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

2026

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

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

2026

PEDRO OTÁVIO PROBST ZAMPIER

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

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.

Data de aprovação: 02/julho/2026

Eleandro Maschio
Doutor
Universidade Tecnológica Federal do Paraná

Andres Jessé Porfirio
Doutor
Universidade Tecnológica Federal do Paraná

Luciano Ogiboski
Doutor
Universidade Tecnológica Federal do Paraná

GUARAPUAVA
2026

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, presença constante em cada momento de alegria e em cada batalha, segurando a minha mão ao longo de toda esta caminhada. A Ele, toda a glória.

À minha família, agradeço pelo apoio e pelo amor incondicionais. O suporte recebido desde o início desta jornada foi o alicerce sobre o qual esta conquista foi construída. Sem ele, nada disso seria possível.

De modo especial, agradeço ao meu orientador, Prof. Dr. Eleandro Maschio. Minha primeira aula na universidade foi ministrada por ele e, desde aquele momento, minha admiração por sua pessoa não cessou de crescer. Sou grato por cada orientação, por cada aula e por todo o apoio oferecido ao longo desta trajetória. São pessoas como você que agregam à nossa jornada e nos auxiliam a seguir em frente.

Aos professores do Curso Superior de Tecnologia em Sistemas para Internet da UTFPR, registro minha profunda gratidão. Quando ingressei na universidade, sequer compreendia o funcionamento de uma linguagem de programação; toda a base que hoje possuo é fruto dos ensinamentos transmitidos por vocês. Saibam que, a cada meta e conquista de minha vida, eu os trarei em meu coração, pois reconheço o valor de cada ensinamento recebido e sei que, sem eles, não alcançaria os objetivos que almejo.

Agradeço à Prof.ª Dr.ª Kelly Lais Wiggers e ao Prof. Me. Dênis Lucas Silva, que, durante toda a etapa de projeto deste trabalho, estiveram presentes avaliando-o e oferecendo conselhos que foram de suma importância para o desenvolvimento da pesquisa.

Agradeço, também, ao Prof. Dr. Andres Jessé Porfirio e ao Prof. Dr. Luciano Ogiboski, que avaliaram este trabalho agregando seus conhecimentos e contribuindo para o seu aprimoramento.

Por fim, agradeço a todas as pessoas que, de alguma forma, somaram a este caminho e que não foram nominalmente mencionadas nestes parágrafos. Cada gesto de incentivo, cada palavra de apoio e cada momento compartilhado fazem parte desta conquista. A todas elas, meu sincero reconhecimento e minha gratidão.

RESUMO

Introdução: 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 produzidos por iniciantes. Modelos de linguagem ampla (LLMs) têm mostrado capacidades promissoras na análise automatizada de código, porém os estudos existentes concentram-se majoritariamente em modelos individuais, sem análises comparativas que considerem o contexto educacional e os custos operacionais envolvidos. **Objetivo:** Estabelecer um comparativo, sobre a capacidade de diagnóstico automatizado de um subconjunto de erros em fundamentos de programação, na linguagem TypeScript, por quatro modelos de linguagem ampla (GPT-4, Claude, Gemini e DeepSeek), correlacionando desempenho técnico, qualidade pedagógica do *feedback* e viabilidade econômica de cada modelo. **Metodologia:** O desenvolvimento da pesquisa foi dividido em duas fases: (1) avaliação comparativa: (a) seleção e configuração dos modelos; (b) definição do subconjunto de erros; (c) adaptação do subconjunto de erros à linguagem TypeScript; (d) composição de um conjunto padronizado (*corpus*) de códigos de teste; (e) avaliação quantitativa dos modelos; (d) análise qualitativa dos modelos; e (f) correlação das avaliações com os custos operacionais dos modelos; e (2) implementação do protótipo de MVP (API REST). **Resultados:** O Gemini apresentou o maior percentual de acertos (86,0%), menor índice de alucinações e melhores médias qualitativas gerais. GPT-4 e DeepSeek obtiveram desempenho idêntico em acertos (78,7%), enquanto o Claude registrou o menor índice (74,7%) e o maior custo médio por requisição (US$ 0,0752). O DeepSeek destacou-se pela relação custo-benefício, com custos até 34 vezes inferiores aos do Claude. A análise evidenciou ainda que assertividade técnica e qualidade pedagógica do *feedback* não são dimensões equivalentes, e que erros de natureza estilística representam desafio maior para os modelos do que erros de natureza estrutural ou lógica. Como produto da pesquisa, foi desenvolvido um protótipo MVP na forma de uma API REST em Laravel com MariaDB, que utiliza o Laravel AI SDK para integrar os quatro LLMs avaliados em um mesmo fluxo de diagnóstico automático de erros.

Palavras-chave: modelos de linguagem ampla; ensino de programação; diagnóstico automatizado de erros; TypeScript; Inteligência Artificial na Educação.

ABSTRACT

Introduction: Programming education in higher education courses faces significant challenges related to high failure and dropout rates, partially attributed to difficulties in diagnosing conceptual errors in code produced by beginners. Large language models (LLMs) have shown promising capabilities in automated code analysis; however, existing studies focus mostly on individual models, lacking comparative analyses that consider the educational context and the operational costs involved. **Objective:** To establish a comparison regarding the automated diagnosis capability of a subset of errors in programming fundamentals, in the TypeScript language, by four large language models (GPT-4, Claude, Gemini, and DeepSeek), correlating technical performance, pedagogical quality of the feedback, and the economic viability of each model. **Methodology:** The research was divided into two phases: (1) comparative evaluation: (a) selection and configuration of the models; (b) definition of the subset of errors; (c) adaptation of the subset of errors to the TypeScript language; (d) composition of a standardized set (*corpus*) of test codes; (e) quantitative evaluation of the models; (f) qualitative analysis of the models; and (g) correlation of the evaluations with the operational costs of the models; and (2) implementation of the MVP prototype (REST API). **Results:** Gemini presented the highest percentage of correct diagnoses (86.0%), the lowest hallucination rate, and the best overall qualitative averages. GPT-4 and DeepSeek achieved identical performance in correct diagnoses (78.7%), while Claude registered the lowest rate (74.7%) and the highest average cost per request (US$ 0.0752). DeepSeek stood out for its cost-benefit ratio, with costs up to 34 times lower than those of Claude. The analysis further evidenced that technical accuracy and pedagogical quality of the feedback are not equivalent dimensions, and that errors of a stylistic nature represent a greater challenge for the models than errors of a structural or logical nature. As a product of the research, an MVP prototype was developed in the form of a REST API in Laravel with MariaDB, which uses the Laravel AI SDK to integrate the four evaluated LLMs into a single automated error diagnosis flow.

Keywords: large language models; programming education; automated error diagnosis; TypeScript; Artificial Intelligence in Education.

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Siglas

ACID	*Atomicity, Consistency, Isolation and Durability* (Atomicidade, Consistência, Isolamento e Durabilidade)
API	*Application Programming Interface* (Interface de Programação de Aplicações)
B	Baixa gravidade
BERT	*Bidirectional Encoder Representations from Transformers* (Representações de Codificador Bidirecional de Transformers)
C	Concordo
CBIE	Congresso Brasileiro de Informática na Educação
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico (originalmente, Conselho Nacional de Pesquisas)
CS1	*Computer Science 1* (corresponde à disciplina introdutória de programação)
CT	Concordo Totalmente
D	Discordo
DIF	Diferença (índice de gravidade)
DT	Discordo Totalmente
GPT	*Generative Pre-trained Transformer* (Transformador Pré-treinado Generativo)
HTTP	*Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto)
IA	Inteligência Artificial
IAG	Inteligência Artificial Generativa
INPI	Instituto Nacional da Propriedade Industrial
JSON	*JavaScript Object Notation* (Notação de Objetos JavaScript)
LGPD	Lei Geral de Proteção de Dados
LLM	*Large Language Model* (Modelo de Linguagem Ampla)
LTS	*Long Term Support* (Suporte de Longo Prazo)

MVP	*Minimum Viable Product* (Produto Mínimo Viável)
N	Neutro
ORM	*Object-Relational Mapping* (Mapeamento Objeto-Relacional)
PC^3	Problemas de Compreensão em Códigos Corretos
RAG	*Retrieval-Augmented Generation* (Geração Aumentada por Recuperação)
REST	*Representational State Transfer* (Transferência de Estado Representacional)
RLHF	*Reinforcement Learning from Human Feedback* (Aprendizado por Reforço com *Feedback* Humano)
SBC	Sociedade Brasileira de Computação
SDK	*Software Development Kit* (Kit de Desenvolvimento de Software)
TCC	Trabalho de Conclusão de Curso
TS	TypeScript
UNICAMP	Universidade Estadual de Campinas
USP	Universidade de São Paulo
UTFPR	Universidade Tecnológica Federal do Paraná

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.*, 2023).

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 modelo apresenta melhor desempenho nesse contexto de diagnóstico de erros (SUN *et al.*, 2024; RAIHAN *et al.*, 2025; PRATHER *et al.*, 2023). 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 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, Claude, Gemini e DeepSeek), 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.*, 2023), 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) na formação do acadêmico proponente.

1.5 Desafios do Projeto

O primeiro desafio residiu 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 do documento.

Desafios relacionados à avaliação incluíram o estabelecimento de métricas comparativas, bem como a configuração dos LLMs considerados, que possuem diferentes interfaces e limitações. Nisso, foi 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 residiu no próprio processo de formalização e escrita científica, que não é abordado nos quatro primeiros semestres do curso. Também houve dificuldade 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 diagnóstico automatizado de um subconjunto de erros em fundamentos de programação, na linguagem TypeScript, por quatro modelos de linguagem ampla (GPT-4, Claude, Gemini e DeepSeek), correlacionando desempenho técnico, qualidade pedagógica do *feedback* e viabilidade econômica de cada modelo.

1.7.2 Objetivos Específicos

Em consonância com o objetivo geral, foram definidos os seguintes objetivos específicos:

1. Selecionar e configurar quatro 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.

1.8 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 (e.g., diagnóstico integrado ao Moodle). 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.

[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 informações

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). Os 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) afirmam 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.* (2023) 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*., 2023), a falta de contexto completo dificulta diagnósticos precisos, e há riscos de que uma dependência excessiva prejudique o aprendizado (dívida cognitiva) (KOSMYNA *et al.*, 2025).

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 foi publicado por 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.

Durante o desenvolvimento da presente pesquisa, era previsto que fosse definido um subconjunto representativo de erros frequentes em fundamentos de programação (Seção 4.2.1), com base na literatura técnica da área. Nesse sentido, encontrou-se a classificação PC^3 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. A evolução da pesquisa (SILVA; CACEFFO; AZEVEDO, 2025; SILVA, 2024; SILVA; CACEFFO; AZEVEDO, 2022), inclusive, foi premiada como melhor tese de doutorado no Concurso de Teses e Dissertações do V Simpósio Brasileiro de Educação em Computação (EduComp 2025), promovido pela Sociedade Brasileira de Computação (SBC).

Silva, Caceffo e Azevedo (2023) realizaram um estudo empírico analisando códigos corretos (que passam em todos os casos de teste) submetidos por estudantes de uma disciplina introdutória de programação, evidenciando que a aprovação nos testes não garante a qualidade do código. A pesquisa se destaca por estabelecer um subgrupo específico denominado Proble-

mas de Compreensão em Códigos Corretos (PC^3), 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 (denominados de PC^3 pelos autores) divididos em 8 categorias. A recência e a abrangência da pesquisa justifica a escolha para fundamentar a solução apresentada neste documento.

A Tabela 1 apresenta o ranqueamento dos 45 erros identificados em ordem decrescente de gravidade, calculada pela diferença (DIF) entre o somatório de opiniões de alta gravidade (CT+C)[1] e baixa gravidade (N+B)[2], ajustada pelo percentual de discordância (DT-D)[3]. As opiniões vieram de 32 docentes que ministram disciplinas introdutórias de programação (CS1). Erros com maior valor de DIF apresentam maior consenso quanto à gravidade e maior impacto no aprendizado de fundamentos de programação. No próprio estudo, os autores estabeleceram os 15 primeiros erros como aqueles considerados mais graves, destacando-os por uma linha divisória na referida tabela.

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

[1] CT = Concordo Totalmente; C = Concordo.
[2] N = Neutro; B = Baixa gravidade.
[3] DT = Discordo Totalmente; D = Discordo.

Tabela 1 – Ranqueamento dos 45 PC[3] 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
C1	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	Muitos *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 para 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

Fonte: Silva, Caceffo e Azevedo (2023).

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), que 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 pertinentes como referências à presente pesquisa.

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. São usados códigos com erros comuns de iniciantes em programação e, na sequência, testado o desempenho de cada um dos referidos modelos em detectar e explicar tais erros. Assim, foi possível analisar e comparar quais modelos oferecem melhor suporte, a iniciantes, nessas situações.

4 MATERIAIS E MÉTODOS

Na sequência, são apresentados os materiais (Seção 4.1), que correspondem aos recursos tecnológicos utilizados, e os passos metodológicos executados em cada fase do desenvolvimento da pesquisa (Seção 4.2).

4.1 Materiais

Os principais recursos tecnológicos utilizados na pesquisa são detalhados a seguir, em termos das características pertinentes ao projeto:

Laravel Framework 13

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 11.4 LTS

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).

Hoppscotch v2026.4.0

Cliente HTTP de código aberto, utilizado em sua versão *desktop*, para testar manualmente os *endpoints* da API REST durante o desenvolvimento e para disparar as chamadas aos modelos de linguagem na etapa de coleta de dados dos experimentos. Permite a organização de coleções de requisições, a parametrização de cabeçalhos e corpos, e a inspeção de respostas, contribuindo para a reprodutibilidade dos testes (HOPPSCOTCH, 2026).

GPT-4o (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 (OPENAI, 2025).

deepseek-chat (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 Sonnet 4 (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.

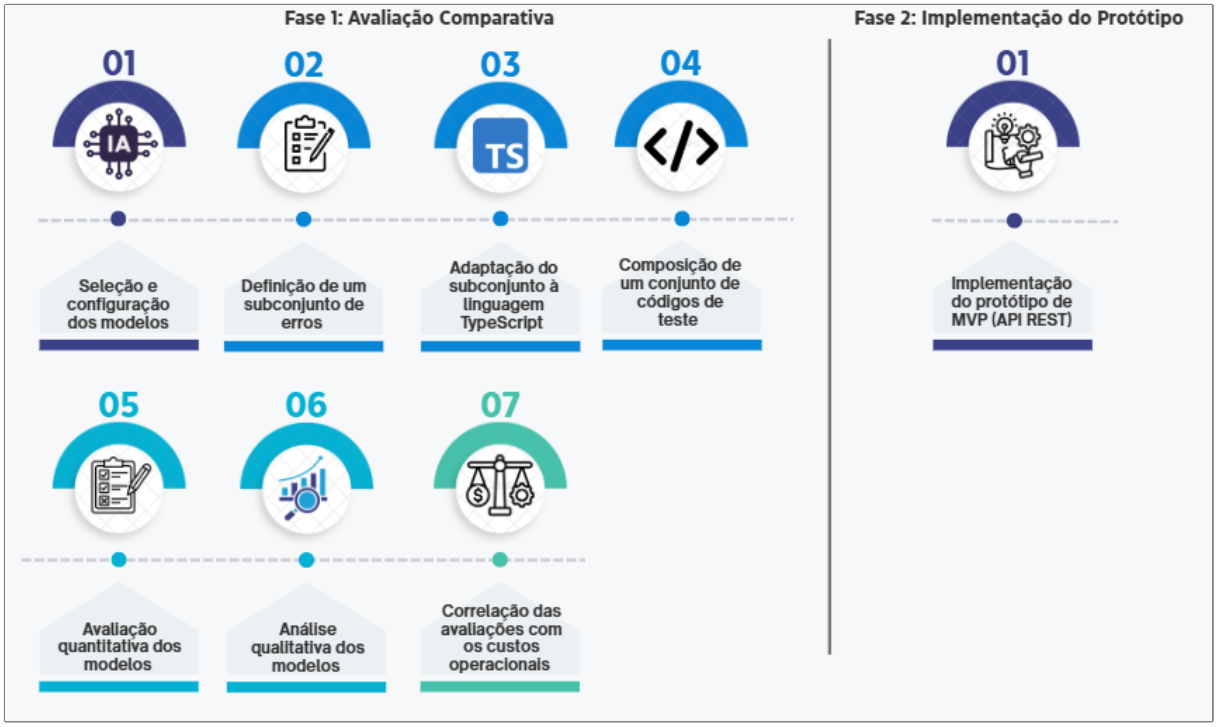
Gemini 2.0 Flash (Google)

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 (GOOGLE, 2023).

4.2 Passos Metodológicos

Foram executados os seguintes passos metodológicos para o desenvolvimento da pesquisa, divididos em duas fases: (1) avaliação comparativa e (2) implementação do protótipo de MVP. As duas fases, bem como os passos relacionados, são detalhados na sequência e também pela Figura 1.

Figura 1 – Fases executadas para o desenvolvimento da pesquisa



Fonte: Autoria própria.

4.2.1 Fase 1: Avaliação Comparativa

Passo 1: Seleção e configuração dos modelos

Foram selecionados quatro LLMs para a análise comparativa: GPT-4o (OpenAI), Claude Sonnet 4 (Anthropic), Gemini 2.0 Flash (Google) e deepseek-chat (DeepSeek). A seleção pautou-se em dois critérios principais. O primeiro foi a representatividade de mercado: priorizaram-se provedores de ampla adoção e relevância no período de realização dos experimentos (OpenAI, Anthropic, Google e DeepSeek), cujos modelos figuram entre os mais utilizados tanto em aplicações comerciais quanto em pesquisas acadêmicas, o que amplia o interesse prático dos resultados do comparativo. O segundo critério foi a diversidade de abordagens: buscou-se contemplar perfis distintos e complementares entre si: a especialização em análise e geração de código (DeepSeek), a ênfase em respostas seguras e pedagogicamente orientadas por meio de Constitutional AI (Claude), a arquitetura multimodal nativa (Gemini) e o caráter generalista de larga adoção (GPT), permitindo observar como diferentes filosofias de projeto se comportam diante de uma mesma tarefa de diagnóstico. Complementarmente, para cada provedor, optou-se pelo modelo de nível intermediário disponível no período de realização dos experimentos. Após a seleção, foram adquiridas as chaves de API de cada plataforma, necessárias para a integração e execução dos testes. Depois disso, cada modelo foi configurado para receber o subconjunto de erros e o conjunto de códigos de teste. O *prompt* criado para os testes iniciais de configuração dos modelos consta no Apêndice A;

Passo 2: Definição do subconjunto de erros

A partir da literatura técnica da área, foi definido um subconjunto representativo de erros frequentes em fundamentos de programação, baseado na classificação PC^3 (vide Seção 3.1, Tabela 1). Dentre os 45 erros categorizados, Silva, Caceffo e Azevedo (2023) destacaram os 15 primeiros da classificação como aqueles considerados mais graves e de maior impacto no aprendizado dos fundamentos de programação. Foram, então, selecionados dez erros para comporem o subconjunto considerado para o escopo desta pesquisa, a saber: B6, B8, B9, B12, C1, C3, C8, G3, G4 e H1. A seleção observou a relevância dos erros para o ensino dos fundamentos de programação em TypeScript, bem como a respectiva distribuição por entre os conteúdos (categorias) mais básicos (instruções sequenciais, estruturas condicionais e de repetição);

Passo 3: Adaptação do subconjunto de erros à linguagem TypeScript

O subconjunto de erros, originalmente abordados na linguagem Python, precisou ser adaptado às especificidades da linguagem TypeScript, a fim de fornecer informações relevantes para que o modelo procedesse com o diagnóstico automático. Foram notadas inconsistências terminológicas em Silva (2024) e, para saná-las, buscou-se a tese de doutorado (SILVA, 2024), escrita em inglês, ao invés daquele artigo em português que

a antecedia. O catálogo de erros perfaz o Apêndice B deste documento e a versão atualizada (e estendida) pode ser consultada on-line[1];

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

Esses códigos precisaram conter erros específicos de forma controlada, mantendo realismo pedagógico e representatividade de erros reais. O conjunto padronizado (*corpus*) de códigos de teste foi composto a partir de algoritmos (soluções) do livro *Introdução à Programação: 500 Algoritmos Resolvidos*[2] (LOPES; GARCIA, 2002) traduzidos para TypeScript, nos quais foram injetados erros do subconjunto adaptado no passo anterior. Para cada um dos 10 erros, foram consideradas 5 ocorrências desse erro, totalizando 50 ocorrências por modelo (50 x 4 modelos = 200 ocorrências no total). Entretanto, ao invés de 50 códigos por modelo, foram usados 36, visto que foi simulada a ocorrência de até três erros por código. O Apêndice B traz, em caráter de exemplo, as cinco ocorrências para o erro B9. No repositório do projeto no GitHub[3], todos os códigos de teste estão disponíveis;

Passo 5: Avaliação quantitativa dos modelos

Diante do conjunto de códigos de teste, foi avaliado o percentual de acerto de cada modelo, bem como métricas adicionais de desempenho – critérios que tiveram como referência inicial Pimentel *et al.* (2025). Para organização e análise dos dados coletados, foi elaborada uma matriz tridimensional no formato de planilha (correlacionando código de teste x critério x modelo), com uma aba dedicada a cada LLM, consolidando as métricas de forma comparativa entre os modelos;

Passo 6: Análise qualitativa dos modelos

Atendo-se ao mesmo conjunto de códigos de teste, foi analisada qualitativamente a natureza do *feedback* fornecido por cada modelo. A avaliação foi conduzida por meio da escala Likert de 5 pontos (sendo 1 a menor concordância e 5 a maior concordância), aplicada a critérios subjetivos relacionados à relevância pedagógica e à clareza das explicações geradas. Os dados foram consolidados na mesma matriz tridimensional utilizada na avaliação quantitativa;

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

Dados os comparativos dos passos 6 e 7, foram detalhados os custos por análise de cada modelo, estendendo-se a projeções de custos para diferentes volumes de uso.

---

1 Disponível em: cutt.ly/Bt8aUpBm.

2 Referência amplamente usada nos cursos de Computação de nível superior brasileiros. Embora algumas soluções trazidas não sejam as melhores, o livro foi uma boa referência para a pesquisa por representar o que é estudado e praticado nas disciplinas de fundamentos de programação.

3 Disponível em: github.com/pedrozampier/pc3-submission-evaluator.

4.2.2 Fase 2: Implementação do Protótipo de MVP

A Fase 2 consistiu na implementação do protótipo de MVP de API REST que materializa, em uma ferramenta funcional, o estudo comparativo conduzido na Fase 1. Diferentemente do escopo inicialmente previsto, que contemplava a integração apenas do modelo de melhor desempenho geral identificado, o protótipo final foi desenvolvido integrando simultaneamente os quatro LLMs avaliados (GPT-4, Claude, Gemini e DeepSeek). Essa mudança de escopo foi viabilizada pelo lançamento, durante o período de desenvolvimento, do Laravel AI SDK, que oferece uma camada de abstração unificada para invocação de múltiplos provedores de LLMs. O SDK permitiu que as chamadas simultâneas às quatro APIs fossem realizadas com baixo custo de implementação, tornando a integração de todos os modelos tão simples quanto a integração de apenas um. Entretanto, muito esforço anterior de implementação foi despendido antes desse lançamento do Laravel.

O protótipo foi construído reaproveitando integralmente a infraestrutura desenvolvida no Passo 5 da Fase 1 – a API REST em Laravel Framework 13, o banco de dados MariaDB 11.4 LTS e as integrações com os quatro modelos por meio do Laravel AI SDK. O resultado é um MVP que, diante de um código TypeScript submetido, é capaz de fornecer diagnósticos de erros (por meio de respostas estruturadas) de fundamentos de programação produzidos por qualquer um dos quatro LLMs avaliados, evidenciando, na prática, como esses modelos podem ser utilizados em ferramentas educacionais voltadas ao ensino de programação.

Cabe justificar, por fim, a opção pela construção de uma API REST em detrimento de alternativas mais simples de interação com os modelos, como a submissão manual de *prompts* em interfaces de conversação ou a automação das chamadas por meio de *scripts* de linha de comando (arquivos *.sh*). Dois fatores foram determinantes nessa escolha. O primeiro diz respeito à padronização e à reprodutibilidade dos experimentos: por meio da API, todas as execuções compartilharam exatamente os mesmos parâmetros de invocação e o mesmo formato estruturado de resposta, sem a interferência do contexto conversacional que as interfaces de *chat* mantêm entre mensagens e sem variações decorrentes da condução manual dos testes. O segundo fator corresponde ao reaproveitamento e à continuidade do esforço de implementação: enquanto *scripts* de chamadas constituiriam artefatos descartáveis, restritos à etapa de coleta de dados, a infraestrutura da API pôde ser integralmente reutilizada, constituindo, por si própria, o protótipo de MVP entregue como resultado prático da pesquisa (Seção 5.4). Mais do que isso, por se tratar de uma API REST de código aberto, capaz de acionar os quatro LLMs em paralelo, o protótipo estabelece uma fundação sobre a qual novos projetos podem avançar como o desenvolvimento de interfaces *web* para submissão de código, a integração com ambientes virtuais de aprendizagem e a incorporação de novos modelos – possibilidades registradas como trabalhos futuros no Capítulo 7.

4.3 Declaração de Uso de Inteligência Artificial Generativa

Declara-se que a seguinte ferramenta de Inteligência Artificial Generativa (IAG) foi utilizadas nas respectivas etapas desta pesquisa:

Claude (Anthropic)

1. Revisão da escrita do texto inicial da monografia;
2. Transposição das soluções do livro de portugol para a linguagem TypeScript, bem como estruturação dos arquivos para os modelos, a fim de compor o conjunto de códigos de teste (Fase 1, Passo 4); e
3. Injeção controlada de erros, do subconjunto definido, nos códigos de teste, para posterior diagnóstico pelos modelos (Fase 1, Passo 4).

Todo o processo, bem como os resultados, foram criteriosamente revisados pelo acadêmico. Além disso, o acadêmico assume integral responsabilidade pelo conteúdo intelectual e técnico da pesquisa, em conformidade com a Portaria CNPq nº 2.664/2026. O acadêmico ainda declara que o conteúdo transcrito neste documento é autoral ou devidamente referenciado, não sendo usadas ferramentas de Inteligência Artificial para geração. A revisão do documento final foi feita manualmente pelo orientador da pesquisa.

5 RESULTADOS

O presente capítulo sumariza os resultados decorrentes da pesquisa, sendo: adaptação do subconjunto de erros à linguagem TypeScript (Seção 5.1), composição de um conjunto padronizado (*corpus*) de códigos de teste (Seção 5.2), análise comparativa entre os modelos (Seção 5.3) e implementação do protótipo de MVP (Seção 5.4).

5.1 Adaptação do Subconjunto de Erros à Linguagem TypeScript

Como primeiro resultado, tem-se a adaptação do subconjunto de erros, identificados por Silva, Caceffo e Azevedo (2023) em Python, para a linguagem TypeScript. A adaptação envolveu 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.

A adaptação transpôs para TypeScript não apenas os 10 erros selecionados para a pesquisa, mas se estendeu aos 18 de maior gravidade do conjunto de 45 erros (portanto, 40% do catálogo foi adaptado). O catálogo de erros encontra-se no Apêndice B e a versão atualizada (e estendida) pode ser consultada on-line[1].

5.2 Composição de um Conjunto Padronizado (*Corpus*) de Códigos de Teste

O conjunto padronizado (*corpus*) de códigos de teste foi idealizado de forma que, para cada um dos 10 erros, houvesse 5 ocorrências (instâncias de teste) daquele erro, totalizando 50 ocorrências por modelo (50 x 4 modelos = 200 ocorrências no total). Todavia, foram usados 36 códigos por modelo, ao invés de 50, visto que foi simulada a ocorrência de até três erros por código.

O identificador de cada código de teste é constituído obedecendo o formato: *[Erro] – [Instância de Teste] _ [Enunciado]*. Portanto, o Código de Teste *B9–1_097* trata-se do erro B9, instância de teste 1 (de 5) e diz respeito ao enunciado 97 do livro *Introdução à Programação: 500 Algoritmos Resolvidos* (LOPES; GARCIA, 2002). Em caráter de exemplo, o Apêndice B reúne as cinco ocorrências para o erro B9. No repositório do projeto no GitHub, todos os códigos de teste podem ser acessados.

5.3 Análise Comparativa entre os Modelos

Serão descritos os resultados alcançados a partir da avaliação comparativa dos quatro modelos de linguagem ampla (LLMs) considerados nesta pesquisa – a saber, GPT-4 (OpenAI),

[1] Disponível em: cutt.ly/Bt8aUpBm.

Claude (Anthropic), Gemini (Google) e DeepSeek – quanto à capacidade de diagnóstico automatizado de erros de compreensão (PC^3) em códigos de teste na linguagem TypeScript. Para cada modelo, foram realizadas 50 execuções de teste (conforme *corpus* de teste descrito na Seção 5.2), distribuídas entre as diferentes categorias do subconjunto de erros adaptado (Seção 5.1), considerando tanto repetições de um mesmo código de teste quanto variações entre diferentes códigos.

Os resultados são apresentados sob duas perspectivas complementares: critérios quantitativos (Seção 5.3.1), correspondentes a métricas objetivas mensuráveis diretamente a partir dos dados de execução; e critérios qualitativos (Seção 5.3.2), que avaliam, por meio de uma escala Likert de 5 pontos, a qualidade pedagógica do *feedback* textual fornecido por cada modelo. Para cada critério, é apresentada inicialmente uma definição conceitual, seguida de um comparativo entre os quatro modelos avaliados.

5.3.1 Critérios Quantitativos

Os critérios quantitativos correspondem a métricas objetivas, coletadas automaticamente durante a execução dos testes, que permitem comparar os modelos sob aspectos de desempenho técnico, operacional e de assertividade. A Tabela 2 sintetiza os valores médios obtidos para cada um desses critérios, considerando as 50 execuções (*corpus* de teste) realizadas por modelo. Em seguida, cada critério é detalhado, juntamente com os respectivos resultados.

Tabela 2 – Síntese comparativa dos critérios quantitativos por modelo

Modelo	Tempo de resposta (ms)	Alucinações/execução	Tokens (média)	Custo médio (US$)	Acertos (%)
GPT-4 (OpenAI)	2.354	0,64	761	0,0445	78,7
Claude (Anthropic)	5.783	0,76	857	0,0752	74,7
Gemini (Google)	6.567	0,42	1.915	0,0280	86,0
DeepSeek	2.270	0,66	1.348	0,0022	78,7

Fonte: Autoria própria.

QT1 – Tempo de resposta

O tempo de resposta corresponde à latência mediana, em milissegundos, observada entre o envio da requisição contendo o código de teste e o recebimento da resposta completa de cada modelo. Trata-se de um indicador relevante para a viabilidade prática de uma ferramenta educacional baseada em LLM, uma vez que tempos de resposta elevados podem comprometer a experiência do usuário e a fluidez da interação em cenários de uso real, como em uma ferramenta de *feedback* integrada ao ambiente de desenvolvimento, ou de aprendizagem, de um estudante.

Os resultados mostram uma divisão clara entre dois grupos de modelos. GPT-4 (2.354 ms, em média) e DeepSeek (2.270 ms) apresentaram os menores tempos de resposta, situando-se na casa dos dois segundos. Por vez, Claude (5.783 ms) e Gemini (6.567 ms) apresentaram tempos de resposta entre duas e três vezes maiores, o

que pode ser parcialmente explicado pela maior quantidade de *tokens* processados por esses modelos (conforme critério QT4). O Gemini, especificamente, foi o modelo mais lento dentre os quatro avaliados.

QT2 – Validade da resposta

Esse critério verifica, de forma binária ("sim" ou "não"), se o modelo foi capaz de retornar uma resposta no formato estruturado solicitado, ou seja, um objeto JSON contendo a lista de problemas identificados, conforme especificado no *prompt* de testes (Apêndice A), sem falhas de comunicação, truncamentos ou desvios de formatação que impedissem o processamento automatizado da resposta pelo sistema avaliador.

Os quatro modelos avaliados retornaram respostas válidas em 100% das execuções (50 de 50 em cada caso). Esse resultado indica que todas as integrações via API mostraram-se tecnicamente estáveis e que os quatro modelos foram capazes de respeitar consistentemente o formato de saída estruturado solicitado, eliminando essa dimensão como fator de diferenciação entre modelos na pesquisa.

QT3 – Consistência (alucinações por execução)

A consistência foi avaliada a partir da repetição da mesma requisição de análise para um mesmo código de teste, verificando-se a ocorrência de alucinações, isto é, divergências entre as respostas obtidas – como a indicação de problemas inexistentes no código, elementos (funções, variáveis, sintaxes) inventados pelo modelo, ou diagnósticos contraditórios entre execuções equivalentes (conforme definido na Seção 2.1.2). Cada execução foi classificada quanto ao número de alucinações observadas (0, 1, 2 ou 3).

O Gemini apresentou a melhor consistência entre os modelos avaliados, com média de apenas 0,42 alucinações por execução e 78% (39 de 50) das execuções completamente livres de alucinações. Em seguida, aparecem o GPT-4 (0,64 alucinações em média; 72% sem alucinação) e o DeepSeek (0,66; 64% sem alucinação), com desempenhos relativamente próximos. O Claude apresentou a maior média de alucinações por execução (0,76), embora tenha ficado à frente do DeepSeek quanto ao percentual de execuções totalmente livres de alucinações (66%). De modo geral, observa-se que alucinações tendem a se concentrar em categorias específicas de erros (como G3 e G4[2]), sugerindo que a consistência dos modelos pode variar conforme a natureza do problema analisado, e não apenas conforme características intrínsecas de cada modelo.

QT4 – Tokens usados

O critério registra a quantidade de *tokens* (Seção 2.1.2) consumidos em cada execução, somando-se os *tokens* de entrada (*prompt* e código submetido) e de saída (resposta

[2] Respectivamente, G3 = muitas declarações em uma única linha de código e G4 = identificadores com nomes não significativos.

gerada pelo modelo). Trata-se de um critério diretamente relacionado tanto ao desempenho quanto ao custo operacional de cada modelo, pois serve de base para o cálculo do valor cobrado por requisição (critério QT5).

Verificou-se uma diferença expressiva entre os modelos quanto ao volume de *tokens* processados. O Gemini consumiu, em média, 1.915 *tokens* por execução, mais que o dobro do GPT-4 (761) e do Claude (857), seguido pelo DeepSeek, com consumo intermediário (1.348). Esse maior consumo de *tokens* pelo Gemini está associado tanto ao seu maior tempo de resposta quanto à extensão das explicações geradas, podendo refletir uma tendência desse modelo a produzir respostas mais detalhadas e elaboradas. Já o GPT-4 destacou-se pela maior economia de *tokens*, o que sugere respostas mais concisas e diretas para a mesma tarefa.

QT5 – Custo da requisição

O custo da requisição corresponde ao valor monetário (em dólares) de cada execução, obtido pela multiplicação entre a quantidade de *tokens* consumidos (critério QT4) e o valor cobrado por *token* em cada plataforma. Esse critério é central para a correlação entre desempenho e viabilidade econômica proposta nesta pesquisa, pois projeta diretamente os custos operacionais de uma eventual ferramenta educacional construída sobre cada um dos modelos avaliados.

As diferenças de custo observadas entre os modelos foram substanciais. O DeepSeek apresentou um custo médio por requisição de aproximadamente US$ 0,0022, entre 12 e 34 vezes inferior ao dos demais modelos, consolidando-se como a opção mais econômica dentre as avaliadas. Em seguida, aparece o Gemini (US$ 0,0280), depois o GPT-4 (US$ 0,0445) e, por fim, o Claude, que apresentou o maior custo médio por requisição (US$ 0,0752), mais de 30 vezes superior ao do DeepSeek. Esse panorama evidencia que o valor cobrado por *token* de cada provedor tem peso decisivo no custo final, podendo superar até mesmo o efeito do volume de *tokens* consumidos, como ocorre no caso do Gemini, que, apesar de processar significativamente mais *tokens* que o Claude, apresenta um custo médio por requisição inferior.

QT6 – Quantidade de acertos

A quantidade de acertos registra o número de erros corretamente identificados pelo modelo dentre os presentes em cada código de teste, podendo variar de 0 a 3, visto que cada código de teste foi construído para conter, estrategicamente, até três ocorrências do erro investigado (Seção 4.2). Trata-se do critério mais diretamente associado à eficácia do modelo na tarefa central desta pesquisa: o diagnóstico automatizado de erros de compreensão em código.

O Gemini apresentou o melhor desempenho nesse quesito, identificando corretamente 129 dos 150 erros possíveis (86%), valor expressivamente superior ao dos demais mo-

delos. GPT-4 e DeepSeek obtiveram desempenho idêntico, com 118 acertos (78,7% cada), enquanto o Claude apresentou o menor índice de acertos, com 112 (74,7%). Observa-se ainda que os erros das categorias G3 (muitas declarações em uma única linha de código) e G4 (identificadores com nomes não significativos) concentraram a maior parte das execuções com zero acertos entre os modelos, sugerindo que esses tipos de erro, de natureza mais estilística que estrutural, representam um desafio maior para o diagnóstico automatizado por LLMs do que erros de natureza lógica ou estrutural, como os das categorias B9, B8, B12, C1, C3 e H1[3], nos quais os quatro modelos apresentaram desempenho consistentemente alto.

5.3.2 Critérios Qualitativos

Os critérios qualitativos foram avaliados por meio de uma escala Likert de 5 pontos (ARAUJO *et al*., 2024) (sendo 1 a menor concordância e 5 a maior concordância com a afirmação avaliada), aplicada à análise do *feedback* textual gerado por cada modelo diante dos códigos de teste. O propósito dessa avaliação foi captar aspectos subjetivos relacionados à qualidade pedagógica das explicações fornecidas, dimensão essencial no contexto educacional desta pesquisa, mas que não pode ser plenamente representada por métricas puramente quantitativas. Cabe destacar que, nas execuções em que o modelo não identificou nenhum dos erros presentes no código (zero acertos), a pontuação qualitativa atribuída foi nula, refletindo a ausência de um *feedback* pedagogicamente relevante a ser avaliado nesses casos.

A Tabela 3 apresenta a média obtida por cada modelo em cada um dos seis critérios qualitativos avaliados, considerando as 50 execuções realizadas (*corpus* de teste). Cada critério é detalhado, em diante da respectiva definição e comparativo, nas subseções seguintes.

Tabela 3 – Síntese comparativa dos critérios qualitativos por modelo

Modelo	Contexto	Localização	Utilidade	Especificidade	Clareza	Precisão conceitual
GPT-4 (OpenAI)	3,84	3,78	3,80	4,02	3,92	3,76
Claude (Anthropic)	3,74	3,72	3,66	3,86	3,62	3,62
Gemini (Google)	3,94	3,92	3,96	3,98	3,96	3,90
DeepSeek	3,98	3,70	3,98	4,02	3,98	3,84

Fonte: Autoria própria.

QL1 – Compreensão correta do contexto da solução

Pergunta: Compreende corretamente o contexto da solução?

O critério avalia se a explicação textual gerada pelo modelo mostra uma compreensão adequada do propósito e do funcionamento geral do código analisado, antes de discorrer

[3] Respectivamente, B9 = instrução *else-if* retestando condições já verificadas; B8 = não utilização correta da estrutura *if-else*; B12 = consecutivas declarações de *if*'s iguais com operações distintas nos blocos; C1 = condição *while* testada novamente no interior do bloco; C3 = operações redundantes dentro de um laço; e H1 = código sem efeito.

sobre os problemas identificados. Essa capacidade é considerada fundamental para que o *feedback* seja percebido como relevante e confiável pelo estudante, e não como uma crítica genérica desconexa do contexto da tentativa de solução.

O DeepSeek obteve a maior média nesse quesito (3,98), seguido de perto pelo Gemini (3,94). Por vez, GPT-4 (3,84) e Claude (3,74) apresentaram médias inferiores. A proximidade entre os quatro modelos sugere que, de modo geral, todos apresentaram capacidade satisfatória de compreender o propósito dos códigos de teste antes de apontar os problemas neles contidos.

QL2 – Precisão ao localizar o erro

Pergunta: Localiza precisamente o erro?
Mede o quanto a explicação aponta, de forma específica, a localização do problema identificado no código. Isso é feito, por exemplo, indicando a linha, o trecho ou o elemento (variável, função, estrutura) envolvido, em contraposição a diagnósticos vagos ou genéricos que dificultam o entendimento e a correção pelo estudante.

O Gemini apresentou a maior média nesse critério (3,92), seguido pelo GPT-4 (3,78) e pelo Claude (3,72). O DeepSeek, apesar do bom desempenho nos demais critérios qualitativos, obteve a menor média nesse quesito específico (3,70), indicando que, embora costume identificar corretamente os problemas (critério QL1), nem sempre consegue apontar, com a mesma precisão, onde exatamente eles ocorrem no código – aspecto relevante a ser considerado caso esse modelo seja escolhido para compor o protótipo de MVP.

QL3 – Utilidade da explicação para o aprendizado

Pergunta: A explicação é útil para o aprendizado de um estudante?
Avalia o valor pedagógico da explicação como um todo, ou seja, se ela contribui efetivamente para que um estudante iniciante compreenda o erro cometido e avance em sua aprendizagem. Isso, ao invés de apenas apontar o problema de forma corretiva, sem agregar entendimento conceitual.

DeepSeek (3,98) e Gemini (3,96) apresentaram as maiores médias nesse critério, com desempenho bastante próximo entre si. O GPT-4 (3,80) e, principalmente, o Claude (3,66) ficaram atrás, com este último apresentando a menor pontuação entre os quatro modelos. Esse resultado sugere que, no conjunto de testes avaliado, DeepSeek e Gemini tenderam a produzir explicações percebidas como mais úteis sob a ótica do aprendizado para o estudante.

QL4 – Especificidade da explicação para o código analisado

Pergunta: A explicação é específica para o código analisado (ou é demasiadamente genérica)?

Verifica se o *feedback* textual faz referência direta a elementos concretos do código submetido (como nomes de variáveis, estruturas e trechos específicos), em vez de apresentar respostas genéricas que poderiam se aplicar a qualquer código com problema semelhante, aspecto que impacta diretamente a percepção de relevância e de cuidado do retorno fornecido ao estudante.

DeepSeek e GPT-4 empataram com a maior média nesse critério (4,02 cada), seguidos de perto pelo Gemini (3,98). O Claude apresentou a menor média (3,86), ainda que próxima das demais. Esse foi o critério qualitativo em que os quatro modelos apresentaram, de modo geral, as melhores avaliações, sugerindo que, quando conseguem identificar o problema, todos tendem a relacionar a explicação ao código efetivamente submetido.

QL5 – Clareza da explicação

Pergunta: A explicação é clara?

Avalia a compreensibilidade da mensagem e linguagem utilizada na explicação, considerando aspectos como objetividade, organização das ideias e ausência de jargões desnecessários, fator essencial para que o *feedback* cumpra seu papel pedagógico junto a estudantes iniciantes, que costumam ter dificuldade para interpretar mensagens técnicas (Seção 3.1).

DeepSeek (3,98) e Gemini (3,96) novamente se destacaram com as maiores médias, seguidos pelo GPT-4 (3,92). O Claude apresentou a menor média nesse critério (3,62), com diferença mais expressiva em relação aos demais modelos do que a observada nos outros critérios qualitativos, sugerindo que a clareza das explicações foi, entre os aspectos avaliados, um dos pontos relativamente mais frágeis desse modelo no conjunto de testes realizado.

QL6 – Precisão conceitual da explicação

Pergunta: A explicação é conceitualmente precisa?

Avalia se o conteúdo da explicação está tecnicamente correto do ponto de vista conceitual, sem introduzir equívocos, simplificações inadequadas ou informações conflitantes com os fundamentos de programação, aspecto crítico no contexto educacional desta pesquisa. Eventuais imprecisões conceituais no *feedback* podem reforçar, em vez de corrigir, concepções erradas (*misconceptions*) já presentes no raciocínio do estudante.

O Gemini apresentou a maior média nesse critério (3,90), seguido pelo DeepSeek (3,84), pelo GPT-4 (3,76) e pelo Claude, com a menor média (3,62). Tomados em conjunto com os critérios anteriores, esses resultados indicam que, embora os quatro modelos tenham demonstrado capacidade de gerar explicações compreensíveis, Gemini e DeepSeek tenderam a produzir *feedback* qualitativamente superior na maior parte dos critérios avaliados, ao passo que o Claude apresentou, de forma recorrente, as menores

médias qualitativas, padrão que será retomado e aprofundado no capítulo de discussão (Capítulo 6).

5.3.3 Ranqueamento dos Modelos

De forma sumarizada, esta subseção apresenta o ranqueamento dos quatro modelos avaliados sob duas perspectivas complementares: a eficácia diagnóstica (Tabela 4), que reúne os critérios relacionados à capacidade de identificar corretamente os erros e à qualidade pedagógica do *feedback* gerado; e a viabilidade operacional (Tabela 5), que reúne os critérios relacionados ao custo prático de operação do modelo em produção. A separação em duas perspectivas foi adotada por reconhecer que essas dimensões podem ser conflitantes entre si, e que uma única nota agregada poderia ocultar *trade-offs* relevantes para a escolha de um modelo em ferramentas educacionais reais.

Para permitir a comparação direta entre critérios de natureza diversa, os critérios quantitativos foram normalizados para a escala de 1 a 5 por meio da técnica *min-max*, com inversão para os critérios em que "menor é melhor" (tempo de resposta, alucinações, *tokens* usados e custo da requisição), de modo que, em todos os critérios, a nota maior corresponde ao melhor desempenho. Os critérios qualitativos, já originalmente em escala Likert de 1 a 5, foram utilizados diretamente. A nota final de cada modelo, em cada perspectiva, corresponde à média aritmética simples das notas atribuídas em seus respectivos critérios.

Tabela 4 – Ranqueamento dos modelos por eficácia diagnóstica

Classificação	Modelo	Nota (1–5)
1º	Gemini	4,30
2º	DeepSeek	3,68
3º	GPT-4	3,66
4º	Claude	3,25
	Média Geral	**3,72**

Fonte: Autoria própria.

Tabela 5 – Ranqueamento dos modelos por viabilidade operacional

Classificação	Modelo	Nota (1–5)
1º	DeepSeek	4,32
2º	GPT-4	4,20
3º	Claude	2,47
4º	Gemini	1,86
	Média Geral	**3,21**

Fonte: Autoria própria.

A análise conjunta dos dois ranqueamentos evidencia que não há, entre os modelos avaliados, um vencedor absoluto em todas as dimensões, ainda que o DeepSeek se destaque por figurar entre os dois melhores em ambas as perspectivas. O Gemini destaca-se como o

modelo de melhor eficácia diagnóstica (nota 4,30), com o maior percentual de acertos, a menor taxa de alucinações e as maiores médias na maioria dos critérios qualitativos, mas é penalizado em viabilidade operacional (nota 1,86), ocupando a última posição nessa perspectiva em razão do maior tempo de resposta e do maior consumo de *tokens* dentre os modelos avaliados. O DeepSeek apresenta o quadro mais equilibrado: lidera a viabilidade operacional (nota 4,32), com custo por requisição entre 12 e 34 vezes inferior ao dos demais e o menor tempo de resposta, e ainda ocupa a segunda posição em eficácia diagnóstica (nota 3,68), praticamente empatado com o GPT-4. Por vez, o GPT-4 ocupa a terceira posição em eficácia diagnóstica (nota 3,66) e a segunda em viabilidade operacional (nota 4,20), mantendo um desempenho consistente, ainda que sem liderar nenhuma das perspectivas. O Claude não lidera em nenhuma das perspectivas, apresentando as menores médias nos critérios qualitativos e o maior custo médio por requisição.

Esse padrão evidencia um *trade-off* concreto entre qualidade diagnóstica e custo operacional, com implicações diretas para a escolha do modelo em ferramentas educacionais: contextos em que a qualidade pedagógica do *feedback* é prioritária e o custo não é restritivo tendem a favorecer o Gemini; já contextos em que a escalabilidade econômica é central, como em instituições públicas com uso em larga escala, ou que demandam um equilíbrio razoável entre as duas dimensões, tendem a favorecer o DeepSeek, tendo o GPT-4 como alternativa de desempenho mais consistente. Esse achado é aprofundado e discutido no Capítulo 6.

5.4 Implementação do Protótipo de MVP

A implementação do protótipo de MVP consolidou, em uma ferramenta funcional, os resultados da análise comparativa conduzida nesta pesquisa. O protótipo foi desenvolvido como uma API REST que recebe códigos TypeScript submetidos pelo usuário e retorna diagnósticos estruturados de erros de fundamentos de programação, gerados por qualquer um dos quatro LLMs avaliados (GPT-4, Claude, Gemini e DeepSeek). A integração simultânea com os quatro modelos, viabilizada pelo Laravel AI SDK, permite que a mesma submissão de código seja analisada em paralelo pelos diferentes modelos, oferecendo ao usuário a possibilidade de comparar diretamente o *feedback* produzido (vide Apêndice D), em aplicação prática do *trade-off* entre eficácia diagnóstica e viabilidade operacional identificado na Seção 5.3.3.

O protótipo encontra-se em estado funcional na conclusão desta pesquisa, com a infraestrutura de integração estável e validada por meio das execuções realizadas durante a coleta de dados dos experimentos (seções 5.3.1 e 5.3.2). O código-fonte está disponível em repositório público[4], podendo servir de ponto de partida para o desenvolvimento de pesquisas ou ferramentas educacionais voltadas ao ensino de fundamentos de programação em TypeScript.

[4] Disponível em: github.com/pedrozampier/pc3-submission-evaluator.

6 DISCUSSÃO

Este capítulo discute os resultados apresentados no Capítulo 5, articulando-os entre si e relacionando-os ao problema de pesquisa (Seção 1.2), aos objetivos estabelecidos (Seção 1.7) e ao referencial teórico apresentado (Capítulo 2). Inicialmente, é apresentada uma síntese comparativa geral entre os quatro modelos avaliados (Seção 6.1). Em seguida, são discutidas as implicações pedagógicas dos achados para o contexto do ensino de programação (Seção 6.2). Por fim, são apontadas as principais limitações do estudo, que devem ser consideradas na interpretação dos resultados (Seção 6.3).

6.1 Síntese Comparativa Geral

Ao se observar, em conjunto, os critérios quantitativos e qualitativos apresentados no capítulo anterior, nota-se que nenhum dos quatro modelos avaliados se sobressai de forma absoluta em todas as dimensões consideradas. Ao contrário, cada um apresenta um perfil de *trade-offs* (compensações) distinto entre assertividade, qualidade do *feedback*, velocidade de resposta e custo operacional.

O **Gemini** apresentou o melhor desempenho quantitativo na tarefa central da pesquisa, identificando corretamente 86,0% dos erros propostos, percentual expressivamente superior ao dos demais, além de ter sido o modelo mais consistente (menor índice de alucinações) e um dos mais bem avaliados qualitativamente. Esse desempenho, no entanto, tem como contrapartida o maior tempo de resposta entre os quatro modelos (em média, 6.567 ms) e o maior consumo de *tokens* por execução (1.915, em média), fatores que podem comprometer a fluidez de uso em aplicações que demandem respostas em tempo real.

O **DeepSeek**, por sua vez, destacou-se pelo melhor custo-benefício entre os modelos avaliados: obteve o segundo melhor tempo de resposta (2.270 ms, em média), o menor custo médio por requisição (entre 12 e 34 vezes inferior ao dos demais, conforme critério QT5) e médias qualitativas competitivas, equiparando-se ao Gemini em quesitos como utilidade pedagógica e clareza da explicação (critérios QL3 e QL5). Sua principal fragilidade esteve na localização precisa dos erros apontados (critério QL2), em que obteve a menor média entre os quatro modelos, ainda que tenha mantido uma taxa de acertos equivalente à do GPT-4.

O **GPT-4** apresentou um perfil de desempenho intermediário e equilibrado: taxa de acertos idêntica à do DeepSeek (78,7%), tempo de resposta competitivo (2.354 ms, em média) e médias qualitativas consistentemente próximas da segunda colocação, sem, no entanto, liderar isoladamente em nenhum dos critérios quantitativos centrais. Seu menor consumo médio de *tokens* (761) sugere a geração de respostas mais concisas, o que pode ser vantajoso ou desvantajoso a depender do nível de detalhamento desejado para o *feedback* pedagógico.

Já o **Claude** apresentou o desempenho mais modesto dentre os quatro modelos comparados: menor percentual de acertos (74,7%), maior custo médio por requisição (US$ 0,0752)

e as menores médias em praticamente todos os critérios qualitativos avaliados, com destaque para clareza (3,62) e precisão conceitual (3,62), pontuações mais baixas registradas entre os 24 pares modelo-critério qualitativo analisados. Isso não significa que o modelo seja inadequado para a tarefa, já que sua taxa de acertos permanece relativamente próxima àquela dos demais – mas sim que, no recorte específico desta pesquisa (diagnóstico de erros de compreensão em código TypeScript, com *feedback* em português), apresentou resultados consistentemente inferiores aos dos concorrentes.

De modo geral, os resultados sugerem que **Gemini** e **DeepSeek** reúnem, juntos, os atributos mais relevantes para a proposta desta pesquisa: o primeiro pela assertividade no diagnóstico e pela qualidade do *feedback*; o segundo pelo equilíbrio entre desempenho, agilidade e, sobretudo, viabilidade econômica. Essa constatação é especialmente relevante para o terceiro objetivo específico desta pesquisa (Seção 1.7.2), que prevê a implementação de um protótipo de MVP integrado ao modelo de melhor desempenho geral identificado, sugerindo que tanto o Gemini quanto o DeepSeek constituem candidatos plausíveis para essa etapa, a depender de qual dimensão (assertividade ou custo operacional) for priorizada na decisão final.

6.2 Implicações Pedagógicas

Os resultados alcançados trazem implicações relevantes para a discussão, já apresentada na fundamentação teórica e nos trabalhos relacionados (capítulos 2 e 3), sobre o papel das LLMs no apoio ao ensino de programação.

Em primeiro lugar, o fato de os quatro modelos terem retornado respostas estruturadas e válidas em 100% das execuções (critério QT2), associado a taxas de acerto que variaram entre 74,7% e 86,0%, corrobora o que apontam Leinonen *et al.* (2023) e Finnie-Ansley *et al.* (2022) a respeito do potencial das LLMs em auxiliar no diagnóstico de problemas em código além da simples verificação sintática realizada por compiladores tradicionais. Ao mesmo tempo, nenhum modelo atingiu 100% de acerto, o que reforça a importância de não se depositar confiança irrestrita nesses sistemas em contextos educacionais, indo ao encontro das ressalvas levantadas por Prather *et al.* (2023) quanto ao risco de alucinações e diagnósticos equivocados.

Em segundo lugar, observou-se que os erros das categorias G3 (muitas declarações em uma única linha de código) e G4 (identificadores com nomes não significativos) representaram, de forma recorrente, o maior desafio para todos os modelos avaliados, concentrando a maioria das execuções com zero acertos (critério QT6). Esse achado sugere que LLMs tendem a apresentar maior dificuldade em identificar problemas de natureza estilística, que dependem de critérios mais subjetivos, como aquilo que torna um nome de variável “significativo”, do que problemas de natureza estrutural ou lógica, como laços mal construídos ou comparações booleanas redundantes, nos quais o desempenho dos quatro modelos foi consistentemente superior. Essa distinção pode orientar decisões de *design* em ferramentas educacionais futuras, por exemplo, priorizando o uso de LLMs para o diagnóstico de problemas estruturais e reservando

outras abordagens (como análise estática de código ou regras heurísticas) para aspectos mais estilísticos.

Em terceiro lugar, a análise qualitativa reforça a percepção de que assertividade técnica e qualidade pedagógica do *feedback* não são, necessariamente, dimensões equivalentes. Modelos com taxas de acerto semelhantes (como GPT-4 e DeepSeek, ambos com 78,7%) apresentaram perfis qualitativos distintos entre si, cada um com pontos fortes e fracos específicos (critérios QL1 e QL6). Esse resultado sugere que, no desenvolvimento de ferramentas educacionais baseadas em LLMs, a escolha do modelo não deve se basear apenas em métricas de acerto, mas também, e talvez sobretudo, na adequação do *feedback* gerado às necessidades pedagógicas do público-alvo, alinhando-se ao alerta de Rosa *et al.* (2025) e Filho, Souza e Paula (2023) sobre a instabilidade e a heterogeneidade das respostas de LLMs em contextos de ensino de programação.

Por fim, a expressiva diferença de custos observada entre os modelos (critério QT5), com o DeepSeek apresentando custos até 34 vezes menores que o Claude, evidencia que considerações puramente técnicas (acerto e qualidade) não são suficientes para orientar a adoção de LLMs em ferramentas educacionais de uso contínuo. Em cenários de uso em larga escala, como uma disciplina com centenas de estudantes submetendo código repetidamente ao longo de um semestre, diferenças de custo dessa magnitude podem ser determinantes para a viabilidade financeira do projeto, especialmente em instituições públicas brasileiras com recursos limitados, reforçando a relevância da correlação entre desempenho e custo operacional proposta como um dos objetivos específicos desta pesquisa (Seção 1.7.2).

6.3 Limitações do Estudo

Os resultados e as discussões apresentados devem ser interpretados à luz de algumas limitações inerentes ao escopo e ao desenho desta pesquisa. Primeiramente, a avaliação concentrou-se em um subconjunto restrito de erros, composto por dez categorias de PC^3 (B6, B8, B9, B12, C1, C3, C8, G3, G4 e H1) selecionadas a partir dos critérios de gravidade discutidos na Seção 5.1, e não na totalidade dos 45 erros originalmente identificados por Silva, Caceffo e Azevedo (2023). Embora a seleção tenha priorizado erros de maior consenso quanto à gravidade, é possível que os modelos apresentem desempenhos distintos diante de categorias não contempladas neste estudo, o que limita a generalização dos resultados para o espectro completo de problemas de compreensão em código.

Em segundo lugar, o volume de execuções (50 por modelo, totalizando 200 execuções), embora suficiente para identificar tendências e diferenças relevantes entre os modelos, constitui uma amostra relativamente modesta do ponto de vista estatístico, sobretudo quando segmentada por categoria de erro (média de cerca de cinco execuções por categoria, por modelo). Estudos futuros poderiam ampliar esse volume para consolidar estatisticamente as diferenças

observadas, sobretudo aquelas mais sutis entre modelos com desempenho próximo, como GPT-4 e DeepSeek.

Em terceiro lugar, a pesquisa restringiu-se à linguagem TypeScript e a códigos sintéticos, elaborados especificamente para conter, de forma controlada, os erros do subconjunto selecionado (Seção 4.2). Embora essa abordagem tenha sido necessária para garantir comparabilidade entre os modelos, ela não captura toda a diversidade e a imprevisibilidade presentes em códigos reais produzidos por estudantes (demanda trâmite e aprovação prévia pelo Comitê de Ética em Pesquisa), o que pode influenciar tanto a taxa de acertos quanto a qualidade do *feedback* observadas em um cenário de uso real.

Em quarto lugar, a avaliação qualitativa apoiou-se em uma escala Likert de 5 pontos atribuída pelo próprio autor da pesquisa, o que introduz um componente de subjetividade ao processo, ainda que mitigado pela aplicação de critérios bem definidos e uniformes a todos os modelos (critérios QL1 e QL6). A ausência de múltiplos avaliadores (por exemplo, professores ou estudantes de fundamentos de programação) limita a validação externa dessas pontuações, podendo ser endereçada em trabalhos futuros por meio de avaliações cegas e independentes.

Por fim, cabe destacar que os resultados refletem o desempenho das versões específicas dos modelos disponíveis no período em que os experimentos foram realizados, bem como os valores de custo por *token* vigentes nesse momento. Dada a velocidade com que LLMs evoluem, tanto em capacidades quanto em política de preços, é possível que novas versões dos modelos avaliados (ou de seus concorrentes) apresentem resultados distintos dos aqui relatados, reforçando o caráter de *benchmark* replicável e atualizável que se buscou conferir a esta pesquisa.

7 CONSIDERAÇÕES FINAIS

A presente pesquisa realizou um comparativo sobre a capacidade de diagnóstico automatizado de erros frequentes em fundamentos de programação, considerando códigos em TypeScript, por quatro modelos de linguagem ampla. Foram estabelecidos critérios qualitativos e quantitativos, correlacionando-os com o custo operacional de cada modelo. Também, foi implementado um protótipo de MVP funcional tanto para que a comparação fosse realizada quanto para mostrar a aplicabilidade prática do modelo de melhor desempenho na comparação.

Destaca-se que, durante o desenvolvimento, os objetivos originais da pesquisa foram estendidos. Primeiro, abrangendo quatro ao invés de três modelos. Segundo, pela implementação do protótipo de MVP contemplar não apenas o modelo de melhor desempenho, mas todos os quatro modelos considerados.

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 traz evidências para orientar escolhas tecnológicas em ferramentas educacionais. A partir disso, também se buscou 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, consiste em um exemplo do uso das evidências fornecidas para o desenvolvimento de ferramentas educacionais no escopo abrangido.

Como resultados imediatos da pesquisa, feita em nível de graduação, tem-se: (a) adaptação do subconjunto de erros à linguagem TypeScript (Seção 5.1); (b) composição de um conjunto padronizado (*corpus*) de códigos de teste (Seção 5.2); (c) análise comparativa entre os modelos (Seção 5.3); e (d) implementação do protótipo de MVP (Seção 5.4). Além dos resultados inerentes à pesquisa, registra-se a evolução proporcionada ao acadêmico, uma vez que aspectos de pesquisa e escrita científicas não concernem ao enfoque do curso. Dessa forma, foi possibilitado o desenvolvimento de capacidades que são requeridas ao se avançar para um curso de pós-graduação.

Dentre as principais limitações, relatam-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; (c) amostra de códigos de teste (sintéticos) relativamente modesta diante do ponto de vista estatístico, com um único avaliador para os critérios qualitativos; e (d) avaliação direcionada aos quatro LLMs específicos selecionados, considerando as versões atuais.

Ainda a respeito das escolhas de projeto, é razoável supor que modelos mais recentes – e, sobretudo, os modelos pagos de nível superior oferecidos por cada provedor – elevariam as avaliações obtidas, tanto nos critérios quantitativos quanto nos qualitativos. A opção pelos modelos de nível intermediário de cada provedor foi, entretanto, deliberada: prezou-se por favorecer a replicabilidade da pesquisa, permitindo que os experimentos sejam reexecutados inclusive di-

ante de novas versões dos modelos sem que os custos operacionais se tornem impeditivos, em consonância com o caráter de *benchmark* replicável e atualizável discutido na Seção 6.3.

Com a intenção de avançar frente às limitações, como também estender a presente pesquisa, citam-se como trabalhos futuros:

1. Estender o subconjunto de erros considerado e generalizar o diagnóstico para outras linguagens de programação;

2. Utilizar um segundo conjunto (*corpus*) de código de testes com instâncias de soluções reais produzidas por estudantes, com uma amostra numericamente mais significativa, bem como adotar múltiplos avaliadores para os critérios qualitativos;

3. Acompanhar, diante da análise comparativa, as novas versões dos quatro LLMs selecionados, como também permitir fácil incorporação de novos modelos que possam surgir;

4. Evoluir o protótipo para uma plataforma completa, dotada de interface web para submissão de código e visualização dos diagnósticos, persistência do histórico de avaliações e gerenciamento de usuários, de modo a viabilizar o uso por docentes e estudantes em contexto educacional real;

5. Investigar a especialização de *prompts* por elemento da classificação, formulando instruções distintas conforme a categoria de erro PC^3 sob análise; e avaliando se a contextualização específica eleva a acurácia do diagnóstico em relação ao *prompt* único e generalista adotado nesta pesquisa;

6. Implementar mecanismo de autenticação e autorização nas chamadas à API REST, por meio de *tokens* de acesso, garantindo o controle de uso, a rastreabilidade das requisições e a proteção dos recursos expostos pelo protótipo;

7. Desenvolver um *plugin* de integração com o ambiente virtual de aprendizagem Moodle, permitindo que o diagnóstico automático de erros seja acionado diretamente a partir das atividades submetidas pelos estudantes, sem a necessidade de ferramentas externas;

8. Adotar processamento assíncrono das chamadas aos LLMs por meio de *jobs* em fila, evitando o bloqueio das requisições durante o tempo de resposta dos modelos e conferindo maior escalabilidade e resiliência à plataforma;

9. Replicar o experimento comparativo com modelos de linguagem abertos executados localmente, como Qwen e Gemma, inclusive em versões quantizadas (reduzidas para consumirem menos recursos computacionais), capazes de serem executadas em *hardware* convencional. Nesse cenário, a correlação com os custos operacionais passaria a

considerar o consumo de energia elétrica, em vez do custo por requisição às APIs comerciais, permitindo avaliar se tais modelos constituem alternativas economicamente vantajosas e de desempenho competitivo;

10. Priorizar, na integração com o ambiente virtual de aprendizagem Moodle institucional (Item 7), a execução local de um modelo aberto pequeno e atual (por exemplo, Qwen 3.5 9B), cenário em que se estimam resultados similares, ou até superiores, aos dos modelos aqui avaliados, com os benefícios adicionais da maior privacidade dos dados dos estudantes e da independência de provedores externos.

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.

ARAUJO, L. V. *et al.* Qual tipo de feedback ajuda os estudantes novatos em programação? Um estudo exploratório com múltiplos feedbacks em um curso introdutório de Python. *In*: **Anais do IV Simpósio Brasileiro de Educação em Computação (EduComp)**. São Paulo, Brasil: Sociedade Brasileira de Computação, 2024. p. 39–49.

ARIMOTO, M.; OLIVEIRA, W. Dificuldades no processo de aprendizagem de programação de computadores: um survey com estudantes de cursos da área de Computação. *In*: **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*: **Anais do XXXIV Simpósio Brasileiro de Informática na Educação**. Porto Alegre: Sociedade Brasileira de Computação, 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. **arXiv preprint arXiv:2312.11805**, 2023. Disponível em: https://arxiv.org/abs/2312.11805.

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

HOPPSCOTCH. **Hoppscotch: open source API development ecosystem**. 2026. Disponível em: https://hoppscotch.io.

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.

KOSMYNA, N. *et al.* **Your brain on ChatGPT: accumulation of cognitive debt when using an AI assistant for essay writing task**. 2025. ArXiv:2506.08872. Disponível em: https://arxiv.org/abs/2506.08872.

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 enhance programming error messages. *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. 563–569. ISBN 9781450394314. Disponível em: https://doi.org/10.1145/3545945.3569770.

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.

LOPES, A.; GARCIA, G. **Introdução à programação: 500 algoritmos resolvidos**. 15. ed. Rio de Janeiro: Elsevier, 2002. 488 p. ISBN 85-352-1019-9.

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.

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

OPENAI. **ChatGPT**. 2025. Disponível em: https://chatgpt.com/pt-BR/overview?openaicom_ referred=true.

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: Sociedade Brasileira de Computação, 2025. p. 590–602. 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.

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. **Misconceptions in correct code: assisting instructors and students by shedding light on what is potentially overshadowed by automated correction**. 2024. 197 p. Tese (Tese de Doutorado) — Universidade Estadual de Campinas (UNICAMP), Campinas, SP, 2024.

SILVA, E.; CACEFFO, R.; AZEVEDO, R. Análise dos tópicos mais abordados em disciplinas de introdução à programação em universidades federais brasileiras. *In*: **Anais do II Simpósio Brasileiro de Educação em Computação**. Porto Alegre: Sociedade Brasileira de Computação, 2022. p. 29–39. ISSN 3086-0733. Disponível em: https://sol.sbc.org.br/index.php/educomp/article/view/19196.

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: Sociedade Brasielira de Computação, 2023. p. 119–129. ISSN 3086-0733. Disponível em: https: //sol.sbc.org.br/index.php/educomp/article/view/23881.

SILVA, E.; CACEFFO, R.; AZEVEDO, R. Além da corretude: investigando os limites da correção automática ao analisar códigos corretos. *In*: **Anais Estendidos do V Simpósio Brasileiro de Educação em Computação**. Porto Alegre: Sociedade Brasielira de Computação, 2025. p.

91–95. ISSN 3086-0741. Disponível em: https://sol.sbc.org.br/index.php/educomp_estendido/article/view/34924.

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.

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.

APÊNDICES

APÊNDICE A – *PROMPT* UTILIZADO

Na sequência, é apresentado o *prompt* que foi criado para os testes com os modelos GPT-4, Claude, Gemini e DeepSeek.

```
<<<'PROMPT'
Você é um revisor especialista de código analisando exercícios TypeScript de alunos.
Classifique quaisquer erros de programação encontrados usando duas taxonomias complementares.
Todas as suas respostas devem ser em português do Brasil (pt-BR).

## TAXONOMIA PC (natureza do erro)

**Predicate**  Erros de lógica ou condição: comparações erradas, erros de off-by-one,
expressões booleanas incorretas. Exemplo: usar `>` em vez de `>=` em uma verificação de
limite, excluindo erroneamente o último elemento.

**Concept**  Conceito de linguagem ou API mal compreendido: método mal utilizado,
tipo incorreto, semntica de operador errada. Exemplo: chamar `.push()` em um array
readonly, ou usar `==` quando igualdade estrita (`===`) é necessária.

**Context**  Problemas de escopo, ambiente ou configuração: variável errada no escopo,
import ausente, toolchain mal configurado. Exemplo: referenciar uma variável antes de sua
declaração `let` (zona morta temporal), ou importar do caminho de módulo errado.

## CODIGO DE ERRO ESPECIFICO

Classifique também o erro em exatamente um dos códigos abaixo. Se nenhum se aplicar, use NONE.

- **B6**  Laço `while` usado onde uma única verificação booleana (`if`) era a intenção
- **B8**  Estrutura `if-else` incorreta (ramificações ausentes, ordem errada)
- **B9**  `else if` retesta condição já provada falsa por ramificação anterior
- **B12**  `if`s consecutivos com condições idênticas e corpos distintos ( deveria ser `if-else-if`)
```

```
- **C1**   Condição de guarda do `while` reverificada explicitamente dentro do corpo do laço
- **C3**   Operações dentro do laço invariantes à iteração (redundantes a cada ciclo)
- **C8**   Variável contadora do `for` sobrescrita dentro do corpo do laço
- **G3**   Múltiplas declarações de variáveis concentradas em uma única linha
- **G4**   Identificadores com nomes não descritivos (ex.: `a`, `x1`, `n`) sem significado semntico
- **H1**   Instruções sem efeito (valores calculados descartados, código inacessível)
- **NONE**  Nenhum dos padrões acima se aplica

Retorne um JSON com exatamente estes campos:
- `diagnosis`: descrição concisa do erro encontrado (em pt-BR)
- `pc3_category`: exatamente um de "Predicate", "Concept" ou "Context"
- `error_code`: exatamente um de "B6", "B8", "B9", "B12", "C1", "C3", "C8", "G3", "G4", "H1" ou "NONE"
- `feedback`: orientação prática para o aluno corrigir o erro (em pt-BR)
- `confidence`: sua confiança como float entre 0.0 e 1.0
- `tokens_input`: estimativa de tokens de entrada como inteiro
- `tokens_output`: estimativa de tokens de saída como inteiro
PROMPT;
```

APÊNDICE B – SUBCONJUNTO DE ERROS ADAPTADO À LINGUAGEM TYPESCRIPT

A adaptação do subconjunto de erros ocorreu conforme descrito nas seções 4.2.1 (Passo 4 da Fase 1) e 5.1. Os erros estão listados em ordem decrescente de gravidade. Cada erro foi detalhado nos seguintes termos: denominação original em Python (nos casos em que houve diferença), respectivas métricas (conforme Seção 3.1), descrição, intervenção para TypeScript e um exemplo de código incorreto (contendo aquele erro). A versão atualizada (e estendida) deste catálogo pode ser consultada on-line[1].

C8 – Laço *for* com variável responsável pela iteração sendo sobrescrita

Métricas

CT+C	N+B	DT+D	**DIF**
31	0	1	**30**

Descrição

A variável de controle do laço *for* é atualizada manualmente dentro do corpo da estrutura e isso causa um comportamento inesperado na iteração. Geralmente, ocorre por falta de entendimento ou desatenção, mas pode ser intencional.

Intervenção para TypeScript

Não alterar a variável de controle. Usar *while* ou *do-while*. Usar *for-of* ou *forEach()*.

Exemplo de código incorreto

```
1  for (let i = 0; i < arr.length; i++) {
2    i = i + 1;  // sobrescreve a variável de controle
3    console.log(arr[i]);
4  }
```

[1] Disponível em: cutt.ly/Bt8aUpBm.

B6 – Tentativa de comparação booleana feita com laço *while*

Métricas

CT+C	N+B	DT+D	**DIF**
26	4	2	**20**

Descrição

Um laço *while* é usado apenas para verificar uma condição que poderia ser testada diretamente com *if* ou por meio de uma expressão booleana. Pode estar relacionado à tentativa de aplicar um conceito recém-aprendido, mesmo quando não se faz necessário.

Intervenção para TypeScript

Substituir o *while* pela estrutura condicional *if*.

Exemplo de código incorreto

```
1  while (idade >= 18) {
2    console.log("Atingiu a maioridade");
3    break;  // executa apenas uma vez, WHILE é desnecessário
4  }
```

C1 – Condição *while* testada novamente no interior do bloco

Métricas

CT+C	N+B	DT+D	**DIF**
26	3	3	**20**

Descrição

Ao final do bloco de instruções de um *while*, inclui-se um *if* para verificar se a condição de parada tornou-se falsa, seguido de um *break*. Surge, em geral, do entendimento de que a condição do laço *while* deve ser novamente verificada no final do bloco de instruções.

Intervenção para TypeScript

Eliminar a estrutura condicional *if* que faz a retestagem.

Exemplo de código incorreto

```
1  while (count > 0) {
2    console.log(count);
3    count--;
4    if (count <= 0) {  // verificação redundante
5      break;
6    }
7  }
```

B8 – Não utilização correta da estrutura *if-else*

Denominação original em Python

Não utilização da estrutura *if-elif-else*.

Métricas

CT+C	N+B	DT+D	**DIF**
24	8	0	**16**

Descrição

Múltiplas estruturas condicionais *if* independentes são usadas quando as opções são exclusivas (e apenas um bloco é executado). Também pode ser um problema terminar o encadeamento com *else-if* ao invés de *else*. Indica dificuldade de identificar condições mutuamente exclusivas.

Intervenção para TypeScript

Reestruturar o trecho considerando estruturas condicionais *if* aninhadas ou *switch-case*.

Exemplo de código incorreto

```
1  if (x == 1)
2    extenso = "um";
3  if (x == 2)
4    extenso = "dois";  // deveria ser ELSE IF
5  if (x == 3)
6    extenso = "três";
```

G4 – Identificadores com nomes não significativos

Denominação original em Python

Variáveis/funções com nomes não significativos.

Métricas

CT+C	N+B	DT+D	**DIF**
24	7	1	**16**

Descrição

Identificadores com nomes genéricos (*a*, *b*, *x*, *temp*, *foo*) que não comunicam a intenção do código. A escolha de nomes significativos é negligenciada, uma vez que o código funciona com quaisquer que sejam os nomes utilizados.

Intervenção para TypeScript

Usar nomes significativos baseados no propósito de cada variável, método ou classe. Aplicar convenções de nomenclatura TypeScript (*camelCase*).

Exemplo de código incorreto

```
1  let x = a * (b**2);  // nenhum nome significativo foi utilizado
2  console.log(x);
```

H1 – Código sem efeito

Métricas

CT+C	N+B	DT+D	**DIF**
24	4	4	**16**

Descrição

Um trecho de código é escrito mas não produz efeito na execução do programa (e também não atribui o resultado a nada). É um código morto.

Intervenção para TypeScript

Revisar e remover linhas de código não utilizadas.

Exemplo de código incorreto

```
1  let nota1, nota2, soma, media;  // variável SOMA não utilizada
2  nota1 = teclado.questionInt();
3  nota2 = teclado.questionInt();
4  media = (nota1 + nota2) / 2;
5  Math.round(media);  // resultado não atribuído
6  console.log(media);
```

B12 – Consecutivas declarações de *if*'s iguais com operações distintas nos blocos

Métricas

CT+C	N+B	DT+D	**DIF**
23	5	4	**14**

Descrição

Duas ou mais instruções *if* testam a mesma condição, embora executem comandos diferentes em cada bloco. Uma possível causa é se pensar que cada *if* pode conter um número limitado de instruções.

Intervenção para TypeScript

Agrupar os blocos de código em uma única instrução *if*.

Exemplo de código incorreto

```
1  if (senhaDigitada != senha)
2    console.log("Senha incorreta!");
3  if (senhaDigitada != senha)  // testa a mesma condição anterior
4    tentativas++;
```

B9 – Instrução *else-if* retestando condições já verificadas

Denominação original em Python

elif/else retestando condição superior.

Métricas

CT+C	N+B	DT+D	**DIF**
23	4	5	**14**

Descrição

Um *else-if* verifica a condição oposta de um *if* ou *else-if* anterior. Geralmente, provém da falta de entendimento de que *if* e *else* são alternativas exclusivas. Outra possibilidade é querer usar verificações redundantes para assegurar o comportamento pretendido.

Intervenção para TypeScript

Remover as verificações redundantes.

Exemplo de código incorreto

```
1  if (idade < 18)
2    console.log("Não atingiu a maioridade");
3  else
4    if (idade >= 18)  // condição redundante
5      console.log("Já atingiu a maioridade");
```

C3 – Operações redundantes dentro de um laço

Métricas

CT+C	N+B	DT+D	**DIF**
21	9	2	**10**

Descrição

As operações estão sendo calculadas (ou instruções sendo executadas) desnecessariamente dentro de um laço. Acontece quando não se percebe as possibilidades de refatoração do código e isso aumenta a ineficiência da solução.

Intervenção para TypeScript

Identificar e remover as instruções redundantes.

Exemplo de código incorreto

```
1  for (i = 0; i < 10; i++) {
2    soma += this.m[i];
3    media = soma / 10;  // a média é repetidamente calculada
4  }
```

G3 – Muitas declarações em uma única linha de código

Métricas

CT+C	N+B	DT+D	**DIF**
21	7	4	**10**

Descrição

Várias operações, distintas ou não, são declaradas na mesma linha de código. Isso prejudica a legibilidade e a depuração. Acontece devido a preferências de estilo na escrita do código. Muitos optam por escrever assim por acreditarem que isso otimiza o consumo de recursos computacionais.

Intervenção para TypeScript

Reestruturar para uma instrução por linha. Basear-se em recursos como *Prettier* e *ESLint*.

Exemplo de código incorreto

```
1  for (i=0; i<10; i++) if (this.m[i] % 2 == 0); soma += this.m[i];
2  // deveria estar estruturado em três linhas
```

APÊNDICE C – CONJUNTO PADRONIZADO (*CORPUS*) DE CÓDIGOS DE TESTE

A composição do conjunto padronizado (*corpus*) de teste foi feita conforme descrito nas seções 4.2.1 (Passo 4 da Fase 1) e 5.2. Cada entrada contém o enunciado do exercício seguido do respectivo código-fonte da tentativa de solução em TypeScript.

Em caráter de exemplo, seguem as cinco instâncias de teste para o erro B9 (instrução *if-else* retestando condições já verificadas). O identificador de cada código de teste é constituído obedecendo o formato: *[Erro] – [Instância de Teste] _ [Enunciado]*. Portanto, o Código de Teste *B9–1_097* trata-se do erro B9, instância de teste 1 (de 5) e diz respeito ao enunciado 97 do livro *Introdução à Programação: 500 Algoritmos Resolvidos* (LOPES; GARCIA, 2002).

O conjunto completo de códigos de teste pode ser acessado no repositório público do projeto[1] (vide pasta *corpusTeste*).

Código de Teste B9–1_097

Enunciado: Entrar com um número e informar se ele é divisível por 10, por 5, por 2 ou se não é divisível por nenhum destes.

```
1  import teclado from "npm:readline-sync";
2
3  let numero: number = 0;
4
5  console.log("\ndigite numero: ");
6  numero = teclado.questionInt();
7
8  if (numero % 10 === 0) {
9    console.log("\nmúltiplo de 10");
10 } else if (numero % 10 !== 0 && numero % 2 === 0) {
11   console.log("\nmúltiplo de 2");
12 } else if (numero % 10 !== 0 && numero % 2 !== 0 && numero % 5 === 0) {
13   console.log("\nmúltiplo de 5");
14 } else {
15   console.log("\nnão é múltiplo de 2 nem de 5");
16 }
17
18 console.log("\n");
```

[1] Disponível em: github.com/pedrozampier/pc3-submission-evaluator

Código de Teste B9–2_099

Enunciado: Ler um número inteiro de 3 casas decimais e imprimir se o algarismo da casa das centenas é par ou ímpar.

```
1  import teclado from "npm:readline-sync";
2
3  let num: number = 0,
4      c: number = 0;
5
6  console.log("\número de 3 algarismos: ");
7  num = teclado.questionInt();
8
9  c = Math.trunc(num / 100);
10
11 if (c % 2 === 0) {
12   console.log("\no algarismo das centenas e par: ", c);
13 } else if (c % 2 !== 0) {
14   console.log("\no algarismo das centenas é impar: ", c);
15 }
16
17 console.log("\n");
```

Código de Teste B9–3_102

Enunciado: Entrar com um número e imprimir uma das mensagens: maior do que 20, igual a 20 ou menor do que 20.

```
1  import teclado from "npm:readline-sync";
2
3  let numero: number = 0;
4
5  console.log("\ndigite numero: ");
6  numero = teclado.questionFloat();
7
8  if (numero > 20) {
9    console.log("\nmaior que 20");
10 } else if (numero <= 20 && numero < 20) {
11   console.log("\nmenor que 20");
12 }
13 if (numero === 20) {
14   console.log("\nigual a 20");
15 }
16
17 console.log("\n");
```

Código de Teste B9–4_103

Enunciado: Entrar com o ano de nascimento de uma pessoa e o ano atual. Imprimir a idade da pessoa. Não se esqueça de verificar se o ano de nascimento é um ano válido.

```
1  import teclado from "npm:readline-sync";
2
3  let anon: number = 0,
4      anoa: number = 0;
5
6  console.log("\nEntre com ano atual: ");
7  anoa = teclado.questionInt();
8  console.log("\nEntre com ano de nascimento: ");
9  anon = teclado.questionInt();
10
11 if (anon > anoa) {
12   console.log("\nAno de Nascimento Invalido");
13 } else if (anon <= anoa) {
14   console.log("\nIdade: ", anoa - anon);
15 }
16 if (anon === anoa) {
17   console.log("\nIdade: 0");
18 }
19
20 console.log("\n");
```

Código de Teste B9–5_105

Enunciado: Entrar com a sigla do estado de uma pessoa e imprimir uma das mensagens: carioca, paulista, mineiro, outros estados.

```
1  import teclado from "npm:readline-sync";
2
3  let sigla: string = "";
4
5  console.log("\ndigite sigla: ");
6  sigla = teclado.question();
7
8  if (sigla === "RJ" || sigla === "rj") {
9    console.log("\ncarioca");
10 } else if (sigla !== "RJ" && sigla !== "rj" && (sigla === "SP" || sigla ===
       "sp")) {
11   console.log("\npaulista");
12 } else if (sigla !== "RJ" && sigla !== "rj" && sigla !== "SP" && sigla !== "
       sp" && (sigla === "MG" || sigla === "mg")) {
13   console.log("\nmineiro");
14 }
15 if (sigla !== "RJ" && sigla !== "rj") {
16   if (sigla !== "SP" && sigla !== "sp") {
17     if (sigla !== "MG" && sigla !== "mg") {
18       console.log("\noutros estados");
19     }
20   }
21 }
22
23 console.log("\n");
```

APÊNDICE D – EXEMPLO DE REQUISIÇÃO E DE RETORNO DO PROTÓTIPO DE MVP

Apresenta-se um exemplo de requisição do código de teste *B9–1_097* (primeiro código do Apêndice C), no formato JSON, para o protótipo de MVP. A requisição contém o código-fonte (*code*) e o enunciado do exercício (*statement*). Na sequência, é mostrado o retorno fornecido pelo protótipo para cada modelo considerado (Claude, GPT-4, Gemini e DeepSeek).

Entrada

```
{
  "code": "import teclado from \"npm:readline-sync\";\n\nlet numero: number
    = 0;\n\nconsole.log(\"\\ndigite numero: \");\nnumero = teclado.
    questionInt();\n\nif (numero % 10 === 0) {\n  console.log(\"\\nmúltiplo
    de 10\");\n} else if (numero % 10 !== 0 && numero % 2 === 0) {\n
    console.log(\"\\nmúltiplo de 2\");\n} else if (numero % 10 !== 0 &&
    numero % 2 !== 0 && numero % 5 === 0) {\n  console.log(\"\\nmúltiplo de
    5\");\n} else {\n  console.log(\"\\nnão é múltiplo de 2 nem de 5\");\n}\
    n\nconsole.log(\"\\n\");",
  "statement": "Entrar com um número e informar se ele é divisível por 10,
    por 5, por 2 ou se não é divisível por nenhum destes."
}
```

Retorno fornecido pelo Claude


```
{
  "provider": "anthropic",
  "model": "claude-sonnet-4-6",
  "diagnosis": "Os `else if` retestarão condições já provadas falsas pelas
    ramificações anteriores. Por exemplo, o segundo `else if` verifica `
    numero % 10 !== 0` novamente, mas essa condição já é garantidamente
    verdadeira por ter chegado até ali (o `if` anterior só é pulado quando `
    numero % 10 === 0` é falso). O mesmo ocorre no terceiro `else if`, que
    retesta tanto `numero % 10 !== 0` quanto `numero % 2 !== 0`, condições
    ambas já descartadas pelas ramificações anteriores.",
  "pc3_category": "Predicate",
  "feedback": "Numa cadeia `if / else if / else`, cada `else if` só é alcanç
    ado se **todas** as condições anteriores já foram avaliadas como falsas
     portanto, retestar essas mesmas condições é redundante e polui o código
    . Simplifique removendo as verificações desnecessárias: o segundo ramo j
    á sabe que `numero % 10 !== 0`, e o terceiro já sabe que `numero % 10
    !== 0` e `numero % 2 !== 0`. O código corrigido ficaria assim:\n\n```
    typescript\nif (numero % 10 === 0) {\n  console.log(\"\\nmúltiplo de
    10\");\n} else if (numero % 2 === 0) {       // % 10 !== 0 já garantido\
    n  console.log(\"\\nmúltiplo de 2\");\n} else if (numero % 5 === 0) {
        // % 10 !== 0 e % 2 !== 0 já garantidos\n  console.log(\"\\nmúltiplo
     de 5\");\n} else {\n  console.log(\"\\nnão é múltiplo de 2 nem de 5\")
    ;\n}\n```\n\nAlém disso, vale observar que **todo múltiplo de 10 também
    é múltiplo de 5 e de 2**, então a ordem das verificações (10  2  5) está
     correta para reportar o divisor mais específico primeiro.",
  "confidence": 0.97,
  "tokens_input": 780,
  "tokens_output": 340,
  "request_id": "d856d9b5-0048-44a0-9597-a2e662cbac59",
  "prompt_version": "v2.1"
}
```


Diagnóstico

Os *else if* retestarão condições já provadas falsas pelas ramificações anteriores. Por exemplo, o segundo *else if* verifica *numero % 10 !== 0* novamente, mas essa condição já é garantidamente verdadeira por ter chegado até ali (o *if* anterior só é pulado quando *numero % 10 === 0* é falso). O mesmo ocorre no terceiro *else if*, que retesta tanto *numero % 10 !== 0* quanto *numero % 2 !== 0*, condições ambas já descartadas pelas ramificações anteriores.

Feedback

Numa cadeia *if / else if / else*, cada *else if* só é alcançado se **todas** as condições anteriores já foram avaliadas como falsas — portanto, retestar essas mesmas condições é redundante e polui o código. Simplifique removendo as verificações desnecessárias: o segundo ramo já sabe que *numero % 10 !== 0*, e o terceiro já sabe que *numero % 10 !== 0* e *numero % 2 !== 0*. O código corrigido ficaria assim:

```
if (numero % 10 === 0) {
  console.log("\nmúltiplo de 10");
} else if (numero % 2 === 0) {       // % 10 !== 0 já garantido
  console.log("\nmúltiplo de 2");
} else if (numero % 5 === 0) {       // % 10 !== 0 e % 2 !== 0 já garantidos
  console.log("\nmúltiplo de 5");
} else {
  console.log("\nnão é múltiplo de 2 nem de 5");
}
```

Além disso, vale observar que **todo múltiplo de 10 também é múltiplo de 5 e de 2**, então a ordem das verificações (10 $\rightarrow$ 2 $\rightarrow$ 5) está correta para reportar o divisor mais específico primeiro.

Retorno fornecido pelo GPT-4


```
{
  "provider": "openai",
  "model": "gpt-4o",
  "diagnosis": "Condição `else if` verifica repetidamente se número não é múltiplo de 10, o que é implícito nas condições subsequentes.",
  "pc3_category": "Predicate",
  "feedback": "Remova as verificações redundantes de não multiplicidade em 10 nos `else if`, pois elas já são implicações lógicas das condições anteriores. Isso simplificará as verificações.",
  "confidence": 1,
  "tokens_input": 260,
  "tokens_output": 95,
  "request_id": "d856d9b5-0048-44a0-9597-a2e662cbac59",
  "prompt_version": "v2.1"
}
```


Diagnóstico

Condição *else if* verifica repetidamente se número não é múltiplo de 10, o que é implícito nas condições subsequentes.

Feedback

Remova as verificações redundantes de não multiplicidade em 10 nos *else if*, pois elas já são implicações lógicas das condições anteriores. Isso simplificará as verificações.

Retorno fornecido pelo Gemini


```
{
  "provider": "gemini",
  "model": "gemini-3.5-flash",
  "diagnosis": "O código faz verificações redundantes em blocos 'else if'
    subsequentes, testando novamente condições que já foram comprovadamente
    falsas para que o fluxo de execução pudesse chegar àquele ponto.",
  "pc3_category": "Predicate",
  "feedback": "Em uma estrutura 'if-else if', o bloco 'else if' só é
    avaliado se a condição anterior for falsa. Assim, se o código chegou no
    primeiro 'else if', já é garantido que o número não é divisível por 10.
    Você pode remover as verificações redundantes como 'numero % 10 !== 0'
    para tornar o seu código mais limpo e legível.",
  "confidence": 0.95,
  "tokens_input": 450,
  "tokens_output": 250,
  "request_id": "d856d9b5-0048-44a0-9597-a2e662cbac59",
  "prompt_version": "v2.1"
}
```


Diagnóstico

O código faz verificações redundantes em blocos *else if* subsequentes, testando novamente condições que já foram comprovadamente falsas para que o fluxo de execução pudesse chegar àquele ponto.

Feedback

Em uma estrutura *if-else if*, o bloco *else if* só é avaliado se a condição anterior for falsa. Assim, se o código chegou no primeiro *else if*, já é garantido que o número não é divisível por 10. Você pode remover as verificações redundantes como *numero % 10 !== 0* para tornar o seu código mais limpo e legível.

Retorno fornecido pelo DeepSeek


```
{
  "provider": "deepseek",
  "model": "deepseek-chat",
  "diagnosis": "A condição do segundo else if verifica se o número não é
    divisível por 10 e é divisível por 2, mas isso exclui números divisíveis
     por 10 (que já são divisíveis por 2). O correto é verificar apenas a
    divisibilidade por 2, sem a negação. Além disso, a mensagem final 'não é
     múltiplo de 2 nem de 5' está incorreta, pois números divisíveis por 10
    não são considerados múltiplos de 5 nessa lógica.",
  "pc3_category": "Predicate",
  "feedback": "Reveja a lógica das condições. A primeira condição já captura
     números divisíveis por 10. No segundo else if, basta verificar se o nú
    mero é divisível por 2, pois se for divisível por 10, já foi capturado.
    No terceiro else if, verifique apenas se é divisível por 5. Ajuste també
    m a mensagem final para incluir '10' ou reestruture as condições.",
  "confidence": 0.9,
  "tokens_input": 298,
  "tokens_output": 166,
  "request_id": "d856d9b5-0048-44a0-9597-a2e662cbac59",
  "prompt_version": "v2.1"
}
```


Diagnóstico

A condição do segundo *else if* verifica se o número não é divisível por 10 e é divisível por 2, mas isso exclui números divisíveis por 10 (que já são divisíveis por 2). O correto é verificar apenas a divisibilidade por 2, sem a negação. Além disso, a mensagem final *'não é múltiplo de 2 nem de 5'* está incorreta, pois números divisíveis por 10 não são considerados múltiplos de 5 nessa lógica.

Feedback

Reveja a lógica das condições. A primeira condição já captura números divisíveis por 10. No segundo *else if*, basta verificar se o número é divisível por 2, pois se for divisível por 10, já foi capturado. No terceiro *else if*, verifique apenas se é divisível por 5. Ajuste também a mensagem final para incluir *'10'* ou reestruture as condições.