do Protótipo de MVP (API REST)							X	X					18
Escrita do capítulo de resultado								X	X				6
Elaboração das conclusões									X				4
Revisão e formatação final do TCC									X	X			6
Elaboração da apresentação										X			3
Preparação e ensaio da defesa											X		3
Defesa do TCC												X	1
TOTAL DE HORAS													**120**

7 RESULTADOS PARCIAIS

Este capítulo apresenta os resultados parciais, alcançados até o presente, no desenvolvimento da pesquisa. São descritos os avanços realizados em cada etapa da Fase 2, conforme a metodologia proposta no Capítulo 5. Inicialmente, na Seção 7.1, são apresentados os critérios e procedimentos adotados para a seleção e configuração dos três modelos de linguagem escolhidos. Na Seção 7.2, é detalhado o processo de definição do subconjunto de erros de programação baseado na classificação PC3, incluindo a análise de gravidade que orientou a seleção. Por fim, na Seção 7.3, são discutidos os desafios e soluções encontrados na adaptação desse subconjunto para as especificidades da linguagem TypeScript, considerando diferenças (principalmente sintáticas e semânticas) em relação à linguagem Python, na qual a classificação original foi baseada.

7.1 Seleção e Configuração dos Modelos

A seleção e configuração dos modelos de linguagem ampla constituiu uma etapa fundamental para o desenvolvimento desta pesquisa. As três subetapas seguintes descrevem como isso foi feito.

7.1.1 Procedimentos de Integração via API

O processo de integração com os modelos selecionados foi realizado por meio de suas respectivas APIs comerciais, garantindo acesso às versões mais recentes e estáveis de cada modelo. Para cada plataforma, foram realizadas assinaturas dos serviços de API, possibilitando que a codificação utilizasse os modelos dentro dos limites estabelecidos pelos planos contratados.

A integração com o Claude foi realizada utilizando a ferramenta oficial de geração de *prompts* fornecida pela Anthropic, que permite a estruturação sistemática de instruções e que a comunicação com o modelo seja otimizada. Essa ferramenta facilita a criação de *prompts* bem formatados e consistentes, aspecto fundamental para garantir a reprodutibilidade dos experimentos.

De forma análoga, a integração com o GPT-4 seguiu as diretrizes estabelecidas pela OpenAI, utilizando a API oficial e ferramentas de suporte para estruturação de *prompts*. A configuração incluiu a definição de parâmetros como temperatura[1], *tokens* máximos de resposta e outras configurações específicas do modelo que influenciam a consistência e a qualidade das respostas geradas.

[1] Parâmetro que controla o nível de aleatoriedade nas respostas geradas pelo modelo, variando tipicamente entre 0 (previsível e consistente) e 1 (mais criativo).

A integração com o Gemini foi realizada por meio da API do Google Cloud, aproveitando a infraestrutura de serviços da plataforma e suas ferramentas de gerenciamento de requisições. A configuração considerou as particularidades da arquitetura multimodal do modelo, embora nesta fase inicial o foco tenha se mantido na análise de código textual.

7.1.2 Testes de Requisição e Validação de *Payload*

Após a configuração inicial das APIs, foram conduzidos testes sistemáticos de requisição e recebimento de *payload*[2] para validar a comunicação com cada modelo. Esses testes tiveram como objetivo verificar a correta formatação das requisições, a consistência das respostas e a adequação dos formatos de dados trocados entre o sistema e as APIs.

Os testes incluíram a submissão de trechos de código TypeScript simples, acompanhados de instruções básicas de análise, permitindo verificar a capacidade de cada modelo em interpretar corretamente a linguagem e gerar respostas estruturadas. A subetapa foi essencial para identificar possíveis limitações técnicas, ajustar parâmetros de configuração e estabelecer padrões de comunicação que seriam posteriormente utilizados nos experimentos principais.

Durante esta subetapa, foram observadas diferenças na estrutura de resposta de cada modelo, na velocidade de processamento e na forma como cada API gerencia contextos e histórico de conversação. Tais observações orientaram ajustes na implementação específica para cada plataforma, garantindo que as diferenças técnicas não comprometessem a comparabilidade dos resultados.

7.1.3 Engenharia de *Prompt* com Exemplos TypeScript

Com as integrações validadas, iniciou-se o processo de engenharia de *prompt*, fundamental para direcionar os modelos especificamente para a tarefa de detecção de erros em códigos de iniciantes. Essa subetapa baseou-se em técnicas estabelecidas de engenharia de *prompt*, incluindo o fornecimento de exemplos concretos em TypeScript que ilustrassem tanto códigos corretos quanto códigos contendo problemas específicos.

A estratégia de *few-shot learning*[3] foi empregada, fornecendo aos modelos exemplos representativos de cada tipo de problema que se deseja detectar. Os exemplos foram cuidadosamente selecionados para cobrir diferentes níveis de complexidade e variações sintáticas em TypeScript, incluindo casos que exploravam características específicas da linguagem como tipagem estática, interfaces e recursos de ES6+[4].

[2] Conjunto de dados transmitidos no corpo de uma requisição ou resposta HTTP.

[3] Técnica que consiste em fornecer ao modelo alguns exemplos da tarefa desejada diretamente no *prompt*, permitindo que ele aprenda o padrão esperado sem necessidade de treinamento adicional.

[4] Ou ECMAScript 6+, é um padrão para linguagens de *script* como o TypeScript e o JavaScript.

Cada exemplo foi acompanhado de anotações explicativas que detalhavam o problema identificado, a gravidade e as razões pelas quais o código, embora funcionalmente correto, apresentava questões de compreensão conceitual. O opção por uma abordagem mais pedagógica nos *prompts* visou alinhar as respostas dos modelos com os objetivos educacionais da pesquisa.

7.2 Definição de um Subconjunto de Erros

A definição do subconjunto de erros baseou-se na classificação PC3 proposta por Silva, Caceffo e Azevedo (2023), que representa o trabalho mais recente e abrangente na identificação e categorização de problemas comuns em códigos de iniciantes em programação. Os autores realizaram um estudo empírico analisando códigos corretos (que passam em todos os casos de teste) submetidos por estudantes de uma disciplina introdutória de programação, demonstrando que a aprovação nos testes não garante a qualidade do código. A pesquisa se destaca por estabelecer um subgrupo específico denominado Problemas de Compreensão em Códigos Corretos (PC3), com escopo delimitado apenas a códigos considerados corretos por mecanismos de correção.

A pesquisa mencionada fundamenta-se em uma análise comparativa sistemática com estudos anteriores relevantes na área. Os autores comparam seus resultados com:

1. a documentação de antipadrões de Gama *et al.* (2018), que identificou 28 hipóteses de problemas de compreensão em Python por meio da análise de frequência em avaliações;

2. os estudos sobre análise de complexidade de código de Ihantola e Petersen (2019), que investigaram a complexidade em cursos introdutórios;

3. as pesquisas sobre estilo semântico de Ruvo *et al.* (2018), que identificaram 16 indicadores de estilos semânticos relacionados a comandos condicionais e ao uso de variáveis; e

4. os trabalhos sobre detecção automatizada de antipadrões de Ureel e Wallace (2019), que desenvolveram o WebTA para crítica de código com aproximadamente 200 regras.

A comparação permitiu que os autores identificassem lacunas nas classificações existentes e propusessem uma taxonomia mais completa. São 45 erros (chamados de PC3, segundo os autores) divididos em 8 categorias. A recência e a abrangência da pesquisa justifica a escolha para fundamentar a solução objetivada. A Tabela 2 apresenta o ranqueamento dos 45 erros identificados em ordem decrescente de gravidade, calculada pela diferença (DIF) en-

tre o somatório de votos de alta gravidade (CT+C)[5] e baixa gravidade (N+B)[6], ajustada pelo percentual de discordância (DT-D)[7].

Para a presente pesquisa, foram selecionados os erros com maior valor de DIF, priorizando aqueles que apresentam maior consenso quanto à gravidade e maior impacto no aprendizado de fundamentos de programação. A seleção considerou também a viabilidade de reprodução desses erros de forma controlada em códigos de teste e sua relevância para o contexto de ensino de TypeScript.

7.3 Adaptação do Subconjunto à Linguagem TypeScript

Atualmente, a pesquisa encontra-se nesta etapa, realizando o processo de adaptação do subconjunto de erros identificados por Silva, Caceffo e Azevedo (2023) em Python para a linguagem TypeScript. A adaptação envolve análise criteriosa das diferenças sintáticas e semânticas entre as duas linguagens, considerando que alguns erros se manifestam de forma equivalente, enquanto outros requerem ajustes específicos ou não são aplicáveis.

Um exemplo de erro não aplicável ao TypeScript é a indentação inadequada, que os autores consideram na classificação PC3 para Python. Como Python utiliza indentação obrigatória para delimitar blocos de código, problemas de indentação representam erros de sintaxe ou de lógica significativos. Em TypeScript, por outro lado, a indentação é um elemento meramente estético (convenção de formatação para melhor legibilidade), uma vez que a linguagem utiliza chaves para delimitar blocos, tornando esse tipo de erro menos importante para análise.

Por outro lado, TypeScript apresenta erros específicos ausentes em Python, como problemas relacionados à tipagem estática. Um exemplo é o uso inconsistente de tipos em declarações de variáveis e parâmetros de métodos (ou funções), quando iniciantes podem declarar tipos incorretos ou incompatíveis, gerando erros de compilação. Esse tipo de problema não existe em Python devido à sua natureza de tipagem dinâmica.

O trabalho de adaptação está em progresso, com documentação detalhada dos erros selecionados e os exemplos de código TypeScript correspondentes. A tabela completa com a adaptação dos erros pode ser consultada on-line em: Adaptação de Erros para TypeScript.

[5] CT = Concordo Totalmente; C = Concordo.
[6] N = Neutro; B = Baixa gravidade.
[7] DT = Discordo Totalmente; D = Discordo.

Tabela 2 – Ranqueamento dos 45 PC3 em relação à gravidade. Tabela ordenada de forma decrescente pela coluna DIF

ID	Nome	CT+C	N+B	DT-D	DIF
C8	Laço *for* com variável responsável pela iteração sendo sobrescrita	31	0	1	30
B6	Comparação Booleana feita utilizando laço *while* desnecessário	26	4	2	20
G1	Condição *while* testada novamente em seu interior	26	3	3	20
B8	Não utilização da estrutura *if-elif-else*	24	8	0	16
C2	Laço redundante ou desnecessário	24	5	3	16
C4	Laço *for* executado por vezes arbitrárias ao invés de usar laço *while*	24	5	3	16
D4	Funções acessando variáveis fora de seu escopo	24	4	4	16
G4	Variáveis/funções com nomes não significativos	24	7	1	16
H1	Declaração sem efeito	24	7	1	16
B12	Consecutivas declarações de *if*'s iguais com operações distintas em seus blocos	23	5	4	14
B9	*elif/else* retestando condição superior	23	4	5	14
E2	Uso redundante ou desnecessário de listas	23	3	6	14
A4	Redefinição de *built-in*	22	3	7	12
F2	Verificação individualizada de casos de teste abertos	22	8	2	12
G5	Organização arbitrária de declarações	22	6	4	12
C3	Operações redundantes calculadas em laço	21	9	2	10
E1	Realização desnecessária de todas combinações possíveis para uma finalidade	21	7	4	10
G3	Muitas declarações numa mesma linha	21	7	4	10
A2	Variável atribuída a si mesma	20	7	5	8
A6	Variáveis com valores arbitrários (*Magic Numbers*) para operações	20	6	6	8
A7	Manipulação arbitrária para modificar variáveis declaradas	20	7	5	8
B11	Consecutivas declarações de *if*'s distintas com a mesma operação em seus blocos	20	6	6	8
B10	*elif/else* desnecessário	19	9	4	6
B3	Expressão aritmética ao invés de comparação Booleana	19	6	7	6
B4	Comandos repetidos dentro de *if-elif-else*	19	11	2	6
D1	Declaração de *return* inconsistente	19	6	7	6
A8	Tratamento arbitrário do fim de condições de leitura de dados	18	8	6	4
B7	Variável Booleana para validação ao invés de *elif/else*	18	5	9	4
C7	Tratamento interno arbitrário para casos de contorno de um laço	17	6	9	2
C6	Múltiplos laços distintos que operam sobre o mesmo conjunto	16	9	7	0
F1	Verificação para condições de entrada não especificadas	16	9	7	0
H2	*Typecast* redundante	16	8	8	0
G6	Funções não comentadas no formato *Docstring*	14	14	4	-4
A1	Variável não utilizada	13	9	10	-6
A3	Variável iniciada sem necessidade	12	8	12	-8
B1	Comparação Booleana redundante ou simplificável	12	12	8	-8
D2	Muitas *return* numa mesma função	12	8	12	-8
B5	Aninhamento de *if* ao invés de comparação Booleana	11	12	9	-10
G2	Atribuição demasiada de expressões em variáveis	11	13	8	-10
C5	Uso de variáveis intermediárias pra controle de laço	10	11	11	-12
D3	Declaração de *return* redundante ou desnecessária	10	12	10	-12
H3	Ponta ou vírgula redundante ou desnecessário	8	8	16	-16
B2	Comparação Booleana feita em variáveis intermediárias	7	9	16	-18
G1	Comentários longos numa mesma linha	7	8	17	-18
A5	*Import* não utilizado	5	8	19	-22

8 CONSIDERAÇÕES FINAIS

O presente projeto busca estabelecer um comparativo sobre a capacidade de diagnóstico automatizado de erros frequentes em fundamentos de programação, considerando códigos em TypeScript, por três modelos de linguagem ampla. Intenciona-se estabelecer critérios qualitativos e quantitativos, correlacionando-os com o custo operacional de cada modelo. Depois, pretende-se implementar um protótipo MVP funcional que mostre a aplicabilidade prática do modelo de melhor desempenho na comparação.

A relevância da pesquisa está na necessidade de compreender as capacidades e limitações dos LLMs na Educação em Computação e, mais especificamente, no diagnóstico de erros em programação. Como principal resultado, o estudo busca fornecer evidências empíricas para orientar escolhas tecnológicas em ferramentas educacionais. A partir disso, também se procura contribuir para melhor integrar o conhecimento humano e as potencialidades da IA. Nesse sentido, o protótipo de MVP implementado, com código-fonte disponível para adaptação e extensão, será um exemplo do uso das evidências fornecidas para o desenvolvimento de ferramentas educacionais no escopo abrangido.

Os resultados parciais, diante da fase atual de desenvolvimento, correspondem à seleção e configuração dos modelos, bem como à definição de um subconjunto de erros. A pesquisa encontra-se no estágio em que está sendo adaptado o subconjunto de erros definido à linguagem TypeScript, conforme mostrado pelo cronograma (Tabela 1).

Dentre as principais limitações, citam-se: (a) foco restrito ao subconjunto de erros considerado; (b) diagnóstico de erros concentrado na linguagem TypeScript, sem generalização imediata para outras linguagens de programação; e (c) avaliação direcionada aos três LLMs específicos selecionados, considerando as versões atuais.

REFERÊNCIAS

ALTADMRI, A.; BROWN, N. C. 37 million compilations: Investigating novice programming mistakes in large-scale student data. *In*: **Proceedings of the 46th ACM technical symposium on computer science education**. [*S.l.*: *s.n.*], 2015. p. 522–527.

ALVIM, V.; BITTENCOURT, R. A.; DURAN, R. S. Evasão nos cursos de graduação em computação no brasil. *In*: **Anais do IV Simpósio Brasileiro de Educação em Computação**. Evento Online: Sociedade Brasileira de Computação, 2024. p. 1–11.

ANTHROPIC. **Constitutional AI: Harmlessness from AI Feedback**. 2022. Disponível em: https://www.anthropic.com/research/constitutional-ai-harmlessness-from-ai-feedback.

ANTHROPIC. **The Claude 3 model family: Opus, Sonnet, Haiku**. [*S.l.*], 2024.

ANTHROPIC. **Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet**. 2025. Disponível em: https://www.anthropic.com/engineering/swe-bench-sonnet.

ARIMOTO, M.; OLIVEIRA, W. Dificuldades no processo de aprendizagem de programação de computadores: um survey com estudantes de cursos da Área de computação. *In*: SOCIEDADE BRASILEIRA DE COMPUTAçãO. **Anais do XXVII Workshop sobre Educação em Computação**. Belém: Sociedade Brasileira de Computação, 2019. p. 244–254.

ASSOCIAÇÃO BRASILEIRA DE MANTENEDORAS DO ENSINO SUPERIOR. **Inteligência Artificial na Educação Superior**. Brasília, 2024. Relatório de Pesquisa em parceria com Educa Insights.

BECKER, B. A. An effective approach to enhancing compiler error messages. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 48, n. 1, p. 126–131, 2016.

BRASIL. Ministério da Educação. **Plano Brasileiro de Inteligência Artificial (PBIA) 2024-2028**. Brasília, 2024. Disponível em: https://www.gov.br/mec/pt-br/assuntos/noticias/2024/julho/mec-fara-parte-do-plano-brasileiro-de-inteligencia-artificial.

BROWN, T. *et al.* Language models are few-shot learners. **Advances in neural information processing systems**, v. 33, p. 1877–1901, 2020.

CASTRO, F.; TEDESCO, P. Promovendo a reflexão sobre o erro em disciplinas introdutórias de programação no ensino superior. **Revista Brasileira de Informática na Educação**, v. 28, p. 150–165, 2020.

DeepSeek. **DeepSeek**. 2025. Disponível em: https://www.deepseek.com/en.

DEVLIN, J. *et al.* Bert: Pre-training of deep bidirectional transformers for language understanding. **arXiv preprint arXiv:1810.04805**, 2018.

FILHO, L. C. P.; SOUZA, T. d. P. de; PAULA, L. B. de. Análise das respostas do chatgpt em relação ao conteúdo de programação para iniciantes. *In*: SBC. **Anais do XXXIV Simpósio Brasileiro de Informática na Educação**. [*S.l.*], 2023. p. 1738–1748.

FINNIE-ANSLEY, J. *et al.* The robots are coming: Exploring the implications of openai codex on introductory programming. *In*: **Australasian Computing Education Conference**. [*S.l.*: *s.n.*], 2022. p. 10–19.

GAMA, G. *et al.* **An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python**. [*S.l.*], 2018. In English, 106 pages.

GOMES, M. *et al.* Um estudo sobre erros em programação: reconhecendo as dificuldades de programadores iniciantes. *In*: **Anais dos Workshops do IV Congresso Brasileiro de Informática na Educação**. [*S.l.*: *s.n.*], 2015.

GOOGLE. **Gemini: A family of highly capable multimodal models**. [*S.l.*], 2023.

GUO, D. *et al.* Deepseek-coder: When the large language model meets programming–the rise of code intelligence. **arXiv preprint arXiv:2401.14196**, 2024.

HUANG, L. *et al.* A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. **arXiv preprint arXiv:2311.05232**, 2023.

IHANTOLA, P.; PETERSEN, A. Code complexity in introductory programming courses. *In*: **Proceedings of the 52nd Hawaii International Conference on System Sciences**. [*S.l.*: *s.n.*], 2019. p. 7662–7670.

JI, Z. *et al.* Survey of hallucination in natural language generation. **ACM Computing Surveys**, ACM New York, NY, v. 55, n. 12, p. 1–38, 2023.

KUTZKE, A.; DIRENE, A. Mediação do erro no ensino de programação de computadores: fundamentos e aplicação da ferramenta farma-alg. *In*: **Anais dos Workshops do V Congresso Brasileiro de Informática na Educação**. [*S.l.*: *s.n.*], 2016.

Laravel Team. **Laravel - The PHP Framework for Web Artisans**. 2025. Disponível em: https://laravel.com.

LEINONEN, J. *et al.* Using large language models to explain programming error messages. **Learning and Individual Differences**, Elsevier, v. 105, p. 102310, 2023.

LEINONEN, J. *et al.* Using large language models to enhance programming error messages. **arXiv preprint arXiv:2305.11520**, 2023.

LEWIS, P. *et al.* Retrieval-augmented generation for knowledge-intensive nlp tasks. **Advances in neural information processing systems**, v. 33, p. 9459–9474, 2020.

LI, J. *et al.* The dawn after the dark: An empirical study on factuality hallucination in large language models. *In*: **Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics**. [*S.l.*: *s.n.*], 2024. p. 10879–10899.

MariaDB Foundation. **MariaDB Server**. 2025. Disponível em: https://mariadb.org.

MCCAULEY, R. *et al.* Debugging: A review of the literature from an educational perspective. **Computer Science Education**, Taylor & Francis, v. 18, n. 2, p. 67–92, 2008.

MIKOLOV, T. *et al.* Efficient estimation of word representations in vector space. **arXiv preprint arXiv:1301.3781**, 2013.

MINAEE, S. *et al.* Large language models: A survey. **arXiv preprint arXiv:2402.06196**, 2024.

Open AI. **ChatGPT**. 2025. Disponível em: https://chatgpt.com/pt-BR/overview?openaicom_referred=true.

OPENAI. **GPT-4 technical report**. [*S.l.*], 2023.

PIMENTEL, E. *et al.* Abordagem com llm para geração automatizada de feedback no ensino de programação de computadores. *In*: **Anais do XXXVI Simpósio Brasileiro de Informática na**

Educação. Porto Alegre, RS, Brasil: SBC, 2025. p. 590–602. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/sbie/article/view/38453.

PRATHER, J. *et al.* The robots are here: Navigating the generative ai revolution in computing education. *In*: **Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2023. (ITiCSE-WGR '23), p. 108–159. ISBN 9798400704055. Disponível em: https://doi.org/10.1145/3623762.3633499.

PRATHER, J. *et al.* The robots are here: Navigating the generative ai revolution in computing education. **arXiv preprint arXiv:2310.00658**, 2023.

RAIHAN, N. *et al.* Large language models in computer science education: A systematic literature review. *In*: **Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1**. New York, NY, USA: Association for Computing Machinery, 2025. (SIGCSETS 2025), p. 938–944. ISBN 9798400705311. Disponível em: https://doi.org/10.1145/3641554.3701863.

ROSA, Y. S. *et al.* Reflexões sobre o uso de llms no ensino de programação. *In*: **Anais do Simpósio Brasileiro de Educação em Computação (EDUCOMP)**. [*S.l.*: *s.n.*], 2025.

RUVO, G. D. *et al.* Understanding semantic style by analysing student code. *In*: **Proceedings of the 20th Australasian Computing Education Conference**. [*S.l.*: *s.n.*], 2018. p. 73–82.

SILVA, E.; CACEFFO, R.; AZEVEDO, R. Passar nos casos de teste é suficiente? identificação e análise de problemas de compreensão em códigos corretos. *In*: **Anais do III Simpósio Brasileiro de Educação em Computação**. Porto Alegre, RS, Brasil: SBC, 2023. p. 119–129. ISSN 3086-0733. Disponível em: https://sol.sbc.org.br/index.php/educomp/article/view/23881.

SILVA, E. P. d. **Misconceptions in correct code: assisting instructors and students by shedding light on what is potentially overshadowed by automated correction**. 2024. 197 p. Dissertação (Mestrado) — Universidade Estadual de Campinas, Campinas, 2024. Dissertação (Mestrado em Ciência da Computação). Disponível em: https://www.repositorio.unicamp.br/acervo/detalhe/1408046. Acesso em: 13 ago. 2024.

SUN, D. *et al.* Would chatgpt-facilitated programming mode impact college students' programming behaviors, performances, and perceptions? an empirical study. **International Journal of Educational Technology in Higher Education**, Springer, v. 21, p. 14, 2024. Disponível em: https://doi.org/10.1186/s41239-024-00446-5.

TEAM, G. G. Gemini: A family of highly capable multimodal models. **arXiv preprint arXiv:2312.11805**, 2023. Disponível em: https://arxiv.org/abs/2312.11805.

UREEL, L. C.; WALLACE, C. Automated critique of early programming antipatterns. *In*: **Proceedings of the 50th ACM Technical Symposium on Computer Science Education**. [*S.l.*: *s.n.*], 2019. p. 738–744.

VASWANI, A. *et al.* Attention is all you need. **Advances in neural information processing systems**, v. 30, 2017.

WATSON, C.; LI, F. W.; GODWIN, J. L. Automated identification of logical errors in programs: Advancing scalable analysis of student misconceptions. **arXiv preprint arXiv:2505.10913**, 2025.

WEI, J. *et al.* Chain-of-thought prompting elicits reasoning in large language models. **Advances in Neural Information Processing Systems**, v. 35, p. 24824–24837, 2022.

WHITE, J. *et al.* A prompt pattern catalog to enhance prompt engineering with chatgpt. **arXiv preprint arXiv:2302.11382**, 2023.

YEPES, I.; FIORIN, A. Chatgpt como assistente de ensino na disciplina de estruturas de dados. **Anais do Encontro Anual de Tecnologia da Informação**, v. 12, n. 1, p. 36–36, 2023.

ZHAO, W. X. *et al.* A survey of large language models. **arXiv preprint arXiv:2303.18223**, 2023.