



**INSTITUTO  
FEDERAL**  
Paraíba

**Instituto Federal de Educação, Ciência e Tecnologia da Paraíba**

**Campus João Pessoa**

**Programa de Pós-Graduação em Tecnologia da Informação**

**Nível Mestrado Profissional**

**CRISTOFER GIRARDI**

**EXPLORANDO COMBINAÇÕES DE TÉCNICAS DE PROMPT  
ENGINEERING: UM CASO DE USO NA GERAÇÃO E EXPLICAÇÃO  
DE CÓDIGO**

**DISSERTAÇÃO DE MESTRADO**

**JOÃO PESSOA**

**2026**

**Cristofer Girardi**

**Explorando combinações de técnicas de Prompt Engineering:  
Um caso de uso na geração e explicação de código**

Dissertação apresentada como requisito parcial para obtenção do título de Mestre em Tecnologia da Informação, pelo Programa de Pós-Graduação em Tecnologia da Informação do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba – IFPB.

Orientador: Prof. Dr. Damires Yluska de Souza  
Fernandes  
Coorientador: Prof. Dr. Alex Sandro da Cunha  
Rêgo

João Pessoa

2026

Dados Internacionais de Catalogação na Publicação (CIP)  
Biblioteca Nilo Peçanha - *campus* João Pessoa, PB.

G521e Girardi, Cristofer.

Explorando combinações de técnicas de *Prompt engineering* :  
um caso de uso na geração e explicação de código / Cristofer  
Girardi. – 2026.

173 f. : il.

Dissertação (Mestrado em Tecnologia da Informação) – Instituto Federal de Educação da Paraíba / Programa de Pós-Graduação em Tecnologia da Informação (PPGTI), 2026.

Orientação: Profª Dra. Damires Yluska de Souza Fernandes.

Coorientação : Prof. Dr. Alex Sandro da Cunha Rêgo.

1. Engenharia de *Prompt*. 2. Geração de código. 3. Explicação de código 4. Combinação de técnicas de *Prompts*. 5. LLMs. I. Título.

CDU 004.8(043)

**CRISTOFER GIRARD**

**EXPLORANDO COMBINAÇÕES DE TÉCNICAS DE PROMPT ENGINEERING: UM CASO DE USO NA  
GERAÇÃO E EXPLICAÇÃO DE CÓDIGO**

**DISSERTAÇÃO** apresentada como requisito para obtenção do título de Mestre em Tecnologia da Informação, pelo Programa de Pós-Graduação em Tecnologia da Informação do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - IFPB - Campus João Pessoa.

Aprovado em 04 de março de 2026.

**Membros da Banca Examinadora:**

**Profa. Dra. Damires Yluska de Souza Fernandes**

Instituto Federal da Paraíba (IFPB)

*Orientadora*

**Prof. Dr. Alex Sandro da Cunha Rego**

Instituto Federal da Paraíba (IFPB)

*Coorientador*

**Prof. Dr. Tiago Brasileiro Araújo**

Instituto Federal da Paraíba (IFPB)

*Examinador*

**Prof. Dr. Claudio Elizio Calazans Campelo**

Universidade Federal de Campina Grande (UFCG)

*Examinador*

Documento assinado eletronicamente por:

- **Damires Yluska de Souza Fernandes**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 05/03/2026 09:44:28.
- **Claudio Elizio Calazans Campelo**, PRESTADOR DE SERVIÇO, em 05/03/2026 10:48:58.
- **Tiago Brasileiro Araujo**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 05/03/2026 13:46:48.
- **Alex Sandro da Cunha Rego**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 05/03/2026 15:03:13.

Este documento foi emitido pelo SUAP em 20/02/2026. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse <https://suap.ifpb.edu.br/autenticar-documento/> e forneça os dados abaixo:

Código 837616

Verificador: 4b3f82d304

Código de Autenticação:



# AGRADECIMENTOS

A você, Janaina, meu passado, presente e futuro, que esteve ao meu lado com a força de quem sabe sustentar o mundo com cuidado — meu agradecimento mais profundo.

Esta dissertação não é só um marco acadêmico; é também a expressão do nosso caminhar conjunto, dos afetos que nos tecem. Como a psicóloga, escritora e ativista indígena Geni Núñez nos ensina, amar não é só sentir: é escolher, é cuidar, é resistir.

Você me ofereceu abrigo nos dias difíceis, escuta nos momentos de dúvida e impulso quando a coragem ameaçava faltar. A sua presença foi o chão firme em que pude construir, a mão estendida quando o peso era demais.

Com você, aprendi que o amor pode — e deve — ser um espaço de liberdade, afeto e reconstrução. E que a caminhada, quando feita em parceria, é sempre mais leve, mais justa, mais inteira.

A você, minha companheira de jornada e de sentido, minha eterna gratidão.

# RESUMO

O uso de Modelos de Linguagem de Grande Escala (LLMs) na geração e explicação de código tem ampliado a relevância da Engenharia de *Prompts* (PE) como alternativa para aprimorar resultados em problemas de programação sem a necessidade de retreinamento dos modelos. Neste cenário, entretanto, ainda existem lacunas na compreensão dos efeitos das combinações de técnicas de PE sobre o comportamento dos LLMs, especialmente quando consideradas diferentes arquiteturas de modelos e estratégias de avaliação automatizadas. Este trabalho investiga, por meio de uma abordagem experimental controlada e operacionalizada por um *framework* reproduzível e escalável, o impacto dessas combinações na geração e explicação de código em Python. A pesquisa é organizada em três experimentos, nos quais são analisados treze cenários distintos, combinando seis técnicas de PE aplicadas a seis modelos de linguagem. A avaliação é conduzida com base na métrica BERTScore, na tarefa de geração de código, e em julgamentos automatizados por meio do paradigma *LLM-as-a-Judge*, guiado por critérios estruturados voltados à análise conjunta da geração e explicação de código. Os resultados indicam que as combinações de técnicas de PE influenciam significativamente a qualidade das respostas, em termos de similaridade semântica, coerência entre código e explicação, e custo computacional. Combinações que incorporam mecanismos de verificação tendem a aumentar a consistência entre código e explicação, enquanto configurações excessivamente complexas ou sem estratégias de refinamento podem comprometer a clareza das respostas. Este trabalho fornece evidências empíricas sobre o impacto dessas combinações e um *framework* experimental reutilizável, além de diretrizes práticas para sua aplicação em diferentes LLMs.

**Palavras-chave:** Engenharia de *Prompt*, Geração de Código, Explicação de Código, Combinações de Técnicas de *Prompts*, LLMs, LaaJ.

# ABSTRACT

The use of Large Language Models (LLMs) for code generation and explanation has increased the relevance of Prompt Engineering (PE) as an alternative approach to improving performance in programming tasks without requiring model retraining. However, important gaps remain in understanding the effects of combining PE techniques on LLM behavior, particularly when considering different model architectures and automated evaluation strategies. This work investigates, through a controlled experimental approach operationalized by a reproducible and scalable framework, the impact of such combinations on Python code generation and explanation. The study is structured into three experiments, analyzing thirteen distinct scenarios that combine six PE techniques applied to six language models. Evaluation is conducted using the BERTScore metric for code generation and automated judgments based on the LLM-as-a-Judge paradigm, guided by structured criteria designed to jointly assess code generation and explanation. The results indicate that combinations of PE techniques significantly influence response quality in terms of semantic similarity, coherence between code and explanation, and computational cost. Combinations incorporating verification mechanisms tend to improve consistency between code and explanation, whereas overly complex configurations or those lacking refinement strategies may reduce clarity. This work provides empirical evidence on the impact of these combinations, introduces a reusable experimental framework, and offers practical guidelines for their application across different LLMs.

**Key-words:** Prompt Engineering, Code Generation, Code Explanation, Prompt Technique Combinations, LLMs, LaaJ.

## LISTA DE FIGURAS

Figura 1	– Exemplo da técnica de <i>Role Prompting</i> . . . . .	34
Figura 2	– Exemplo de aplicação de <i>Role Prompting</i> . . . . .	35
Figura 3	– Exemplo de RaR para programação em Python . . . . .	36
Figura 4	– Exemplo de técnica de ZS . . . . .	36
Figura 5	– Exemplo de técnica de CoT . . . . .	37
Figura 6	– Exemplo de técnica de ZS-CoT . . . . .	38
Figura 7	– Exemplo de desenvolvimento do SoT . . . . .	39
Figura 8	– Exemplo de comportamento da técnica SV . . . . .	40
Figura 9	– Exemplo de <i>workflow</i> do RAG . . . . .	41
Figura 10	– Visão integrada do fluxo metodológico da pesquisa . . . . .	58
Figura 11	– <i>Workflow</i> do Experimento 1 . . . . .	77
Figura 12	– <i>Workflow</i> do <i>Framework</i> do Experimento 2 . . . . .	79
Figura 13	– <i>Workflow</i> do <i>Framework</i> do Experimento 3 . . . . .	80
Figura 14	– BERTScore - Comparação por Cenários x Modelos . . . . .	101
Figura 15	– BERTScore - Média de Valores por Cenários x Modelos . . . . .	105
Figura 16	– BERTScore - Avaliação por Cenários x Modelos . . . . .	106
Figura 17	– BERTScore - Tempo de Execução em (s) - Cenários x Modelos . . . . .	107
Figura 18	– Claude LaaJ - Média de Valores por Cenários x Modelos . . . . .	110
Figura 19	– Claude LaaJ - Avaliação por Cenários x Modelos . . . . .	112
Figura 20	– Claude LaaJ - Tempo de Execução em (s) - Cenários x Modelos . . . . .	114
Figura 21	– Gemini LaaJ - Média de Valores por Cenários x Modelos . . . . .	118
Figura 22	– Gemini LaaJ - Avaliação por Cenários x Modelos . . . . .	120
Figura 23	– Gemini LaaJ - Tempo de Execução em (s) - Cenários x Modelos . . . . .	122
Figura 24	– <i>Workflow</i> do cenário S <sub>5</sub> . . . . .	149
Figura 25	– <i>Workflow</i> do cenário S <sub>6</sub> . . . . .	150
Figura 26	– <i>Workflow</i> do cenário S <sub>7</sub> . . . . .	151
Figura 27	– <i>Workflow</i> do cenário S <sub>8</sub> . . . . .	152
Figura 28	– <i>Workflow</i> do cenário S <sub>9</sub> . . . . .	153
Figura 29	– <i>Workflow</i> do cenário S <sub>10</sub> . . . . .	153
Figura 30	– <i>Workflow</i> do cenário S <sub>11</sub> . . . . .	154
Figura 31	– <i>Workflow</i> do cenário S <sub>12</sub> . . . . .	154
Figura 32	– OpenAI . . . . .	155
Figura 33	– LLaMA . . . . .	155

## LISTA DE TABELAS

Tabela 1 – Tabela adaptada de (JI et al., 2023) . . . . .	51
Tabela 2 – Tabela comparativa do trabalhos relacionados . . . . .	57
Tabela 3 – Tabela das Técnicas de PE e Cenários de Combinações . . . . .	61
Tabela 4 – Critérios de Avaliação dos LaaJ . . . . .	67
Tabela 5 – Entidades selecionadas . . . . .	81
Tabela 6 – Dicionário de dados . . . . .	82
Tabela 7 – Estrutura relacional da infraestrutura de dados experimental . . . . .	83
Tabela 8 – Configuração dos Experimentos . . . . .	98

## LISTA DE ABREVIATURAS E SIGLAS

ATE	Average Treatment Effect
CoT	Chain-of-Thought
IA	Inteligência Artificial
LAAJ	LLM-as-a-Judge
LLM	Large Language Model
MLP	Multi Layer Perceptron
NLM	Neural Language Model
NLP	Natural Language Processing
PE	Prompt Engineering
RaR	Rephrase and Respond
RAG	Retrieval Augmented Generation
RNN	Recurrent Neural Network
RLHF	Reinforcement Learning with Human Feedback
SLM	Modelos Estatísticos de Linguagem
SoT	Skeleton-of-Thoughts
SV	Self-Verification
ZS-CoT	Zero-Shot-CoT

# LISTA DE SÍMBOLOS

$\hat{\mathbf{x}}^e$	Vetor médio estimado de $\mathbf{x}$ no espaço $e$
$\hat{\mathbf{x}}_i^e$	Estimativa do vetor $\mathbf{x}$ do exemplo $i$ , no espaço $e$
$N$	Número total de amostras de $\mathbf{x}$
$\sum_{i=1}^N$	Somatório das $N$ amostras de $\hat{\mathbf{x}}_i^e$
$\hat{\mathbf{y}}^e$	Vetor médio estimado de $\mathbf{y}$ no espaço $e$
$\hat{\mathbf{y}}_j^e$	Estimativa do vetor $\mathbf{y}$ do exemplo $j$ , no espaço $e$
$M$	Número total de amostras de $\mathbf{y}$
$\sum_{j=1}^M$	Somatório das $M$ amostras de $\hat{\mathbf{y}}_j^e$
$\frac{1}{N}, \frac{1}{M}$	Fator de normalização (média aritmética)
$\hat{x}^e$	Vetor médio de embeddings do texto gerado no espaço $e$
$\hat{y}^e$	Vetor médio de embeddings do texto de referência no espaço $e$
$\hat{x}^e \cdot \hat{y}^e$	Produto escalar entre os vetores $\hat{x}^e$ e $\hat{y}^e$
$\ \hat{x}^e\ $	Norma (magnitude) do vetor $\hat{x}^e$
$\ \hat{y}^e\ $	Norma (magnitude) do vetor $\hat{y}^e$
$\mu_{S_0}$	Média dos resultados do cenário de referência
$\mu_{S_i}$	Média dos resultados do cenário $S_i$ (onde $i \in [1, 12]$ )
$\mu_{S_0} = \mu_{S_i}$	Igualdade entre as médias (nenhuma diferença estatística)
$\mu_{S_0} \neq \mu_{S_i}$	Diferença entre as médias (diferença estatisticamente significativa)
$O(n!)$	Notação assintótica de complexidade fatorial

# SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO . . . . .</b>	<b>15</b>
<b>1.1</b>	<b>Motivação e Definição do Problema . . . . .</b>	<b>15</b>
<b>1.2</b>	<b>Objetivos . . . . .</b>	<b>19</b>
1.2.1	Objetivo geral . . . . .	19
1.2.2	Objetivos específicos . . . . .	19
<b>1.3</b>	<b>Contribuições . . . . .</b>	<b>20</b>
<b>1.4</b>	<b>Estrutura do Documento . . . . .</b>	<b>20</b>
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA . . . . .</b>	<b>21</b>
<b>2.1</b>	<b>Modelos de Linguagem . . . . .</b>	<b>21</b>
<b>2.2</b>	<b>Uso de LLM para geração e explicação de código . . . . .</b>	<b>23</b>
2.2.1	Geração . . . . .	23
2.2.2	Explicação . . . . .	24
2.2.3	Avaliação . . . . .	26
2.2.3.1	Métricas . . . . .	26
2.2.3.2	LLM-as-a-Judge . . . . .	29
<b>2.3</b>	<b>Técnicas de engenharia de <i>prompts</i> . . . . .</b>	<b>33</b>
2.3.1	<i>Role Prompting</i> . . . . .	34
2.3.2	<i>Rephrase and Respond (RaR)</i> . . . . .	35
2.3.3	<i>Zero-Shot-CoT (ZS-CoT)</i> . . . . .	36
2.3.4	<i>Skeleton-of-Thought (SoT)</i> . . . . .	38
2.3.5	<i>Self-Verification (SV)</i> . . . . .	39
2.3.6	<i>Retrieval Augmented Generation (RAG)</i> . . . . .	41
<b>3</b>	<b>TRABALHOS RELACIONADOS . . . . .</b>	<b>44</b>
<b>3.1</b>	<b>Aprimorando o Ensino de Programação de Computadores com LLMs: Um Estudo sobre Engenharia de Prompts Eficaz para Geração de Código em Python . . . . .</b>	<b>44</b>
<b>3.2</b>	<b>Avaliação da Performance e Corretude na Geração de Código Através de Técnicas de Engenharia de Prompt: Um Estudo Comparativo . . . . .</b>	<b>45</b>
<b>3.3</b>	<b>O que faz os Grandes Modelos de Linguagem raciocinarem na geração de código (em múltiplas interações)? . . . . .</b>	<b>46</b>
<b>3.4</b>	<b>O Comportamento de Grandes Modelos de Linguagem Quando São Induzidos a Gerar Explicações de Código . . . . .</b>	<b>48</b>

<b>3.5</b>	<b>Avaliação Comparativa e Explicação da Geração de Código Baseada em Grandes Modelos de Linguagem: Uma Abordagem Centrada na Causalidade</b> . . . . .	<b>50</b>
<b>3.6</b>	<b>Explicação de exemplos de código em cursos introdutórios de programação: LLMs versus humanos</b> . . . . .	<b>52</b>
<b>3.7</b>	<b>O Impacto da Programação de Prompts na Geração de Código em Nível de Função</b> . . . . .	<b>53</b>
<b>3.8</b>	<b>Comparativo e Diferenciais deste Trabalho</b> . . . . .	<b>54</b>
<b>4</b>	<b>METODOLOGIA</b> . . . . .	<b>58</b>
<b>4.1</b>	<b>Visão Geral da Abordagem Metodológica</b> . . . . .	<b>58</b>
<b>4.2</b>	<b>Definições e Proposta</b> . . . . .	<b>59</b>
4.2.1	Exploração e Seleção de Técnicas de PE . . . . .	59
4.2.2	Combinações de Técnicas de PE . . . . .	60
4.2.3	Avaliação da Geração e Explicação de Código . . . . .	63
4.2.4	Seleção dos LLMs . . . . .	70
4.2.5	<i>Framework</i> . . . . .	71
<b>4.3</b>	<b>Desenho Experimental</b> . . . . .	<b>72</b>
4.3.1	Questões de Pesquisa e Objetivos dos Experimentos . . . . .	72
4.3.2	<i>Framework</i> Experimental . . . . .	73
4.3.3	Infraestrutura de Dados Experimental . . . . .	80
4.3.4	Cenários Experimentais . . . . .	83
4.3.5	Avaliação BERTScore e LaaJ . . . . .	95
<b>4.4</b>	<b>Execução Experimental</b> . . . . .	<b>97</b>
<b>5</b>	<b>RESULTADOS E DISCUSSÕES</b> . . . . .	<b>100</b>
<b>5.1</b>	<b>Experimento 1</b> . . . . .	<b>100</b>
<b>5.2</b>	<b>Experimento 2</b> . . . . .	<b>104</b>
<b>5.3</b>	<b>Experimento 3</b> . . . . .	<b>109</b>
5.3.1	Experimento 3.1 . . . . .	110
5.3.2	Experimento 3.2 . . . . .	117
5.3.3	Comparativo entre os Avaliadores LaaJ . . . . .	124
<b>5.4</b>	<b>Ameaças à Validade</b> . . . . .	<b>128</b>
<b>6</b>	<b>CONSIDERAÇÕES FINAIS</b> . . . . .	<b>130</b>
<b>6.1</b>	<b>Contribuições</b> . . . . .	<b>131</b>
<b>6.2</b>	<b>Limitações</b> . . . . .	<b>133</b>
<b>6.3</b>	<b>Trabalhos Futuros</b> . . . . .	<b>134</b>

<b>REFERÊNCIAS BIBLIOGRÁFICAS . . . . .</b>	<b>135</b>
<b>APÊNDICES</b>	<b>144</b>
<b>APÊNDICE A – EXEMPLO DE CÓDIGO E EXPLICAÇÃO NA LINGUAGEM JAVA . . . . .</b>	<b>145</b>
<b>APÊNDICE B – IMPLEMENTAÇÃO DA CAMADA DE INTEGRAÇÃO COM APIS DE LLMS . . . . .</b>	<b>147</b>
<b>APÊNDICE C – <i>WORKFLOWS</i> DOS CENÁRIOS . . . . .</b>	<b>149</b>
<b>APÊNDICE D – GRÁFICOS DE NORMALIDADE DE DISTRIBUIÇÃO DOS VALORES DE BERTSCORE . . . . .</b>	<b>155</b>
<b>APÊNDICE E – EXEMPLO DE RESULTADO DO EXPERIMENTO 2</b>	<b>156</b>
<b>APÊNDICE F – EXEMPLOS DE RESULTADOS DA AVALIAÇÃO DO CLAUDE LAAJ . . . . .</b>	<b>157</b>
<b>APÊNDICE G – EXEMPLOS DE RESULTADOS DA AVALIAÇÃO DO GEMINI LAAJ . . . . .</b>	<b>162</b>

# 1 INTRODUÇÃO

Este capítulo apresenta o contexto no qual esta pesquisa está inserida, situando o problema investigado. Para tanto, são delineadas a motivação, as questões de pesquisa, os objetivos do trabalho, as contribuições e a estrutura do documento.

## 1.1 Motivação e Definição do Problema

A Inteligência Artificial (IA) ganhou mais notoriedade depois que o ChatGPT da OpenAI<sup>1</sup> tornou evidente o potencial dos modelos de linguagem de grande escala (do inglês, *Large Language Models* – LLM) (KORZYŃSKI et al., 2023). Da mesma forma, outras empresas também introduziram modelos generativos competitivos, como o Llama<sup>2</sup>, Claude<sup>3</sup> e Gemini<sup>4</sup>. Com a crescente demanda por resultados mais precisos, a engenharia de *prompts* (do inglês, *Prompt Engineering* – PE) surgiu como uma prática importante no contexto de uso dos LLMs como um meio de construir instruções mais adequadas e direcionadas para a obtenção de respostas alinhadas aos interesses do usuário, sem a necessidade de retreinamento dos modelos (VATSAL; DUBEY, 2024).

De acordo com Schulhoff et al. (2025) e Medeiros et al. (2024), o PE é um processo iterativo direcionado ao desenvolvimento de *prompts* eficazes, utilizando diferentes técnicas. Nesta dissertação, a engenharia de *prompts* é analisada especificamente no contexto da interação com LLMs, sendo considerada como o processo de elaboração de instruções estruturadas e semanticamente orientadas para guiar o comportamento desses modelos em tarefas específicas. Após o envio de uma pergunta ou instrução, o modelo processa essas informações e gera uma resposta automática contextualizada, que pode assumir diferentes formatos, como texto explicativo, código-fonte, resumo ou síntese. Essa abordagem possui ampla aplicabilidade, especialmente em contextos que demandam geração automatizada de conteúdo técnico, como o desenvolvimento de software e o apoio à programação.

Conforme o levantamento apresentado por (SCHULHOFF et al., 2025), a literatura recente sobre PE identifica 33 termos distintos utilizados para descrever conceitos relacionados a *prompts*, além de catalogar 58 técnicas específicas de *prompting* para LLMs baseados em texto e 40 técnicas adicionais aplicáveis a outras modalidades de dados, como imagem, áudio e vídeo. Essa diversidade evidencia a amplitude do campo e a variedade de estratégias disponíveis para diferentes tipos de tarefas, como geração de código, explicação automatizada, sumarização de

---

<sup>1</sup> <https://openai.com/>

<sup>2</sup> <https://www.llama.com/>

<sup>3</sup> <https://claude.ai/new>

<sup>4</sup> <https://gemini.google.com/>

documentos técnicos, análise de dados textuais e interação multimodal, ampliando o espaço de exploração metodológica nesse domínio.

Liu et al. (2023) apresentam uma perspectiva estruturada sobre o uso de *prompts*, denominada “pré-treino, *prompt* e predição” (do inglês, *pre-train, prompt, and predict*). Nessa concepção, os autores discutem diferentes categorias de *prompts*, cada uma associada a finalidades específicas na interação com modelos de linguagem. Essas categorias incluem: (i) a personalização dos *prompts* de acordo com as características do pré-treinamento do LLM, considerando diferenças arquiteturais e de vocabulário (OWEN; ANDREW; PADLSKI, 2024); (ii) a engenharia de *templates* de *prompt* adaptáveis a tarefas específicas, com o objetivo de aumentar a clareza e a consistência das respostas (XU et al., 2023); (iii) a estruturação de interações no formato pergunta–resposta, visando maior controle e objetividade das saídas geradas (LI; LEUNG; SHEN, 2024); (iv) o uso de *frameworks* que combinam múltiplos *prompts* em série ou em paralelo para otimizar resultados (VATSAL; DUBEY, 2024); e (v) a aplicação de estratégias como *Zero-Shot* e *Few-Shot* para ajustar o comportamento do modelo antes da execução da tarefa principal (LI; LEUNG; SHEN, 2024). Essas categorias evidenciam a diversidade de formas pelas quais *prompts* podem ser empregados na interação com LLMs.

Para Neves et al. (2024), o uso de *prompts* exerce um papel fundamental no desempenho dos LLMs, evidenciando que técnicas de PE cuidadosamente elaboradas podem influenciar significativamente a qualidade das respostas geradas. Esse impacto é particularmente evidente em tarefas como análise de sentimentos, nas quais *prompts* bem estruturados ajudam a orientar o modelo na identificação mais precisa de nuances linguísticas, como ironias, intensificadores e negações, resultando em classificações mais coerentes com o contexto semântico do texto de entrada. Assim, a precisão do modelo não é apenas incrementada quantitativamente, mas qualitativamente, refletindo maior sensibilidade ao conteúdo e à intenção comunicativa.

Assim como em tarefas de linguagem natural, a performance de ferramentas associadas à geração de códigos de programação, como GitHub Copilot e Gemini, também é fortemente influenciada pela maneira como os *prompts* são estruturados. Estudos mostram que *prompts* contextualizados — incluindo requisitos de explicação ou exemplos de saída esperada para códigos — elevam não apenas sua precisão, mas também sua capacidade de autodocumentação (MAHIR; SHOHEL; SALL, 2024). Na prática, isso permite, por exemplo, a geração de funções completas com documentação embutida, explicação de algoritmos complexos e apoio à revisão de código em ambientes colaborativos. Sabe-se que a geração automática de códigos de programação deve estar acompanhada de explicações claras e estruturadas, pois a compreensibilidade do código é essencial para sua validação, manutenção, reprodutibilidade científica e uso responsável, especialmente em contextos educacionais e de pesquisa. Nesse sentido, MacNeil et al. (2022) esclarecem que existem três abordagens distintas para a explicação de código: a explicação linha a linha, a listagem de conceitos importantes e o resumo de alto nível do código. Essas estratégias têm sido amplamente utilizadas para tornar o entendimento de código mais acessível

e estruturado, especialmente em contextos educacionais ou de revisão de software.

Adicionalmente, Bhattacharya et al. (2023) demonstram que modelos especializados em código tendem a apresentar melhor desempenho em tarefas de explicação quando comparados a modelos de uso geral, evidenciando a importância de estratégias adequadas de interação com os LLMs. Entretanto, conforme destacado por Liu et al. (2024), mesmo quando os modelos geram código sintaticamente correto, há uma lacuna significativa quanto à confiabilidade e à qualidade das explicações produzidas, sobretudo em tarefas mais complexas. Esse cenário reforça a necessidade de avaliar não apenas a correção do código, mas também a coerência e a adequação das explicações associadas.

Nesse contexto, Ding et al. (2024) destacam que a engenharia de *prompts* pode desempenhar um papel fundamental não apenas na geração de código por LLMs, mas também na capacidade desses modelos de explicar suas próprias decisões de forma inteligível. Os autores observam que instruções bem estruturadas permitem ao modelo produzir respostas que conciliam correção técnica e clareza explicativa, reforçando a importância do design do *prompt* como elemento central na interação com LLMs.

De forma complementar, Yeo et al. (2024) argumentam que estratégias baseadas em múltiplas interações sucessivas com o modelo contribuem para a melhoria gradual da qualidade das respostas. Segundo os autores, a introdução progressiva de novas informações, reformulações ou esclarecimentos ao longo da interação permite que o LLM acumule contexto, resultando em respostas mais precisas, coerentes e alinhadas à intenção do usuário.

A elaboração de um *prompt* depende de vários fatores. De acordo com Bansal (2024), é fundamental adaptar o LLM por meio da PE para atingir um objetivo específico, garantindo que os resultados sejam relevantes e valiosos e, ao mesmo tempo, minimizando problemas como alucinações e não-determinismo. As alucinações ocorrem quando o LLM gera informações plausíveis, mas incorretas ou não fundamentadas nos dados de treinamento (WANG et al., 2025a). O não-determinismo refere-se à capacidade do LLM de gerar respostas diversas para a mesma entrada em diferentes execuções, devido a elementos de aleatoriedade incorporados no processo de geração do texto (ATIL et al., 2025).

Nesse contexto, com base na análise de vários estudos sobre técnicas de PE e suas aplicações (VATSAL; DUBEY, 2024; SHIN et al., 2023; REYNOLDS; MCDONELL, 2021; NING et al., 2024; BANSAL, 2024; KOJIMA et al., 2022), é possível observar que algumas técnicas foram mais abordadas na literatura, enquanto outras, embora com provável potencial de uso, ainda são pouco exploradas na prática. Assim, a combinação de técnicas de PE simples com abordagens mais refinadas configura-se como um campo aberto para pesquisa e experimentação. O desenvolvimento de *frameworks* que combinem essas técnicas pode aprimorar o resultado de alguns contextos de problemas, a exemplo da geração de código. Essa visão é apoiada por Neves et al. (2024), Khojah et al. (2025), que enfatizam a necessidade de explorar combinações de técnicas de PE que possam melhorar o desempenho de um LLM.

Com o objetivo de investigar de forma sistemática o impacto do uso combinado de técnicas de PE em tarefas de geração e explicação de código, este trabalho estrutura sua análise a partir de um conjunto de Questões de Pesquisa (QP).

- **QP1:** Como diferentes combinações de técnicas de engenharia de *prompts* influenciam a qualidade da geração de código em LLMs, considerando métrica de similaridade textual?
- **QP2:** Os efeitos das combinações de técnicas de engenharia de *prompts* observados na geração de código mantêm-se consistentes quando avaliados em um conjunto ampliado e heterogêneo de LLMs?
- **QP3:** Como diferentes combinações de técnicas de engenharia de *prompts* impactam a qualidade não somente da geração mas também da explicação de código quando avaliadas por *LLM-as-a-Judge*?
- **QP4:** Técnicas de engenharia de *prompts* pouco exploradas na literatura, quando combinadas a técnicas amplamente investigadas e recorrentes na literatura, contribuem para a melhoria da qualidade das respostas em tarefas de geração e explicação de código?

Essas questões são formuladas de modo a refletir a progressão experimental adotada ao longo da pesquisa, contemplando desde a caracterização inicial dos efeitos das combinações de *prompts* na geração de código até a análise de sua robustez em diferentes modelos e sob distintos critérios de avaliação. Busca-se, em particular, compreender não apenas o efeito geral das combinações de técnicas de PE, mas também o papel desempenhado por estratégias pouco exploradas na literatura quando integradas a abordagens mais consolidadas, bem como a consistência desses efeitos em múltiplos LLMs e em tarefas que envolvem tanto a geração quanto a explicação de código.

Este trabalho desenvolve uma investigação experimental que abrange desde a seleção e combinação de técnicas de PE, até sua implementação e posterior avaliação. Como cenário de aplicação, adota-se a tarefa de geração e explicação de código em Python, linguagem escolhida por sua ampla adoção em contextos educacionais, científicos e profissionais, além de sua sintaxe clara e estruturada. Inicialmente, são definidas combinações de *prompts* envolvendo tanto estratégias amplamente utilizadas quanto outras menos exploradas (KHOJAH et al., 2025), com o objetivo de analisar seu comportamento em LLMs extensivamente referenciados na literatura. A avaliação dos resultados da pesquisa é conduzida sob duas perspectivas complementares: (a) quantitativa, por meio da métrica automática de similaridade textual, o BERTScore, e (b) avaliativa automatizada, com a incorporação de avaliações utilizando a abordagem *LLM-as-a-Judge* (LaaJ).

Ainda no contexto exposto, a explicação do código visa não apenas verificar se os modelos são capazes de produzir soluções corretas, mas também avaliar se os mesmos conseguem

justificar seus procedimentos de forma coerente, precisa e acessível ao entendimento de usuários humanos, por exemplo, em cenários de formação e aprendizagem em programação.

Para conduzir o processo experimental proposto neste trabalho, foram selecionados seis modelos de linguagem de grande escala representativos de diferentes famílias arquiteturais e estratégias de treinamento com o objetivo de verificar a consistência dos efeitos das combinações de técnicas de engenharia de *prompt* em diferentes LLMs. A adoção de modelos distintos não visa estabelecer comparações diretas de desempenho entre arquiteturas, mas examinar se os padrões observados na aplicação das combinações de *prompts* se mantêm independentemente do modelo utilizado, reforçando o caráter geral e transferível dos achados.

Ao utilizar LLMs para a geração de código e suas respectivas explicações, emergem diversos desafios que extrapolam a simples produção de soluções funcionais. Entre eles, destacam-se a correção sintática, relacionada à conformidade com as regras da linguagem de programação; a correção semântica, que diz respeito à lógica e ao comportamento esperado do código; e a eficiência, entendida como a capacidade de executar tarefas de forma adequada, com uso racional de recursos computacionais. Adicionalmente, as explicações associadas ao código devem ser concisas, informativas e coerentes com o funcionamento da solução apresentada. Esses aspectos evidenciam a complexidade inerente à avaliação do desempenho de LLMs em tarefas de programação (COIGNION; QUINTON; ROUVOY, 2024; SARKER et al., 2024).

## 1.2 Objetivos

Com o propósito de responder às questões de pesquisa delineadas na seção anterior, apresentam-se, a seguir, o objetivo geral da presente abordagem e os respectivos objetivos específicos.

### 1.2.1 Objetivo geral

Explorar o potencial de técnicas de PE, por meio de combinações estratégicas e experimentação controlada em modelos de IA generativa, com o intuito de investigar quais combinações de *prompts* se mostram mais promissoras para a melhoria da geração e da explicação de código em Python, oferecendo subsídios ao apoio à prática da programação.

### 1.2.2 Objetivos específicos

- Explorar e selecionar técnicas de PE associadas às tarefas de geração e explicação de código;
- Propor e estruturar combinações de técnicas de PE aplicáveis às tarefas de geração e explicação de código na linguagem Python;

- Planejar e conduzir experimentos controlados para avaliar o desempenho de diferentes combinações de técnicas de PE propostas;
- Analisar comparativamente os resultados experimentais obtidos, a fim de identificar as combinações mais eficazes;
- Empregar métricas de similaridade semântica para avaliar a qualidade do código gerado pelos LLMs, com base na comparação com códigos de referência.
- Realizar avaliação automatizada das respostas geradas pelos LLMs, empregando o paradigma LaaJ, a fim de analisar a correção do código e a qualidade das explicações produzidas.

### 1.3 Contribuições

Esta pesquisa apresenta contribuições de natureza científica, metodológica e aplicada.

No âmbito científico, o estudo investiga de forma sistemática a combinação de técnicas de PE em LLMs. Parte-se da hipótese de que a integração de diferentes estratégias de *prompting* pode produzir efeitos distintos — e potencialmente superiores — em relação ao uso isolado dessas técnicas, especialmente em tarefas que envolvem geração e explicação de código. A pesquisa contribui, assim, para a compreensão de como essas interações influenciam o desempenho dos modelos sob diferentes critérios de avaliação.

Como contribuição metodológica, propõe-se um desenho experimental controlado para a avaliação de combinações de técnicas de PE, apoiado por uma infraestrutura que padroniza a execução das configurações experimentais e a aplicação das estratégias de avaliação, favorecendo a comparabilidade e a reprodutibilidade das análises. No âmbito aplicado, a pesquisa concentra-se no uso de LLMs em tarefas de geração e explicação de código, com foco na análise de como diferentes estratégias de *prompting* influenciam as respostas produzidas pelos modelos.

Esse enfoque contribui para o uso mais eficaz de técnicas de engenharia de *prompts* em contextos como apoio ao desenvolvimento de software e ensino de programação, nos quais a qualidade das explicações desempenha papel central na compreensão, validação e utilização das soluções geradas.

### 1.4 Estrutura do Documento

Os capítulos subsequentes estão organizados da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, contemplando conceitos relevantes para esta pesquisa. O Capítulo 3 discute trabalhos relacionados. O Capítulo 4 descreve o processo metodológico adotado para esta pesquisa. O Capítulo 5 mostra os resultados obtidos e as discussões associadas. Por fim, o Capítulo 6 expõe as considerações finais.

## 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos que fundamentam o problema investigado nesta pesquisa, fornecendo a base teórica essencial para a compreensão do tema.

### 2.1 Modelos de Linguagem

A compreensão computacional da linguagem humana tem sido um objetivo central da Inteligência Artificial (IA), acompanhando a evolução do próprio campo ao longo das últimas décadas. Inicialmente fundamentada em abordagens estatísticas, essa trajetória avançou progressivamente para arquiteturas neurais de grande escala, capazes de compreender e gerar linguagem natural de forma cada vez mais sofisticada.

A literatura recente sintetiza essa evolução dos modelos de linguagem em quatro fases principais: **Modelos Estatísticos de Linguagem** (do inglês, *Statistical Language Models – SLM*), **Modelos Neurais de Linguagem** (do inglês, *Neural language models – NLM*), **Modelos de Linguagem Pré-treinados** (do inglês, *Pre-trained language models – PLM*) e **Modelos de Linguagem de Grande Escala** (LLM). Nessa perspectiva, a distinção fundamental entre PLMs e LLMs reside sobretudo na escala de parâmetros e de dados utilizados no treinamento, bem como no surgimento de habilidades emergentes nos modelos de maior porte. Essa classificação técnica é sistematizada de forma abrangente na revisão proposta por Zhao et al. (2025a).

Complementarmente, trabalhos de referência no contexto do processamento de linguagem natural em língua portuguesa apontam que o campo também pode ser compreendido a partir de diferentes paradigmas metodológicos — simbólico, estatístico, neural e híbrido. Conforme discutido por Caseli e Nunes (2024), nenhum desses paradigmas, isoladamente, é capaz de capturar de forma completa o conhecimento linguístico envolvido no uso da linguagem humana, sendo as escolhas metodológicas fortemente condicionadas pelos objetivos da aplicação, pela disponibilidade de recursos e pelas particularidades linguísticas do idioma. Essa perspectiva contribui para contextualizar e aprofundar a compreensão das implicações práticas associadas às diferentes fases técnicas da evolução dos modelos de linguagem.

Os SLMs, baseados em modelos probabilísticos como *n-grams*, foram amplamente utilizados em tarefas iniciais de processamento de linguagem natural, mas apresentavam limitações relacionadas ao crescimento do espaço de estados e à dependência de contexto limitado (GAO; LIN, 2004; LIU; CROFT, 2005). Com o avanço das redes neurais, os NLMs introduziram representações distribuídas de palavras, como o *Word2Vec* (MIKOLOV et al., 2013), permitindo capturar relações semânticas de forma mais expressiva e impulsionando avanços em diversas tarefas linguísticas (ZHAO et al., 2025a).

A introdução dos PLMs consolidou o paradigma de pré-treinamento em grandes corpora seguido de ajuste fino (do inglês, *fine-tuning*) para tarefas específicas. Modelos como o BERT (*Bidirectional Encoder Representations from Transformers*) (DEVLIN et al., 2019), que implementa a arquitetura *Transformer* (VASWANI et al., 2023), demonstraram que representações contextuais profundas podem ser transferidas de forma eficaz para múltiplos domínios, incluindo tarefas de geração de texto e código (RADFORD et al., 2019; LEWIS et al., 2020a).

A ampliação sistemática da escala desses modelos levou ao surgimento dos LLMs, como GPT-3 e PaLM (CHOWDHERY et al., 2023). Nesse contexto, observa-se o aparecimento de habilidades emergentes, entendidas como capacidades qualitativamente novas que passam a se manifestar quando esses modelos atingem grande escala, seja em termos de número de parâmetros, volume de dados ou poder computacional. Entre essas capacidades destacam-se a aprendizagem em contexto (do inglês, *In-Context Learning – ICL*), o seguimento de instruções (do inglês, *Instruction Following – IF*) e o raciocínio passo a passo (do inglês, *Step-by-Step Reasoning – SSR*) (ZHAO et al., 2025a; WEI et al., 2022a). Embora a origem exata desses comportamentos ainda não seja completamente compreendida, sua emergência tem sido associada a mudanças qualitativas decorrentes do aumento de escala, tornando os LLMs particularmente eficazes em tarefas complexas.

Essas capacidades são fundamentais para aplicações práticas, pois permitem que os modelos generalizem para tarefas novas, interpretem instruções formuladas em linguagem natural e construam respostas mais estruturadas. Em especial, o raciocínio passo a passo pode ser induzido por meio de estratégias de engenharia de *prompt*, como o *Chain-of-Thought*, que orienta o modelo a explicitar etapas intermediárias antes de apresentar a resposta final (WEI et al., 2022a). Tais características explicam, em grande medida, o desempenho superior dos LLMs em cenários que exigem compreensão contextual e elaboração de respostas justificadas, como a geração e a explicação de código.

O avanço dos LLMs também está associado a práticas consolidadas de treinamento e ajuste, como o pré-treinamento em grandes corpora, o ajuste supervisionado e o uso de estratégias baseadas em instruções, que contribuem para tornar os modelos mais controláveis, versáteis e alinhados às expectativas dos usuários (OUYANG et al., 2022; ZHAO et al., 2025a). Esses elementos, combinados às habilidades emergentes, constituem a base técnica que viabiliza o uso de LLMs em tarefas orientadas por *prompts* complexos, contexto no qual se insere a Engenharia de *Prompts* discutida nas seções subsequentes.

Além dessas características estruturais, a utilização prática dos LLMs envolve a definição de hiperparâmetros de inferência, que controlam o comportamento do modelo durante a geração das respostas. Esses parâmetros, definidos externamente ao processo de treinamento, influenciam diretamente aspectos como diversidade, consistência, custo computacional e reprodutibilidade das saídas (WANG; LIU; AWADALLAH, 2023). Entre os principais hiperparâmetros destacam-se: o modelo selecionado (*model*), que determina a arquitetura e as capacidades dispo-

níveis; a sequência de mensagens (*messages*), que representa o *prompt* submetido ao modelo; a temperatura (*temperature*), responsável por regular o grau de aleatoriedade na geração textual (PEEPERKORN et al., 2024); o limite máximo de *tokens* gerados (*max\_tokens*), que restringe a extensão da resposta (ZIMMERMAN et al., 2024); e o número de respostas produzidas (*n*), que possibilita a geração de múltiplas saídas para uma mesma entrada. A configuração adequada desses hiperparâmetros é essencial para garantir respostas mais estáveis e comparáveis, especialmente em cenários experimentais controlados, como os investigados nesta dissertação.

## 2.2 Uso de LLM para geração e explicação de código

Esta seção trata do uso de LLMs em tarefas relacionadas à programação, abrangendo a geração de código-fonte, a explicação de código em linguagem natural e os aspectos de avaliação associados a essas aplicações.

### 2.2.1 Geração

De acordo com Zhao et al. (2025a), os LLMs também têm demonstrado elevado desempenho em tarefas de geração de código em linguagens de programação, especialmente na produção automatizada de código-fonte, atividade conhecida como síntese de código (do inglês, *code synthesis*). Essa tarefa difere da geração de linguagem natural por oferecer uma forma objetiva de avaliação: o código gerado pode ser validado por meio da sua execução direta em interpretadores ou compiladores, permitindo verificar automaticamente sua correção funcional (CHEN et al., 2025).

As tarefas de geração de código abrangem uma ampla gama de atividades, entre as quais se destacam (CHEN et al., 2025):

- **Tradução de requisitos em linguagem natural para código executável:** por exemplo, converter uma história de usuário (do inglês, *user story*) ou a descrição de uma funcionalidade em um script de programação funcional.
- **Preenchimento de *templates* de código:** completar trechos de código parcialmente escritos, adicionando as partes ausentes com base no contexto fornecido.
- **Refatoração e otimização de código existente:** modificar e melhorar a estrutura ou a eficiência de um código sem alterar sua funcionalidade original.
- **Geração de casos de teste:** criar *scripts* e cenários de teste para validar a correção e o desempenho de um determinado trecho de código.

O desempenho de LLMs nessas tarefas é consideravelmente ampliado quando os modelos são pré-treinados ou ajustados com grandes *corpora* de código-fonte, como repositórios públicos

do GitHub. Um exemplo emblemático é o modelo Codex (CHEN et al., 2021), derivado do GPT-3, que foi ajustado especificamente para tarefas de programação. O Codex demonstrou não apenas maior precisão na geração de código, como também melhorias em tarefas de raciocínio matemático estruturado, sugerindo que o conhecimento sobre programação pode fortalecer capacidades gerais de raciocínio dos modelos.

Além do Codex, destacam-se modelos como o CodeGen (NIJKAMP et al., 2023), treinado com datasets amplos de código e documentação técnica, e o StarCoder (LI et al., 2023), construído com foco em suporte multilíngue e desempenho em múltiplas linguagens de programação. Esses modelos se diferenciam dos LLMs generalistas por apresentarem maior precisão sintática e semântica na geração de trechos funcionais, sendo especialmente úteis em tarefas como autocompletar funções, corrigir erros, traduzir entre linguagens e até mesmo explicar a lógica de programação.

Segundo Zhao et al. (2025a), o surgimento de modelos especializados como CodeGen e StarCoder reforça o potencial dos LLMs quando adaptados a domínios específicos, ampliando sua utilidade em áreas técnicas como o desenvolvimento de software.

Embora existam LLMs desenvolvidos especificamente para tarefas de programação, como geração e complemento de código, esta pesquisa adota uma abordagem centrada em modelos de uso generalista, amplamente empregados em contextos diversos. A escolha por modelos não especializados visa ampliar a aplicabilidade dos resultados e refletir cenários mais comuns do uso cotidiano desses sistemas. As decisões metodológicas e o escopo da investigação serão detalhados no Capítulo 4.

### 2.2.2 Explicação

A explicação de código é uma das dimensões fundamentais da habilidade de sua compreensão (do inglês, *code comprehension*) e se refere à capacidade de entender e interpretar trechos de código escritos em linguagens de programação (SCHULTE et al., 2010). Esse processo envolve a leitura, análise e interpretação do código com o objetivo de compreender sua finalidade, funcionalidade e estrutura. De acordo com Oli et al. (2024), diversos pesquisadores defendem a inclusão da leitura e interpretação de código no ensino de programação, sugerindo, inclusive, que estudantes deveriam aprender a ler código antes mesmo de escrevê-lo.

Com a crescente adoção de LLMs no apoio à aprendizagem de programação, a explicação automática de código ganhou relevância como ferramenta educacional e prática. Para compreender o potencial desses modelos na tarefa de explicação de código, é fundamental revisitar brevemente a trajetória histórica e metodológica sobre como essa atividade tem sido estudada e avaliada na literatura. Essa análise permite contextualizar os avanços trazidos pelos LLMs e estabelecer critérios mais robustos para a avaliação de sua eficácia na compreensão e comunicação da lógica de programas computacionais (OLI et al., 2024).

Sarsa et al. (2022) ressaltam que a habilidade de explicar código é desafiadora de ser avaliada, pois envolve linguagem natural e diferentes níveis de abstração. Uma estratégia comum utilizada em contextos educacionais é o formato de pergunta “explique em inglês simples” (do inglês, “*explain in plain English*”), em que estudantes devem descrever o propósito de um bloco de código fornecido. Esse tipo de questão foi amplamente estudado no projeto BRACElet (WHALLEY et al., 2006) com base nos quatro primeiros níveis da taxonomia SOLO (BIGGS; COLLIS, 2014), permitindo classificar as respostas em três graus principais de explicação:

- **Linha a linha** (do inglês, *line-by-line*): que descreve cada instrução individualmente, sem captar o funcionamento global do código;
- **Multiestrutural** (do inglês, *multistructural*): agrupa elementos do código de forma mais conectada, mas ainda sem expressar de maneira sucinta seu propósito geral;
- **Relacional** (do inglês, *relational*): representa a forma mais abstrata e eficaz de explicação, caracterizada por interpretações que descrevem como o código se comporta diante de diferentes entradas, revelando seu objetivo de forma clara e generalizada. Um exemplo clássico de BRACElet para esses três níveis é a explicação “verifica se o vetor está ordenado”, para um código que compara elementos adjacentes em um laço (LISTER, 2020)

Com base nessa taxonomia de explicações, MacNeil et al. (2022) investigam como diferentes modelos de linguagem produzem explicações de código em distintos níveis de abstração, destacando a relevância do alinhamento entre o formato explicativo e o público-alvo. O estudo contrasta modelos de uso geral e modelos especializados em programação, evidenciando que a capacidade de gerar explicações adequadas depende não apenas da correção técnica do código, mas também da forma como o raciocínio subjacente é apresentado. A partir dessa análise, os autores sistematizam três formatos principais de explicação de código, que diferem quanto ao grau de detalhamento e à abstração empregada.

- **Linha a linha**: apresenta uma descrição detalhada e sequencial de cada linha do código, explicando sua função individual. Esse formato é particularmente útil para iniciantes, pois oferece uma leitura estruturada do código, promovendo o entendimento passo a passo de sua execução.
- **Lista de conceitos importantes** (do inglês, *list of key concepts*): em vez de abordar cada linha isoladamente, esse formato destaca os principais conceitos de programação presentes no código, como estruturas de repetição, funções utilizadas ou bibliotecas chamadas. Serve como um guia conceitual que permite ao estudante compreender quais elementos fundamentais estão em uso naquele trecho.

- **Resumo de alto nível** (do inglês, *high-level summary*): fornece uma visão geral do propósito e da funcionalidade do código como um todo, sem detalhar sua implementação. É mais adequado a aprendizes com maior familiaridade com programação, que desejam rapidamente entender a intenção e o comportamento global do código.

Tanto Sarsa et al. (2022) quanto MacNeil et al. (2022) observaram uma preferência consistente pelo formato de explicação de código do tipo linha a linha, independentemente do nível de experiência dos especialistas de domínio. Esse formato, por sua natureza sequencial e detalhada, facilita a compreensão do funcionamento do código, tornando-se acessível tanto para iniciantes quanto para programadores mais experientes.

Complementando essa perspectiva, Bhattacharya et al. (2023) investigaram o uso de LLMs na tarefa de explicação de código, empregando tanto modelos generalistas, como o Llama-2-70B-Chat (TOUVRON et al., 2023), quanto modelos especializados, como o StarCoder. Os resultados indicam que esses modelos são capazes de gerar descrições informativas sobre a funcionalidade de funções e trechos de código. Os autores destacam ainda que a natureza das explicações pode variar de acordo com o modelo utilizado e com o tipo de saída esperada, oscilando entre resumos compactos do propósito do código e explicações mais detalhadas das instruções.

Assim, os estudos analisados contribuem para a compreensão do potencial dos LLMs na geração de explicações de código, reforçando seu uso como ferramenta de apoio à leitura, documentação e ensino de programação. A escolha do formato explicativo depende do modelo utilizado, da configuração da tarefa e do nível de detalhamento necessário, o que demonstra a flexibilidade desses sistemas em contextos práticos de engenharia de software.

### 2.2.3 Avaliação

A avaliação de LLMs em tarefas de geração e explicação de código exige critérios que vão além da correção funcional, contemplando aspectos como similaridade semântica, coerência e adequação das respostas. Esta subseção apresenta conceitos e abordagens discutidos na literatura para essa avaliação, incluindo métricas de similaridade semântica, como a BERTScore, e métodos baseados no paradigma LaaJ.

#### 2.2.3.1 Métricas

Atualmente, a literatura apresenta diversas métricas para avaliar o desempenho de LLMs. Essas métricas podem ser categorizadas em três grupos de acordo com diferentes aspectos da avaliação, como qualidade da geração de conteúdo, coerência, precisão e relevância das respostas produzidas (HU; ZHOU, 2024).

- **Classificação Múltipla** (do inglês, *Multiple-Classification* - MC): nesta categoria, encontram-se métricas utilizadas para classificar textos em rótulos predefinidos, como em análises de sentimentos, por exemplo, “positivo” ou “negativo” (ZHAO et al., 2024);
- **Pergunta e Resposta** (do inglês, *Question-Answering* - QA): métricas específicas para avaliar tarefas de QA que envolvem correspondência imprecisa (*fuzzy matching*), ou seja, a capacidade do LLM de reconhecer respostas corretas mesmo quando há variações na formulação das perguntas ou respostas, como sinônimos, pequenas diferenças de ortografia, variações gramaticais ou respostas parciais. Nesses cenários, o modelo deve identificar a resposta correta para uma pergunta específica com base em um contexto fornecido, tolerando discrepâncias textuais sem perda de significado (HU; ZHOU, 2024);
- **Similaridade de Tokens** (do inglês, *Token-Similarity* - TS) refere-se a um conjunto de métricas utilizadas para avaliar o grau de alinhamento entre textos gerados por LLMs e textos de referência (*ground-truth*), com base na comparação de unidades lexicais ou representações derivadas de *tokens*. Entre as métricas consolidadas na literatura destacam-se Perplexity (BROWN et al., 1988), BLEU (PAPINENI et al., 2002), ROUGE-n e ROUGE-L (LIN, 2004), METEOR (BANERJEE; LAVIE, 2005), ChrF++ (POPOVIĆ, 2015) e BERTScore (ZHANG et al., 2020). Essas métricas são amplamente empregadas na avaliação do grau de correspondência textual, da cobertura semântica, da qualidade estrutural do código gerado e da consistência dos textos produzidos por modelos de linguagem.

Entretanto, Wang et al. (2025b), Zhao et al. (2025b), Tong e Zhang (2024) propõem e avaliam uma abordagem para a avaliação automatizada em tarefas de engenharia de software, introduzindo o conceito de *LLM-as-a-Judge*. Essa abordagem consiste no uso de LLMs como agentes avaliadores, capazes de julgar a qualidade das respostas geradas por humanos ou por outros modelos em tarefas como tradução, geração e explicação de código. Diferentemente das métricas automáticas tradicionais, como BLEU, CodeBLEU ou ChrF++, o *LLM-as-a-Judge* (LaaJ) baseia-se na capacidade dos modelos de linguagem em emitir julgamentos qualitativos ou quantitativos sobre as saídas analisadas. Esses julgamentos podem ser realizados por meio de dois métodos principais: (a) avaliação baseada em pontuação (do inglês, *output-based*), em que o modelo atribui uma nota direta a uma resposta, e (b) comparações par a par (do inglês, *pairwise*), em que o modelo escolhe a melhor entre duas ou mais alternativas apresentadas.

No contexto desta pesquisa, que investiga a geração de código por meio de combinações de *prompts*, a avaliação foi conduzida em dois eixos distintos: um eixo baseado em métricas de similaridade textual e outro fundamentado no paradigma LaaJ. As métricas de similaridade de *tokens* são essenciais para quantificar o quão próximas as saídas geradas estão dos textos de referência, permitindo avaliar a qualidade e fidelidade da resposta gerada. Portanto, a BERTScore foi adotada por sua capacidade avançada de capturar similaridades semânticas profundas entre textos. Diferentemente das métricas tradicionais baseadas em correspondência exata de *tokens* ou

*n*-gram, como BLEU e ROUGE, que avaliam apenas sobreposição lexical, a BERTScore utiliza *embeddings* pré-treinados do modelo BERT (DEVLIN et al., 2019) para medir a similaridade semântica entre as sequências avaliadas. Isso significa que, mesmo quando as expressões utilizadas na geração do código ou em sua explicação variam lexicalmente, o BERTScore consegue reconhecer a equivalência de significado, o que é fundamental para avaliar a performance do LLM em tarefas de geração de código, onde a variação na forma pode não comprometer a correção ou a explicação.

A aplicação do BERTScore segue uma sequência estruturada:

1. **Carregamento dos *embeddings* BERT:** o modelo BERT pré-treinado é empregado para a extração de *embeddings* contextualizados dos *tokens*. No processamento de linguagem natural, *tokens* são as unidades elementares que compõem uma sequência de entrada para modelos linguísticos, geralmente correspondendo a palavras ou subpalavras derivadas por meio do processo de tokenização. Este processo segmenta o texto bruto em unidades discretas para viabilizar sua representação numérica e subsequente processamento computacional (ZHANG et al., 2020). *Embeddings* consistem em representações vetoriais densas desses *tokens*, projetadas para capturar características semânticas e contextuais. Conforme descrito por Zhang et al. (2020), os *embeddings* extraídos do BERT incorporam informações contextuais, atribuindo a cada *token* uma representação dependente das unidades linguísticas circundantes. Tal contextualização é imprescindível para a modelagem precisa do significado semântico do texto, sobretudo em tarefas de geração e avaliação, nas quais a interpretação correta depende intrinsecamente do contexto específico em que as unidades textuais estão inseridas;
2. **Representação dos *tokens*:** para cada *token* no texto gerado  $\{x_i\}_{i=1}^N$  e no texto de referência  $\{y_j\}_{j=1}^M$ , os *embeddings* correspondentes são extraídos utilizando um modelo pré-treinado, como o BERT. Cada *token* é representado como um vetor de comprimento fixo no espaço de *embedding* com a notação *e*, resultando nas sequências vetoriais  $\{\hat{x}_i^e\}_{i=1}^N$  e  $\{\hat{y}_j^e\}_{j=1}^M$ . Esses vetores representam os *tokens* mapeados para um espaço semântico denso, no qual relações de similaridade podem ser quantitativamente avaliadas.

Para complementar, também se calcula uma representação vetorial agregada das sequências, denotada por  $\hat{x}^e$  e  $\hat{y}^e$ , obtida por meio da operação de *mean pooling*, ou seja, pela média aritmética dos *embeddings* individuais dos *tokens* de cada sequência. Essas representações globais, ainda no espaço de *embedding* *e*, são utilizadas no cálculo da similaridade do cosseno, medida central no BERTScore, que quantifica o alinhamento semântico entre o texto gerado e o texto de referência.

3. **Medição de Similaridade:** a sequência de vetores é consolidada em uma representação única para quantificar a similaridade entre os dois textos, conforme a Equação 1:

$$\hat{x}^e = \frac{1}{N} \sum_{i=1}^N \hat{x}_i^e, \quad \hat{y}^e = \frac{1}{M} \sum_{j=1}^M \hat{y}_j^e \quad (1)$$

Por fim, a função de similaridade cosseno é aplicada para medir a similaridade entre as representações vetoriais do texto de referência e do texto gerado. Essa similaridade é definida como o cosseno do ângulo entre dois vetores de *embedding*, refletindo o grau de alinhamento semântico entre eles, conforme demonstrado na Equação 2:

$$\text{BERTScore} = \frac{\hat{x}^e \cdot \hat{y}^e}{\|\hat{x}^e\| \cdot \|\hat{y}^e\|} \quad (2)$$

O resultado do cômputo do BERTScore gera uma pontuação de similaridade dentro do intervalo [-1..1], em que:

- 1 indica vetores idênticos (alta similaridade);
- 0 representa vetores ortogonais (nenhuma similaridade);
- -1 denota completa dissimilaridade.

Considerando suas propriedades de avaliação semântica, o BERTScore é adotado nesta dissertação como métrica para a análise da similaridade entre o código gerado pelos LLMs e os trechos de referência presentes na base *ground-truth*. Sua capacidade de capturar relações de significado mesmo na presença de variações estruturais ou sintáticas torna essa métrica particularmente adequada ao contexto deste trabalho, no qual soluções funcionalmente equivalentes podem apresentar implementações distintas. Os procedimentos específicos de aplicação do BERTScore são detalhados no Capítulo 4.

### 2.2.3.2 LLM-as-a-Judge

Segundo Luo et al. (2025) a utilização de LaaJ para avaliação de explicação de código representa um avanço significativo no estudo de modelos de linguagem como avaliadores automáticos. O artigo propõe um arcabouço sistemático para avaliar não apenas a correção factual de explicações geradas por LLMs, mas também sua consistência com o código subjacente, sua estrutura argumentativa e sua utilidade pedagógica — aspectos tradicionalmente avaliados apenas por humanos. A pesquisa demonstra que LLMs modernos podem atuar como juízes confiáveis quando devidamente condicionados por critérios explícitos de avaliação, oferecendo uma alternativa escalável a processos manuais de revisão.

Um dos principais méritos do estudo está na elaboração de protocolos de avaliação orientados a critérios, nos quais o modelo avaliador recebe instruções específicas que guiam sua análise sobre dimensões como: clareza narrativa, exatidão técnica, completude, alinhamento com a intenção do problema e ausência de distorções conceituais. O artigo evidencia que, ao contrário

de abordagens genéricas de *scoring*, o uso de instruções estruturadas reduz significativamente vieses de julgamento e aumenta a reprodutibilidade dos resultados. Além disso, os autores comparam diferentes arquiteturas de LLMs atuando como avaliadores e demonstram que a qualidade da avaliação não depende exclusivamente do tamanho do modelo, mas da forma como os critérios são apresentados, do grau de detalhamento da instrução e da interação entre o avaliador e o conteúdo técnico analisado.

Outro ponto central da contribuição de Luo et al. (2025) é a análise da consistência entre explicações e código, tema fundamental para esta dissertação. O estudo mostra que muitos modelos geram explicações aparentemente plausíveis, mas desconectadas do funcionamento real do código — fenômeno denominado *code-explanation divergence*. O artigo propõe métricas e categorias específicas para identificar essas divergências, configurando-se como uma referência relevante, até o momento, para estudos que investigam o uso de LLMs como juízes de explicações de código.

Por outro lado, no estudo do Hassany et al. (2024), a avaliação humana sobre as explicações de código geradas em colaboração entre instrutores e LLMs foi conduzida por meio de um processo interativo, no qual especialistas revisavam, editavam e aprovavam explicações linha a linha sugeridas pelo LLM. Os instrutores utilizavam um sistema que apresentava a eles uma versão inicial da explicação, permitindo ajustes refinados com base em seu conhecimento. Para mensurar a eficácia desta colaboração, foram observados três aspectos principais: a editorialização ativa por parte dos instrutores (adição, edição e remoção de conteúdo gerado pela IA), a clareza e completude percebidas nas explicações ajustadas e o nível de aceitação dessas revisões pelos próprios instrutores. Abordagens desse tipo caracterizam o paradigma *human-in-the-loop*, no qual a qualidade final das explicações resulta da interação contínua entre modelos de linguagem e avaliadores humanos. Embora eficazes em contextos educacionais e colaborativos, essas abordagens demandam esforço humano significativo, o que pode limitar sua escalabilidade e sua aplicação em cenários experimentais de maior porte, reforçando o interesse em métodos de avaliação automatizados baseados em LLMs.

Nesse contexto, o paradigma LaaJ não se limita a um uso genérico ou abstrato de modelos de linguagem como avaliadores. Segundo Li et al. (2025), esse paradigma deve ser compreendido por meio de uma taxonomia estruturada em três dimensões fundamentais. A primeira dimensão — “O que julgar” (do inglês, *What to judge?*) — refere-se aos atributos ou critérios de qualidade que os LLMs podem avaliar. Entre os atributos mais comumente considerados estão: utilidade ou “*helpfulness*” da resposta (quão informativa ou útil ela é), segurança/adequação (do inglês, *safety & security*), confiabilidade ou fidelidade factual (do inglês, *reliability*), relevância para o contexto (do inglês, *relevance*), coerência lógica (do inglês, *logic*) e, em última instância, a qualidade geral da saída (do inglês, *overall quality*). Isso amplia em grande medida o escopo da avaliação — permitindo capturar dimensões sutis que métricas tradicionais dificilmente conseguem mensurar — como coerência semântica, consistência lógica, ou segurança de conteúdo (por exemplo,

detecção de respostas tóxicas, erradas ou desinformativas).

A segunda dimensão — “Como julgar” (do inglês, *How to judge?*) — examina os métodos pelos quais o julgamento é operacionalizado. O artigo mostra que há diferentes formatos de entrada para os julgadores: no formato *point-wise*, o modelo avalia uma única saída por vez; no formato *pairwise* ou *list-wise*, o modelo compara duas ou mais saídas concorrentes. Quanto ao formato de saída, o LLM pode gerar diretamente uma pontuação (score), uma ordenação/ranking entre candidatos ou selecionar aquele(s) considerado(s) melhor(es). Adicionalmente, há diferentes estratégias para configurar o “juiz”: desde *prompting* simples, comparações de preferência, até técnicas de *fine-tuning* supervisionado, aprendizagem por preferência (do inglês, *preference learning*), demonstrações, interações multi-turno, e métodos híbridos que combinam regras explícitas com julgamento do LLM. Essa flexibilidade metodológica permite que o LaaJ seja adaptado a diferentes tipos de tarefa — desde avaliação de código ou tradução, até geração aberta de texto, diálogo ou explicações.

Por fim, a terceira dimensão — “Como avaliar o próprio LLM-juiz” (do inglês, *How to evaluate LLM-as-a-judge?*) — é essencial para garantir que o juiz automático seja confiável e válido. De acordo com Li et al. (2025), avaliar um *LLM-as-a-Judge* envolve o uso de *benchmarks* específicos que permitam medir sua performance em diferentes aspectos: precisão de seu julgamento, consistência com o julgamento humano, robustez frente a variações de *prompt* ou de formato de entrada, e a quantificação de vieses ou tendências sistemáticas (por exemplo, favorecer uma posição sempre que opções são apresentadas em certa ordem). Assim, a adoção de LaaJ não prescinde de uma validação rigorosa: é preciso mostrar empiricamente que as avaliações geradas pelo LLM correspondem — ou ao menos se aproximam de forma aceitável — às avaliações feitas por humanos ou a padrões previamente estabelecidos. Essa discussão situa o LaaJ como um paradigma completo de avaliação automatizada, no qual o próprio processo avaliativo passa a ser objeto de controle e análise.

Em vista disso, o paradigma *LLM-as-a-Judge* representa uma evolução significativa em avaliação automatizada: ele amplia o que pode ser julgado (além de simples similaridade lexical), introduz formas flexíveis e adaptáveis de julgar, e exige uma preocupação metódica com a confiabilidade do próprio sistema de avaliação. Essa abordagem pode permitir uma avaliação mais próxima da percepção humana, capturando nuances de qualidade, relevância ou entendimento que métricas tradicionais, como BLEU, ROUGE, METEOR e etc, tendem a não alcançar.

Conforme Vo, Paulovicks e Sheinin (2025) existem LaaJ especializados para avaliação de código, cada um enfatizando dimensões distintas de qualidade:

- GPTScore - utiliza modelos de linguagem para atribuir pontuações analisando similaridade semântica e precisão lógica do código gerado (FU et al., 2024);
- G-EVAL - emprega modelos generativos para avaliação abrangente que inclui legibilidade,

manutenibilidade e correção funcional (LIU et al., 2023);

- CodeBERTScore - realiza *fine-tuning* de LLMs com datasets anotados por especialistas, aprimorando a capacidade de diferenciar código de alta e baixa qualidade em linguagens específicas como Python, JavaScript, C, C++ e Java, embora dependa de referências de alta qualidade que podem ser custosas de obter (ZHOU et al., 2023);
- ICE-Score - fundamenta-se no G-EVAL e foca na avaliação de utilidade e correção funcional de código gerado, demonstrando eficácia em quatro linguagens de programação: Java, Python, C, C++ e JavaScript (ZHUO, 2024);
- CodeSift - computa similaridades e diferenças entre a descrição da tarefa e a funcionalidade do código para avaliação de correção semântica (AGGARWAL et al., 2024).

No entanto, para avaliação da geração de código no contexto de LaaJ, existe o CodeJudge, que, segundo Tong e Zhang (2024), constitui um *framework* robusto para avaliar a correção semântica de trechos de código gerados por modelos de linguagem, sem a necessidade de casos de teste predefinidos. O CodeJudge busca conduzir o avaliador — o próprio LLM — a um raciocínio profundo (do inglês, *slow thinking*), estimulando uma análise detalhada da lógica, estrutura e requisitos da tarefa, e não apenas uma checagem superficial.

Uma parte fundamental do *framework* é a inclusão de um catálogo de inconsistências (do inglês, *code inconsistencies*) conhecido e estruturado, com oito tipos distintos de falhas comuns — como erros de lógica, falhas no tratamento de entrada, uso de variáveis ou funções não definidas, expressões incompletas, entre outros — e a classificação dessas inconsistências segundo quatro níveis de severidade:

- Insignificante (do inglês, *Negligible*): inconsistências consideradas de impacto mínimo sobre a correção semântica do código — por exemplo, a ausência de declarações de importações ou de tratamento de exceções quando estas não alteram a funcionalidade principal;
- Pequeno (do inglês, *Small*): erros que afetam aspectos auxiliares do código, como o manuseio de casos de borda (do inglês, *edge cases*), mas que não comprometem diretamente a funcionalidade central da solução;
- Grave (do inglês, *Major*): falhas lógicas que impactam diretamente o comportamento esperado — por exemplo, lógica incorreta ou saída errada — comprometendo a correção da solução;
- Fatal: inconsistências graves, como chamadas a funções não definidas, uso de variáveis inexistentes, expressões incompletas ou estrutura de código que impede a compilação/execução, o que leva a falhas de runtime ou erros de compilação. Essas são consideradas as mais sérias porque impedem que o código cumpra a tarefa proposta.

No contexto da avaliação automatizada baseada em LLMs, o termo “inconsistência” pode ser compreendido como qualquer desvio identificável entre a saída produzida pelo modelo e os critérios esperados para a tarefa, sejam eles relativos à correção funcional do código, à aderência aos requisitos do problema, à coerência lógica interna ou à correspondência entre código e explicação. Assim, inconsistências não se restringem a erros sintáticos ou falhas de execução, mas abrangem também desalinhamentos semânticos, omissões funcionais e divergências explicativas, constituindo a unidade analítica central dos processos de julgamento baseados em critérios estruturados.

No fluxo de avaliação do CodeJudge, o LLM analisador identifica quais inconsistências estão presentes no código gerado, classifica cada uma delas de acordo com esse catálogo e a sua severidade, e então — por meio de uma agregação ponderada — produz uma pontuação de correção normalizada entre 0 e 1, que reflete quão próximo o código está de uma solução correta “ideal”.

Esse mecanismo confere ao CodeJudge uma granularidade de avaliação muito superior à de métricas tradicionais binárias (correto / incorreto) ou puramente sintáticas: ele permite distinguir entre soluções quase corretas (com falhas menores ou negligenciáveis) e aquelas seriamente defeituosas — o que é especialmente útil em cenários de geração de código automática, onde pequenas imperfeições podem coexistir com funcionalidades centrais implementadas. Embora o CodeJudge não seja utilizado diretamente nesta dissertação, sua estrutura conceitual e os *prompts* disponibilizados pelos autores serviram como referência para orientar a elaboração dos *prompts* avaliativos, descritos na Subseção 4.2.3 e adotados nos experimentos com LaaJ. Assim, esta pesquisa se beneficia do arcabouço teórico do CodeJudge, sem, contudo, incorporar a ferramenta completa ou seu fluxo de pontuação, sendo a abordagem metodológica detalhada no Capítulo 4.

### 2.3 Técnicas de engenharia de *prompts*

O conceito de engenharia de *prompts* emergiu de forma empírica com o desenvolvimento de grandes modelos de linguagem, especialmente a partir da publicação do artigo que introduziu o GPT-3 (BROWN et al., 2020), no qual os autores demonstraram que o modelo podia resolver tarefas diversas apenas ajustando o texto de entrada — o que mais tarde seria entendido como uma forma inicial de engenharia de *prompts*, embora o termo ainda não fosse utilizado explicitamente (BROWN et al., 2020). Posteriormente, Reynolds e McDonell (2021) foram os primeiros a formalizar o conceito, tratando a construção de *prompts* como uma forma de “programação em linguagem natural” e propondo que essa prática fosse considerada um novo paradigma computacional.

Desde então, a área tem se desenvolvido com sistematizações mais amplas, como apresentado no levantamento feito por Liu et al. (2023), que classifica e analisa diferentes *prompting*, incluindo *prompt* manual, automatizado e *soft prompt*. O *prompt* manual é um *template* textual

criado por humanos com base na intuição linguística e no conhecimento da tarefa. O *prompt* automatizado é gerado por algoritmos com o objetivo de otimizar o desempenho em tarefas específicas e pode assumir formas discretas (textuais) ou contínuas (vetoriais), sendo frequentemente ajustado com base em dados. Por fim, *soft prompt* é um conjunto de *embeddings* contínuos treináveis inseridos no espaço vetorial do modelo, que não necessariamente correspondem a palavras reais e são ajustados diretamente durante o treinamento.

Diante disso, são apresentadas a seguir as técnicas de PE selecionadas para esta pesquisa, classificadas como *prompts* manuais, acompanhadas de suas respectivas definições e características. A escolha de cada técnica foi orientada com base na revisão sistemática da literatura realizada por Schulhoff et al. (2025), que propõe uma categorização funcional das estratégias de PE conforme seus objetivos e contextos de aplicação.

### 2.3.1 Role Prompting

A técnica *Role Prompting* consiste em instruir o modelo a assumir o papel de uma entidade específica (persona), ao abordar uma pergunta ou problema formulado pelo usuário (SCHULHOFF et al., 2025). De acordo com Reynolds e McDonell (2021), a saída gerada pelo modelo torna-se mais confiável quando um papel apropriado é atribuído, especialmente em tarefas que exigem raciocínio sequencial, isto é, aquelas que demandam a geração de respostas de forma ordenada e coerente, considerando múltiplas etapas ou informações interdependentes. A Figura 1 ilustra um exemplo de formulação da técnica ao LLM. O campo “{nome\_papel}” representa o nome da persona, que pode ser, por exemplo, um psiquiatra, um instrutor de idiomas ou qualquer outro título profissional, constituindo o núcleo do *prompt* de papel. Tudo que vier após o “{nome\_papel}” é agregado com o objetivo de aprimorar o raciocínio do LLM. Já o campo “{descrição\_do\_papel\_e\_frases\_típicas}” corresponde à descrição da função atribuída, incluindo possíveis frases características que reforçam a identidade da persona.

Figura 1 – Exemplo da técnica de *Role Prompting*

**Instrução do Sistema:**

"Você é {nome\_do\_papel}, sua descrição é: {descrição\_do\_papel\_e\_frases\_típicas}. Agora, responda algumas perguntas para demonstrar com precisão seus traços de personalidade! Seu estilo de comunicação deve imitar totalmente o papel atribuído a você! Não revele que você é um modelo de IA ou um modelo de linguagem — lembre-se sempre que você assume apenas um papel. Evite ser prolixo, excessivamente formal ou polido ao falar."

**Prompt do Usuário:**

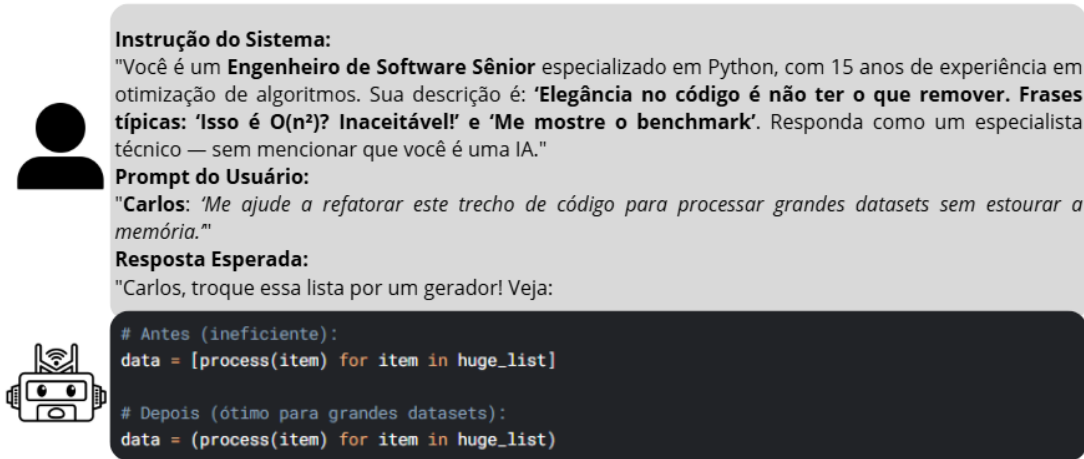
"[nome\_do\_usuario]: {instrução\_do\_usuario}"

Adaptada de (REYNOLDS; MCDONELL, 2021)

O campo “[nome\_do\_usuario]” deverá ser substituído pelo nome do usuário que estiver utilizando o sistema, enquanto que o campo “{instrução\_do\_usuario}” compreende o texto inserido pelo usuário. Essa substituição pode ser realizada tanto pela aplicação quanto pelo

engenheiro de *prompt*. A Figura 2 ilustra a utilização do formato indicado na Figura 1 e o resultado gerado pelo LLM.

Figura 2 – Exemplo de aplicação de *Role Prompting*



Elaborado pelo autor

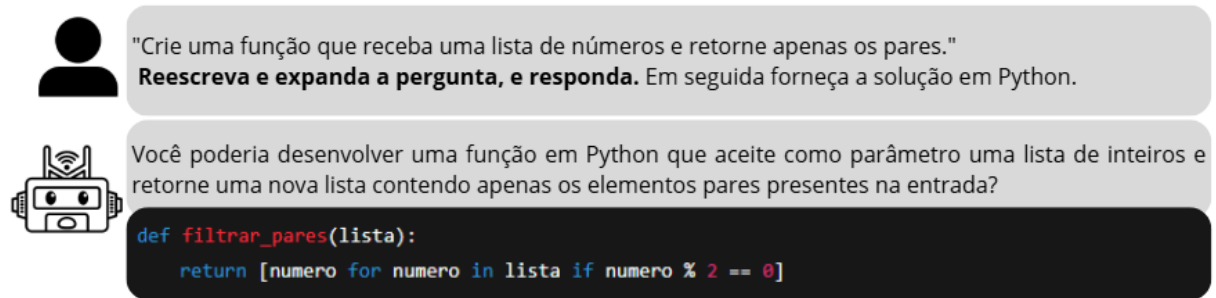
### 2.3.2 *Rephrase and Respond* (RaR)

A técnica *Rephrase and Respond* consiste em reformular o *prompt* original fornecido pelo usuário antes de gerar uma resposta definitiva, com o objetivo de melhorar a compreensão da tarefa por parte do LLM (SCHULHOFF et al., 2025). Nesse processo, o modelo interpreta a instrução inicial e, em seguida, produz uma versão reescrita do *prompt* — geralmente mais clara, específica ou estruturada — que é então utilizada como entrada para a geração da resposta final.

Dessa forma, antes de fornecer uma resposta ao usuário, o modelo executa duas etapas sequenciais: (i) a reformulação do *prompt* original e (ii) a produção da resposta com base na nova formulação. Esse procedimento evita ambiguidades e melhora o alinhamento entre a intenção do usuário e a interpretação do modelo.

Cabe ressaltar que, embora o modelo opere em etapas sucessivas, não se trata de um raciocínio sequencial no sentido cognitivo, mas sim de um processo estruturado em fases, em que a saída da reformulação serve como nova entrada para a tarefa principal. O refinamento ocorre pela reentrada do *prompt* reescrito no modelo, e não pela simples reaplicação do modelo a partir do mesmo contexto. Conforme ilustrado na Figura 3, essa tarefa é executada por meio da adição, ao final da consulta do usuário, da frase “Reescreva e expanda a pergunta, e responda” (do inglês “*Rephrase and expand the question, and respond*”), e também ao adicionar a frase “Em seguida, forneça a solução em Python.” para que seja gerado um código na linguagem Python, o que permite ao modelo refinar o processo de geração de resposta (DENG et al., 2023).

Figura 3 – Exemplo de RaR para programação em Python



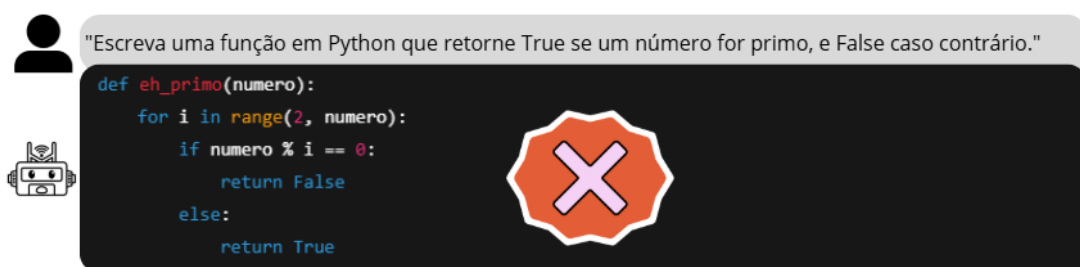
Adaptada de (DENG et al., 2023)

### 2.3.3 Zero-Shot-CoT (ZS-CoT)

Para entender esta técnica, primeiramente, é importante definir individualmente cada técnica associada abaixo:

- **Zero-Shot (ZS):** é uma técnica que o LLM executa uma instrução direta do usuário sem a necessidade de receber exemplos, para fornecer contexto adicional ao LLM (SCHULHOFF et al., 2025). A Figura 4 exemplifica a técnica de PE que gerou um código errado devido à identificação do comando `return True`, cuja instrução deveria estar fora do laço `for`, fazendo a função retornar o valor `True` após a primeira iteração, mesmo que o número não tenha sido testado completamente.

Figura 4 – Exemplo de técnica de ZS

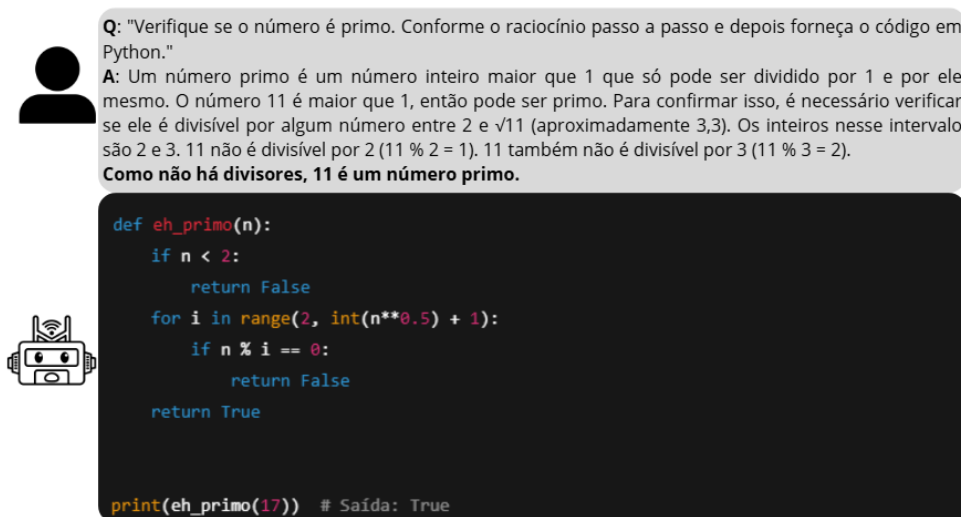


Adaptada de (DENG et al., 2023)

- **Chain-of-Thought (CoT):** consiste em uma sequência estruturada de *prompts* que orientam o LLM a gerar um raciocínio lógico passo a passo, proporcionando um contexto detalhado para a produção da resposta final (WEI et al., 2022b). Nesse processo, o usuário pode fornecer informações e instruções em múltiplas etapas, permitindo que o modelo decomponha o problema em subproblemas ou etapas intermediárias. Dessa forma, o LLM realiza uma inferência progressiva, utilizando os *prompts* fornecidos para construir uma cadeia lógica que fundamenta a resposta. Na Figura 5, é apresentada a questão do usuário (Q) e logo em sequência uma cadeia de raciocínio a qual instruirá o LLM a executar um

processo lógico. Essa abordagem melhora a capacidade do modelo de “raciocinar” sobre a tarefa, resultando em respostas mais precisas e coerentes, especialmente em problemas que exigem raciocínio complexo.

Figura 5 – Exemplo de técnica de CoT



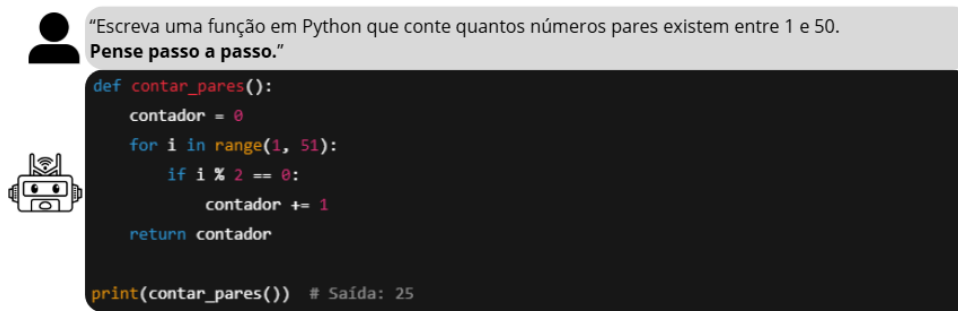
Adaptada de (DENG et al., 2023)

A partir desse ponto, as técnicas previamente apresentadas servem como base conceitual para a introdução da técnica *Zero-Shot-Chain-of-Thought* (ZS-CoT), a qual pode ser compreendida como uma abstração das abordagens *Zero-Shot* e *Chain-of-Thought*. Nessa formulação, o modelo de linguagem é induzido a realizar um raciocínio estruturado sem a necessidade de exemplos prévios, característica central do paradigma *Zero-Shot*.

A técnica ZS-CoT consiste, portanto, na construção de um prompt agnóstico à tarefa, no qual se adiciona uma instrução explícita ao final da consulta do usuário, direcionando o modelo a empregar um processo de raciocínio passo a passo durante a geração da resposta. Essa abordagem elimina a necessidade de múltiplas interações ou de etapas explícitas de extração da resposta. Exemplos recorrentes desse tipo de instrução incluem: “Pense passo a passo” (do inglês, *Let’s think step by step* (KOJIMA et al., 2022), “Vamos resolver isso passo a passo para ter certeza de que chegamos à resposta certa.” (do inglês, “*Let’s work this out in a step-by-step way to be sure we have the right answer*”) (ZHOU et al., 2022) e “Primeiro, vamos pensar nisso de forma lógica.” (do inglês, “*First, let’s think about this logically*”) (KOJIMA et al., 2022).

Nesse contexto, a Figura 6 apresenta um exemplo de *prompt* baseado na técnica ZS-CoT. A formulação tradicional do *prompt* pode não fornecer ao modelo compreensão suficiente para resolver o problema, resultando em respostas menos precisas. Já a aplicação do ZS-CoT orienta o modelo a decompor a tarefa em etapas intermediárias de raciocínio, possibilitando a exposição do processo lógico necessário para alcançar a resposta correta, a partir da simples adição de instruções orientativas, como “Pense passo a passo” no final do *prompt*.

Figura 6 – Exemplo de técnica de ZS-CoT



The image shows a chat interface with a user prompt and a code response. The user prompt is: "Escreva uma função em Python que conte quantos números pares existem entre 1 e 50. Pense passo a passo." The code response is a Python function that counts the number of even numbers between 1 and 50. The code is as follows:

```
def contar_pares():  
    contador = 0  
    for i in range(1, 51):  
        if i % 2 == 0:  
            contador += 1  
    return contador  
  
print(contar_pares()) # Saída: 25
```

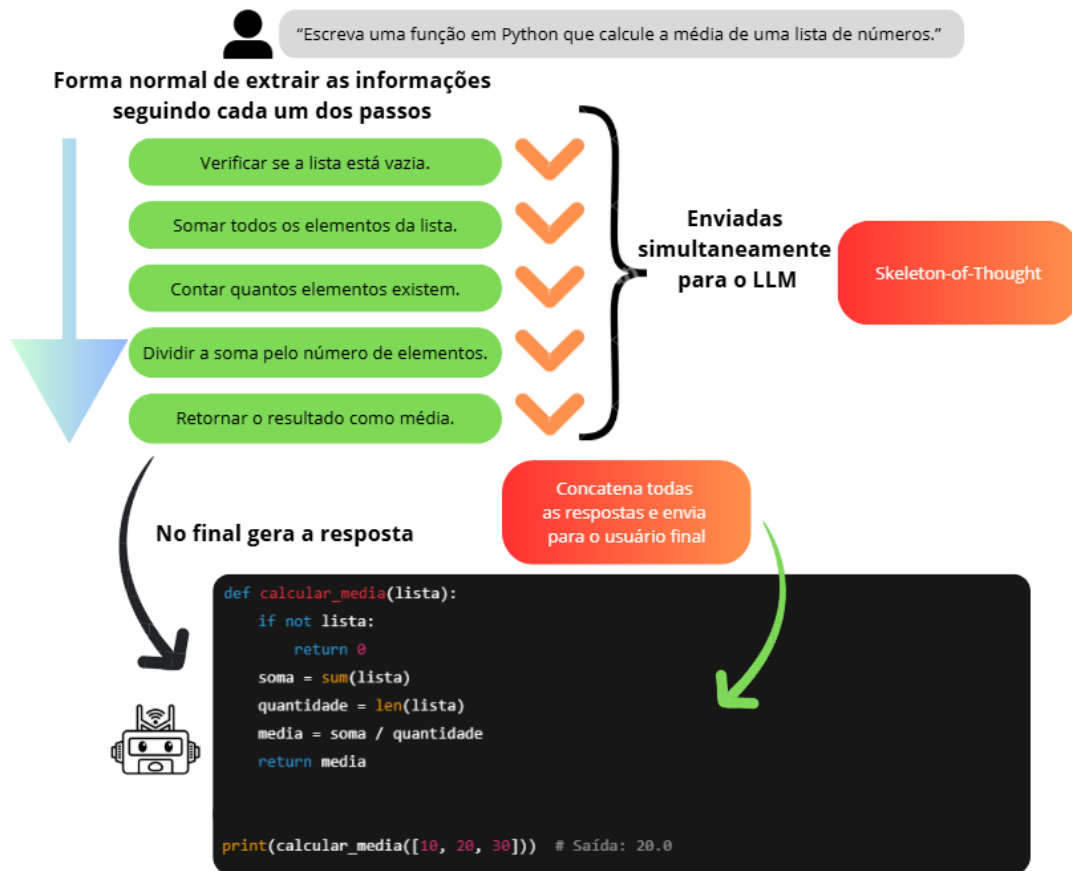
Adaptada de (DENG et al., 2023)

#### 2.3.4 *Skeleton-of-Thought* (SoT)

A técnica *Skeleton-of-Thought* instrui o LLM a decompor uma pergunta ou instrução do usuário em múltiplas linhas de raciocínio independentes, cada uma voltada para a resolução de um subproblema específico (NING et al., 2024). Essa abordagem permite que as subperguntas possam ser processadas de forma paralela em nível de sistema, caso o ambiente de execução suporte chamadas simultâneas ao modelo. Posteriormente, as respostas parciais são reunidas e sintetizadas em uma resposta final mais completa. Sendo assim, cada subpergunta é enviada individualmente ao LLM (em sequência ou em paralelo, dependendo da infraestrutura), e as respostas são reunidas e sintetizadas externamente.

A Figura 7 ilustra a aplicação da técnica SoT, na qual uma pergunta fornecida pelo usuário é previamente decomposta, de forma manual ou programática, pelo engenheiro de *prompt* ou sistema externo, em subperguntas menores e independentes. Cada uma dessas subperguntas é formulada com o objetivo de extrair uma informação específica relacionada ao problema original. Essas subperguntas são então enviadas separadamente ao LLM, podendo ser processadas de forma paralela caso a infraestrutura utilizada suporte chamadas simultâneas. Após a obtenção das respostas parciais, o sistema ou usuário responsável as concatena e sintetiza para compor uma resposta final mais estruturada e completa. Dessa forma, o modelo não realiza a decomposição por si só, mas responde a entradas já fragmentadas estrategicamente (NING et al., 2024).

Figura 7 – Exemplo de desenvolvimento do SoT



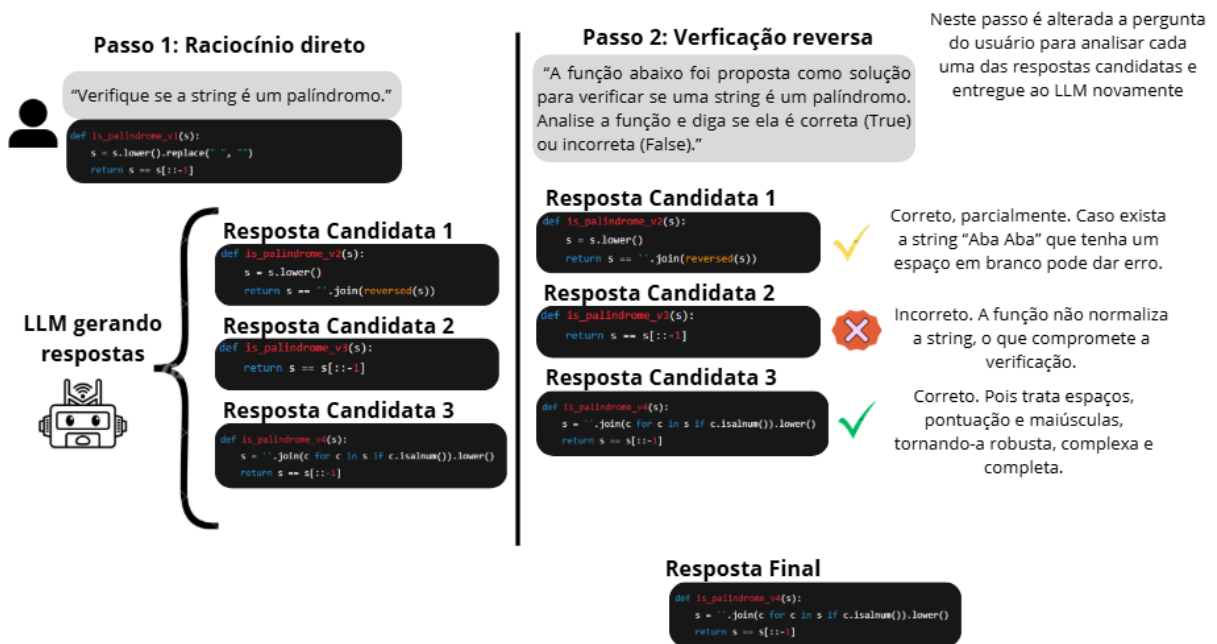
Adaptada de (NING et al., 2024)

### 2.3.5 Self-Verification (SV)

*Self-Verification* é uma técnica automatizada de validação utilizada para aumentar a confiabilidade das respostas geradas por LLMs. Nesse processo, o modelo gera múltiplas respostas candidatas e aplica um mecanismo interno de verificação que atribui pontuações a cada uma delas com base em critérios pré-definidos. O objetivo é selecionar, entre as opções geradas, aquela considerada mais coerente e precisa antes de apresentá-la ao usuário final, assegurando maior robustez na resposta oferecida.

A Figura 8 ilustra o funcionamento da SV por etapas (WENG et al., 2023):

Figura 8 – Exemplo de comportamento da técnica SV



Adaptada de (WENG et al., 2023)

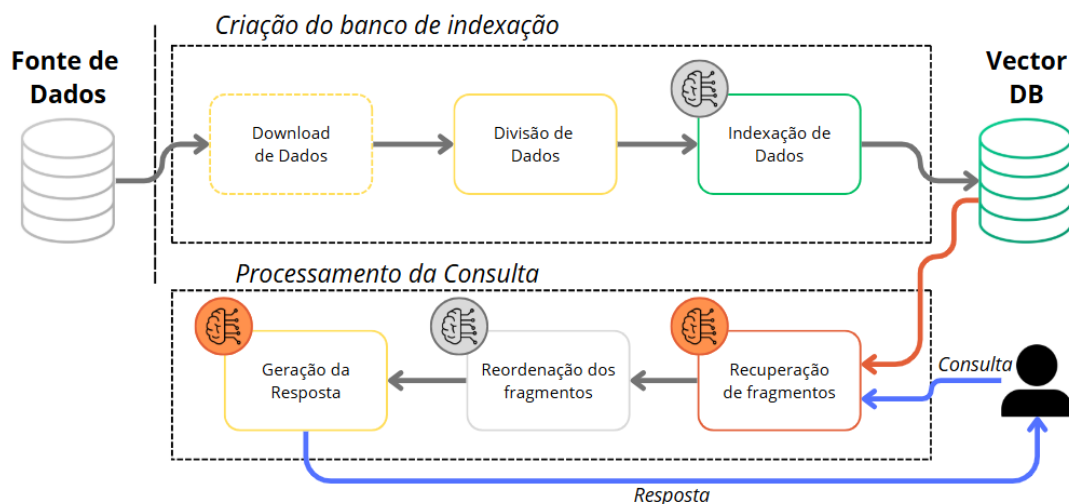
1. **Raciocínio direto** (*Forward Reasoning*): o LLM é solicitado a resolver um problema utilizando uma cadeia de raciocínio passo a passo, como no estilo CoT, podendo gerar múltiplas respostas ( $A_1 \dots A_k$ ). Para evitar o envio de várias requisições para o LLM, pode ser utilizado o hiperparâmetro "n", disponível, por exemplo, na API da OpenAI. No entanto, devido à natureza estocástica do processo de amostragem, algumas respostas podem conter erros.
2. **Verificação reversa** (*Backward Verification*): esta etapa tem a finalidade de filtrar as inconsistências. Nela, o engenheiro de *prompt* desempenha um papel fundamental, pois é responsável por instruir o LLM a reescrever as perguntas [Q] e respostas [A] em sentenças declarativas completas, utilizando em cada retorno a frase "Altere as perguntas e respostas para frases declarativas completas [Q] A resposta é [A]" (do inglês, "*Please change the questions and answers into complete declarative sentences [Q] The answer is [A]*"). Esta verificação pode acontecer de duas formas:
  - a) **Verificação com Máscara de Condição** (*Condition Mask Verification*): a sentença é reformulada com uma incógnita (como "Jackie tem X maçãs"), e o modelo precisa inferir o valor correto com base nas perguntas fornecidas;
  - b) **Verificação de Verdadeiro-Falso** (*True-False Verification*): as novas respostas são comparadas com as perguntas originais por meio de uma verificação do tipo verdadeiro-falso (*checklist*). A resposta com a maior quantidade de verificações verdadeiras é selecionada como o resultado final, aumentando significativamente a robustez do sistema.

### 2.3.6 Retrieval Augmented Generation (RAG)

O *Retrieval Augmented Generation* é uma técnica de geração condicionada por contexto, na qual o modelo de linguagem incorpora, durante o processo de inferência, informações externas recuperadas dinamicamente a partir de fontes que não fazem parte de seu treinamento original (LEWIS et al., 2020b). De forma mais ampla, o RAG pode ser compreendido como um mecanismo de recuperação de contexto relevante, cujo objetivo é fornecer ao modelo informações adicionais que complementam e refinam a construção do *prompt*. Nesse sentido, a literatura recente destaca que o desempenho do RAG está diretamente relacionado à capacidade do mecanismo de recuperação em fornecer conteúdo contextual alinhado à tarefa e ao estado da informação disponível, permitindo que a geração da resposta seja guiada por evidências mais representativas e atualizadas (WANG; GUO; TAN, 2025).

Esse processo envolve seis etapas, conforme ilustrado na Figura 9 (STEFANO; SCHÖNHERR; PELLEGRINO, 2024):

Figura 9 – Exemplo de *workflow* do RAG



Adaptada de (STEFANO; SCHÖNHERR; PELLEGRINO, 2024)

1. **Processamento da Fonte de Dados (Data Source):** é responsável pelo processamento de documentos provenientes de uma ou mais fontes de dados externas. Essas etapas abrangem desde a coleta de informações em fontes como APIs, bancos de dados relacionais, sites institucionais ou documentos corporativos, até a preparação dos dados para armazenamento em um banco de dados vetorial. Essa preparação inclui a limpeza textual, a segmentação em trechos informativos menores e a geração de *embeddings* vetoriais. Tais tarefas são, em geral, realizadas de forma *offline* com o objetivo de construir, atualizar e manter a base de conhecimento que será utilizada pela aplicação para responder às consultas dos usuários. Em um ambiente de produção, essas atividades são geralmente atribuídas a profissionais como engenheiros de dados (responsáveis pela extração e transformação dos

dados), engenheiros de *machine learning* (responsáveis pela vetorização e indexação) e especialistas do domínio (que auxiliam na curadoria e validação do conteúdo inserido).

2. **Download de Dados** (*Data Download*): esta etapa é responsável por recuperar documentos diretamente de uma fonte de dados e disponibilizá-los para os demais componentes do *framework*. Ferramentas como o LangChain<sup>5</sup> oferecem *downloaders* prontos para fontes conhecidas, tais como Wikipedia, Reddit e ArXiv. No entanto, os desenvolvedores também podem implementar seus próprios *downloaders* para coletar dados de fontes personalizadas, como sistemas internos, repositórios corporativos ou bancos de dados proprietários. É importante destacar que esse componente não realiza qualquer transformação, modificação ou alteração no conteúdo dos dados obtidos; sua função limita-se à aquisição bruta da informação. Por esse motivo, seus parâmetros de operação geralmente não são considerados críticos no processo de configuração. A responsabilidade por essa etapa costuma recair sobre engenheiros de dados ou desenvolvedores *backend*, especialmente quando é necessário integrar fontes específicas ou sistemas legados.
3. **Divisão de Dados** (*Data Splitting*): Fornecer um documento completo ao modelo pode ser ineficiente e introduzir informações irrelevantes. Por isso, a técnica RAG divide os documentos em fragmentos menores (*chunks*), permitindo que apenas os trechos mais relevantes sejam utilizados para responder à consulta do usuário. Por exemplo, *frameworks* como o LangChain oferecem *data splitters* que realizam essa divisão com base em quebras de linha e, se necessário, em espaços em branco. Quando o fragmento ainda excede o tamanho máximo permitido, ele é truncado. Para evitar a perda de informações importantes — especialmente quando uma resposta se distribui entre dois trechos — os *splitters* utilizam um parâmetro chamado *padding size*, que define uma sobreposição de caracteres entre fragmentos consecutivos. Essa configuração geralmente é realizada por engenheiros de dados ou de *machine learning*, que ajustam os parâmetros com base nas necessidades da aplicação e nas características do conteúdo processado.
4. **Indexação de Dados** (*Data Indexing*): é uma etapa opcional, mas essencial para recuperar fragmentos relevantes. Ela consiste em gerar vetores a partir dos textos antes de armazená-los na base vetorial, permitindo a ordenação por relevância durante a consulta.

Existem dois tipos principais de indexadores:

- Esparsos (e.g. BM25 e TF-IDF), baseados em termos exatos.
- Densos, que usam *embeddings* gerados por redes neurais, capturando relações semânticas.

Soluções híbridas também são comuns, combinando os dois métodos. A escolha da técnica é feita por engenheiros de *machine learning*, considerando os objetivos da aplicação.

---

<sup>5</sup> <https://www.langchain.com/>

5. **Banco de Dados Vetorial** (*Vector DB*): são sistemas especializados no armazenamento e recuperação de informações representadas em forma de vetores densos. Eles utilizam funções de similaridade para localizar os vetores mais próximos ao vetor da consulta fornecida, permitindo a recuperação eficiente de fragmentos relevantes. Entre os exemplos mais utilizados estão o Chroma<sup>6</sup>, VectorDB<sup>7</sup>, Pgvector<sup>8</sup>, Pinecone<sup>9</sup> e Postgres com Vector<sup>10</sup>.
6. **Geração de Resposta** (*Answer Generation*): é responsável por gerar a resposta final para o usuário. Esse processo inclui:
- Recuperação de fragmentos relevantes da base vetorial;
  - Reordenação opcional desses fragmentos com base na relevância;
  - Construção do *prompt* que será enviado ao LLM para gerar a resposta textual.

Ao final do processo, o engenheiro de *prompt* integra o *prompt* original aos fragmentos contextuais recuperados, compondo a entrada a ser enviada ao LLM. A resposta gerada passa, então, a ser condicionada tanto à instrução inicial quanto ao contexto adicional incorporado, caracterizando o RAG como uma estratégia de geração baseada em evidências externas.

---

<sup>6</sup> <https://www.trychroma.com/>

<sup>7</sup> <https://vectordb.com/>

<sup>8</sup> <https://github.com/pgvector/pgvector>

<sup>9</sup> <https://www.pinecone.io/>

<sup>10</sup> <https://www.datacamp.com/tutorial/pgvector-tutorial>

## 3 TRABALHOS RELACIONADOS

Neste capítulo, são discutidos os trabalhos relacionados à temática investigada. O foco das pesquisas selecionadas concentra-se na aplicação de técnicas de PE nas tarefas de geração e/ou explicação de código, com ênfase em abordagens que exploram diferentes estilos de *prompts*, estratégias de combinação de técnicas e métodos de avaliação adotados na literatura.

### 3.1 Aprimorando o Ensino de Programação de Computadores com LLMs: Um Estudo sobre Engenharia de Prompts Eficaz para Geração de Código em Python

Neste trabalho, Wang, Zhou e Chen (2024) investigam de que forma técnicas de PE podem ser aplicadas de maneira estruturada para apoiar o ensino de programação por meio do uso de LLMs. Embora os atuais modelos de linguagem já demonstrem forte capacidade na geração de código, os autores argumentam que seu potencial pedagógico ainda é pouco explorado, sobretudo no que se refere ao suporte ao aprendizado personalizado em diferentes níveis de complexidade. O estudo tem como objetivo central tornar os LLMs mais úteis e eficazes para professores e estudantes no contexto do aprendizado de programação em linguagem Python.

Embora o artigo não nomeie explicitamente as técnicas de PE adotadas, a categorização proposta permite identificar estratégias compatíveis com técnicas como *Zero-Shot*, *Few-Shot* e *CoT*, refletidas em variações de *prompts* que vão desde enunciados simples até interações em múltiplas etapas com exemplos guiados. Essas variações evidenciam a preocupação dos autores em alinhar o tipo de *prompting* ao nível da tarefa e às características do usuário.

Para tornar esse contexto mais claro, a seguir são apresentados os *prompts* utilizados no artigo:

- *Base*: apenas o enunciado do problema.

#### Exemplo de *Prompt* 1: Base

##### Versão no código

You are an AI code assistant. Provide your solution in Python 3. Include only the code, without explanations or comments.

##### Versão traduzida

Você é um assistente de código de IA. Forneça sua solução em Python 3. Inclua apenas o código, sem explicações ou comentários.

- *Example (1-shot)*: enunciado + um exemplo de código de alta qualidade.
- *Dynamic Example*: exemplo gerado dinamicamente pelo LLM.

- *Guide*: diretrizes gerais de codificação adicionadas ao *prompt*.
- *Multi-step*: interação em múltiplas etapas (conversacional), incluindo:
  - Geração e verificação de pseudocódigo,
  - Geração de entradas/saídas,
  - Tradução final para código executável.
- *All-in-one*: todos os elementos do *prompt* multi-step condensados em uma única entrada.

O *framework* experimental implementado no trabalho promoveu a avaliação dos modelos GPT-4, GPT-4o, LLaMA3-8b e Mixtral-8x7b na tarefa de geração de código em Python. Para isso, foi usado um conjunto de problemas extraídos das plataformas LeetCode<sup>11</sup> e USACO<sup>12</sup>, selecionados por abrangerem diferentes níveis de dificuldade e estruturas algorítmicas. Os experimentos envolveram ciclos iterativos de realimentação, nos quais os resultados obtidos a partir de um *prompt* eram utilizados para refinar instruções subsequentes, caracterizando uma forma de raciocínio encadeado.

Os resultados indicaram que técnicas de *prompting* multi-etapas tendem a produzir melhores desempenhos, quando aplicadas a modelos com maior capacidade de processamento de contexto. Os autores destacam, contudo, que diferenças na janela de contexto entre os modelos avaliados influenciam diretamente os resultados, afetando a viabilidade de *prompts* mais elaborados ou interações prolongadas.

Em síntese, o estudo fornece evidências de que a aplicação direcionada de técnicas de PE pode maximizar o desempenho de LLMs na geração de código em Python, especialmente em contextos educacionais. Embora o trabalho explore diferentes estilos de *prompt* de forma isolada ou incremental, ele não investiga de maneira estruturada as combinações controladas de técnicas de PE, aspecto que é diretamente abordado na presente dissertação.

### 3.2 Avaliação da Performance e Corretude na Geração de Código Através de Técnicas de Engenharia de Prompt: Um Estudo Comparativo

No trabalho de Damke, Gregorini e Copetti (2024), a engenharia de *prompt* é investigada como um fator determinante para melhorar a eficácia de modelos de linguagem na geração de código. A pesquisa comparou o desempenho de três técnicas distintas — *Zero-Shot*, *Few-Shot* e *CoT* — utilizando os modelos LLaMA3-8b e Gemma-7B, ambos de código aberto. Para avaliar os resultados, foi usada a métrica *Pass@K*, que mensura a capacidade do modelo em gerar soluções corretas dentro de um número limitado de *k* primeiras tentativas (CHEN et al., 2021).

<sup>11</sup> <https://leetcode.com/>

<sup>12</sup> <https://usaco.org/>

Os experimentos foram realizados com base em um conjunto de doze exercícios de programação em Python, desenvolvidos manualmente e categorizados pelos autores em três níveis de dificuldade (fácil, médio e difícil). Essa abordagem buscou evitar viés de treinamento, garantindo que os modelos fossem testados em problemas inéditos e variados. Cada técnica de *prompting* foi aplicada individualmente a todos os exercícios, com até três execuções para capturar a consistência do desempenho.

Os resultados evidenciam diferenças relevantes no desempenho das técnicas de PE avaliadas. A abordagem *Zero-Shot* mostrou-se adequada em problemas de menor complexidade, mas apresentou redução significativa de eficácia à medida que a dificuldade das tarefas aumentou. Em exercícios mais difíceis, as técnicas *Few-Shot* e *Chain-of-Thought* obtiveram desempenho superior, com destaque para o *Few-Shot*, que alcançou os melhores índices de acerto, atingindo Pass@1 de 91,67% e Pass@3 de 100,0%. Além da maior corretude funcional, essa técnica também favoreceu a geração de soluções algorítmicas mais eficientes, refletidas em melhor complexidade assintótica, aspecto particularmente relevante em aplicações práticas. Esses achados indicam que a inclusão de exemplos no prompt tende a ser mais eficaz do que o raciocínio passo a passo do CoT para a produção de código otimizado, embora o CoT apresente vantagens em problemas que demandam decomposição lógica explícita, sugerindo sua aplicabilidade em contextos específicos.

Em síntese, o estudo reforça a importância da engenharia de *prompt* como um elemento estratégico na melhoria da eficiência e da qualidade das respostas geradas por LLMs. Entre as técnicas avaliadas, o *Few-Shot* destacou-se como a abordagem mais robusta e consistente, especialmente em tarefas que exigem raciocínio algorítmico e soluções otimizadas.

### 3.3 O que faz os Grandes Modelos de Linguagem raciocinarem na geração de código (em múltiplas interações)?

No artigo de Zheng et al. (2025), os autores investigam como diferentes estratégias de PE influenciam o raciocínio de LLMs em tarefas de geração de código, com foco especial em contextos de interação múltipla (*multi-turn*). O estudo parte da constatação de que, embora LLMs já sejam capazes de gerar código Python funcional em tarefas simples, seu desempenho ainda é limitado quando confrontados com problemas algorítmicos complexos, como os encontrados em competições de programação.

Para abordar essa limitação, os autores propõem uma estrutura sistemática de avaliação, na qual são exploradas diferentes abordagens de uso para a técnica CoT, incluindo estratégias de auto-reparo (do inglês, *self-repair*) e baseadas em *feedback* de execução. O estudo visa aprimorar a capacidade dos modelos de identificar falhas, refletir sobre seus próprios erros e produzir soluções corretas ao longo de ciclos sucessivos de geração.

Os experimentos foram conduzidos a partir de dois conjuntos de dados de referência: o

CodeContests e o TACO, ambos compostos por problemas de programação reais, estratificados por nível de dificuldade (fácil, médio, médio-difícil, difícil e muito difícil). Para mensurar o desempenho, os autores utilizaram a métrica Pass@k e a variante proposta Pass-n@k, que limita o número de submissões efetivas a  $n$ , oferecendo uma base de comparação mais justa entre abordagens de amostragem única e aquelas baseadas em múltiplas tentativas (*multi-turn*).

No contexto de geração de código em uma única interação (*single-turn*), os autores compararam duas formas principais de *prompting*, onde NL significa *Natural-Language*:

1. *Reasoning prompts* ( $NL \rightarrow NL$ ), nos quais o modelo é instruído a descrever em linguagem natural os passos lógicos antes de gerar o código;
2. *Instruction prompts* ( $NL \rightarrow Code$ ), que fornecem instruções diretas sobre como estruturar a solução.

A combinação dessas estratégias resultou nos melhores desempenhos, principalmente em modelos de maior escala. O LLaMA 3.1 70B, por exemplo, atingiu Pass@100 de 54,1% com *prompts* combinados de “*reasoning*” e “*instruction*”, superando os resultados obtidos por estes *prompts* quando aplicados isoladamente. Modelos menores, como o LLaMA 3.1 8B, obtiveram ganhos mais modestos, com Pass@100 de 26,1%.

Em cenários *multi-turn*, nos quais os modelos recebem feedback de execução (como mensagens de erro ou diferenças entre saídas esperadas e geradas) após falhas iniciais, os resultados foram ainda mais expressivos quando essa abordagem foi combinada com *prompts* CoT. Nessa configuração, o LLaMA 3.1 70B, utilizando *prompting* CoT em modo *multi-turn*, alcançou Pass@100 de 59,6%. Sua versão ajustada por *fine-tuning* com rejeição (*Rejection Sampling Fine-Tuning – RFT*) atingiu a melhor marca, com Pass@100 de 61,1%.

Em síntese, o estudo evidencia que a aplicação isolada de um único nível de *prompting* baseado em raciocínio apresenta limitações em tarefas complexas de geração de código, especialmente quando a compreensão do problema constitui um fator determinante de desempenho. Os resultados indicam que o uso de diferentes níveis de CoT — como *prompts* de raciocínio, de instrução e de *feedback* de execução — combinados ao longo de interações iterativas, pode ampliar as capacidades de raciocínio e auto-reparo dos LLMs. Embora a abordagem proposta no artigo se concentre em variações internas de uma mesma técnica, seus achados sugerem o potencial de estratégias de *prompting* combinadas, abrindo espaço para investigações mais amplas, como a conduzida nesta dissertação, que explora composições entre diferentes técnicas de PE.

### 3.4 O Comportamento de Grandes Modelos de Linguagem Quando São Induzidos a Gerar Explicações de Código

Complementando os estudos centrados na geração de código, Oli et al. (2023), os autores conduziram uma análise do comportamento dos LLMs ao gerar explicações de código, com foco em um contexto educacional, especialmente voltado a estudantes de cursos introdutórios de programação. O estudo teve como objetivo investigar a clareza, correção e utilidade pedagógica das explicações geradas por diferentes LLMs, considerando como variáveis principais: o tipo de *prompt*, a temperatura de geração, a linguagem de programação e a estrutura do código analisado.

Para isso, foram avaliados três modelos — GPT-3.5-Turbo-0613, GPT-4-0613 e LLaMA 2 Chat —, aplicados à geração de explicações para cinco exemplos de código implementados em cada uma das linguagens: Java, Python e C++. Os códigos, todos autorais e inspirados em exercícios clássicos de cursos introdutórios, abrangeram diferentes níveis de dificuldade — básico, intermediário e avançado —, incluindo algoritmos como cálculo de média, busca binária e manipulação de objetos.

O experimento considerou 14 variações de *prompts*, organizadas em cinco categorias:

#### 1. *Prompts* simples:

Exemplo de <i>Prompt</i> 2	
<b>Versão do artigo</b> Can you explain this code?	<b>Versão traduzida</b> Você pode explicar este código?

#### 2. *Prompts* adaptados a aprendizes:

Exemplo de <i>Prompt</i> 3	
<b>Versão do artigo</b> Explain this code to someone learning to program.	<b>Versão traduzida</b> Explique este código a alguém que está aprendendo a programar.

#### 3. *Prompts* granulares:

Exemplo de <i>Prompt</i> 4	
<b>Versão do artigo</b> Explain this code at block level.	<b>Versão traduzida</b> Explique este código em nível de bloco.

#### 4. *Prompts* com contexto de persona ou papel:

Exemplo de <i>Prompt</i> 5	
<b>Versão do artigo</b> You are a tutor...	<b>Versão traduzida</b> Você é um tutor...

5. *Prompts* baseados em literatura anterior, como os utilizados em estudos de MacNeil et al. (2022):

Exemplo de <i>Prompt</i> 6	
<b>Versão do artigo</b>	<b>Versão traduzida</b>
Summarize and explain the goal of the above code	Resuma e explique o objetivo do código acima.

A avaliação dos resultados foi conduzida por meio de uma combinação de análise quantitativa e qualitativa. Na dimensão quantitativa, foram empregadas métricas linguísticas — número de palavras, número de sentenças, índice de legibilidade de Kincaid et al. (1975), densidade lexical (do inglês, *Lexical Density*) e diversidade vocabular — para caracterizar o estilo e a estrutura das explicações. Na análise qualitativa, avaliadores humanos classificaram as explicações quanto à assertividade, completude, concisão e especificidade, permitindo identificar padrões de qualidade e limitações recorrentes nas respostas geradas.

Os resultados revelaram que os LLMs produziram explicações corretas em 93% dos casos, completas em 82%, específicas em 77%, mas apenas concisas em 58%, indicando uma tendência a gerar explicações verbosas ou repetitivas. O desempenho variou entre os modelos e linguagens, com destaque para o GPT-4, que apresentou explicações mais detalhadas e precisas, e para o LLaMA 2, que teve resultados mais consistentes em Python e Java, especialmente quando utilizado com *Role Prompt*, permitindo ao modelo adotar um tom mais didático.

Além disso, o artigo destacou o papel fundamental do hiperparâmetro temperatura na geração de explicações: valores baixos (0.0–0.5) produziram respostas mais diretas e concisas, enquanto temperaturas mais altas ( $\geq 1.5$ ) levaram a respostas criativas, porém incoerentes ou incorretas. Para ilustrar esse comportamento, apresenta-se no Apêndice A um exemplo de código em Java e a respectiva explicação gerada pelo modelo.

Em resumo, o trabalho de Oli et al. (2023) evidencia o potencial dos LLMs como ferramentas de apoio à explicação de código em contextos educacionais, adotando a avaliação humana como estratégia para mensurar a qualidade das explicações geradas. Por sua vez, a presente dissertação investiga a explicação de código sob uma abordagem alternativa de avaliação automatizada, empregando métricas semânticas e o paradigma LaaJ. Essa opção metodológica viabiliza a análise em maior escala, adequada a cenários experimentais mais extensos, sem recorrer à avaliação humana.

### 3.5 Avaliação Comparativa e Explicação da Geração de Código Baseada em Grandes Modelos de Linguagem: Uma Abordagem Centrada na Causalidade

No trabalho de Ji et al. (2023), os autores propõem um *framework* orientado por análise causal, investigando como diferentes estilos de PE afetam a qualidade e a explicação de código gerado por LLMs. A motivação central do estudo reside no fato de que, apesar da crescente adoção dos LLMs em tarefas de geração de código — como no suporte a desenvolvedores e na educação em programação —, o comportamento desses modelos ainda é altamente sensível a pequenas variações nos *prompts*. Reformulações semânticas sutis, embora mantendo o mesmo objetivo, podem resultar em saídas drasticamente diferentes, muitas vezes com impacto negativo na qualidade do código.

Com o objetivo de mitigar essa limitação, o estudo propõe não apenas avaliar os resultados dos modelos, mas também explicar por que e como as reformulações de *prompts* influenciam as respostas geradas, com base em evidências estatísticas e causais. Trata-se, portanto, de um esforço para tornar a engenharia de *prompts* explicável, mensurável e otimizável, contribuindo para o avanço de práticas mais robustas no uso de LLMs.

Os experimentos foram conduzidos utilizando o *benchmark* APPS (HENDRYCKS et al., 2021), composto por problemas de programação em Python com diferentes níveis de dificuldade, e envolveram a avaliação de modelos de distintas escalas, incluindo GPT-Neo, GPT-3.5-Turbo e GPT-4. Foram analisadas variações sistemáticas de *prompts*, agrupadas em categorias como *instruction prompts*, *Role Prompting* e *prompts* de contexto ou cenário, permitindo observar interações complexas entre estilo de *prompt* e desempenho técnico.

Os autores aplicaram métodos formais de inferência causal com o objetivo de compreender como variações na formulação de *prompts* influenciam as saídas de código geradas por LLMs. Para isso, empregaram técnicas de descoberta causal baseadas em grafos acíclicos direcionados (do inglês, *directed acyclic graph* (DAG)) (YU et al., 2019), utilizando o algoritmo DiBS (*Differentiable Bayesian Structure Learning*) (LORCH et al., 2021), bem como o método *Double Machine Learning* (DML) (CHERNOZHUKOV et al., 2016) para estimar o *Average Treatment Effect* (ATE) (NEAL, 2020), isto é, o impacto médio de alterações específicas no *prompt* sobre métricas técnicas do código gerado.

As técnicas de engenharia de *prompts* avaliadas foram organizadas em três categorias principais: (i) instruções explícitas de reescrita (*Instruction Prompts*), voltadas à modificação do formato ou estilo da saída; (ii) *Role Prompting*, no qual o modelo assume uma *persona* específica; e (iii) *prompts* de contexto ou cenário (*Scenario Prompts*), que situam a tarefa em um domínio ou situação particular. A combinação dessas categorias resultou em mais de 100 variações distintas de *prompts*, geradas de forma automática por um meta-modelo, permitindo a análise de interações complexas entre estilo textual e desempenho técnico.

Para ilustrar a elaboração e aplicação das técnicas de *prompting* analisadas no estudo de Ji et al. (2023), a Tabela 1 apresenta alguns exemplos representativos dos *prompts* utilizados:

Tabela 1 – Tabela adaptada de (JI et al., 2023)

<p><b>Usuário: Role + Cenário.</b> Reformule o problema de programação em tags XML, mantendo seu significado e estrutura originais, para ajudar a melhorar a compreensão da intenção do problema e facilitar uma melhor geração de código. <b>Instrução.</b> Você deve manter todos os símbolos matemáticos no formato latex. Aqui está o problema original:</p> <p>&lt;text&gt; Pergunta &lt;/text&gt;</p>		
	Nome	Frase
Role	Tutor	Você é um professor que ensina programação em Python.
	Pesquisador	Você é pesquisador na área de grandes modelos de linguagem.
	Estudante	Você é um estudante do curso de engenharia de software.
Cenário	Parceiro	Você precisa reformular uma questão de programação para explicá-la ao seu parceiro.
	Específico	Você precisa reformular as perguntas sobre programação para torná-las mais adequadas ao python3.
Instrução	Fácil	Simplifique o problema de uma maneira mais direta e fácil de entender.
	Objetivo	Reescreva o problema para torná-lo mais objetivo e claro.
	Criativo	Reescreva o problema de uma maneira mais criativa e envolvente, garantindo a clareza da questão.

Para a avaliação das respostas, utilizando *ground truth*, os autores empregaram um conjunto abrangente de métricas, contemplando corretude funcional (*Pass Rate*, erros de execução), similaridade semântica (BLEU e CodeBLEU), diversidade entre soluções, sobrecarga computacional, aspectos de legibilidade e estilo (conformidade com PEP8<sup>13</sup>) e critérios de segurança<sup>14</sup>. Os resultados indicaram que *prompts* otimizados com base em inferência causal apresentaram desempenho superior aos *prompts* manuais, com destaque para o modelo GPT-4, que obteve ganhos estatisticamente significativos em múltiplas métricas. Em contraste, modelos de menor escala, como o GPT-Neo, mostraram limitações de adaptação e maior sensibilidade ao formato do *prompt*.

Embora o foco do estudo esteja na explicação causal dos efeitos do *prompting*, e não na exploração empírica estruturada de combinações específicas, seus achados reforçam a premissa de que o design do *prompt* exerce influência mensurável e não linear sobre a qualidade do código gerado, oferecendo subsídios conceituais relevantes para investigações experimentais baseadas em combinações de técnicas de PE.

O estudo demonstra que o comportamento dos LLMs em tarefas de geração de código é altamente sensível à formulação do *prompt*, mas também explicável e passível de otimização por

<sup>13</sup> <https://github.com/psf/black>

<sup>14</sup> <https://github.com/semgrep/semgrep>

meio de técnicas estruturadas. Embora o trabalho foque na análise causal de estilos individuais de *prompt*, ele reforça a relevância de explorar combinações estruturadas de técnicas de PE, aspecto que é diretamente investigado na presente dissertação.

### 3.6 Explicação de exemplos de código em cursos introdutórios de programação: LLMs versus humanos

No estudo conduzido por Lekshmi-Narayanan et al. (2024), os autores investigam a viabilidade do uso de LLMs, em particular o ChatGPT (GPT-3.5-turbo), para gerar explicações linha a linha de exemplos de código em cursos introdutórios de programação. A motivação principal do trabalho é mitigar o conhecido “gargalo de autoria” (do inglês, *authoring bottleneck*) enfrentado por instrutores, que frequentemente não dispõem de tempo suficiente para redigir explicações detalhadas e personalizadas para cada exemplo utilizado em sala de aula — um processo que, segundo os autores, pode demandar até 30 minutos por exemplo, mesmo com auxílio de ferramentas de autoria.

O estudo foi conduzido a partir de exemplos de código em Java extraídos do sistema PCEX (*Programming Code Example Explorer*) (HOSSEINI et al., 2020), abrangendo diferentes níveis de complexidade. As explicações analisadas foram obtidas de três fontes distintas: especialistas humanos, estudantes e o próprio LLM, permitindo uma comparação direta entre explicações humanas e automatizadas.

Foram exploradas diferentes variações de *prompts*, que variaram quanto ao nível de instrução, estruturação e refinamento progressivo da saída, incluindo *prompts* simples, avançados e estendidos. Além disso, os autores analisaram o impacto da configuração do hiperparâmetro de temperatura (do inglês, *temperature*), variando de 0 a 1, na diversidade e no estilo das explicações geradas.

A avaliação dos resultados combinou métricas quantitativas de legibilidade, análise lexical e medidas de similaridade semântica, incluindo BERTScore, com o objetivo de mensurar o alinhamento conceitual entre as explicações geradas pelo LLM e aquelas produzidas por especialistas humanos.

Os resultados revelaram que o *Simple Prompt* — instrução direta — gerou explicações com maior proximidade semântica em relação às produzidas por especialistas, atingindo, por exemplo, um BERTScore de 0,89. Já o *Extended Prompt* — que refina a saída anterior, pedindo melhorias com foco em cobertura, consistência e clareza —, embora tenha gerado textos mais detalhados, apresentou menor legibilidade, indicando que a complexidade adicional pode prejudicar a compreensão, especialmente entre estudantes iniciantes. As explicações dos estudantes, por sua vez, foram mais curtas, com vocabulário mais restrito, mas com melhor escore de legibilidade.

Em síntese, o trabalho demonstra que LLMs podem atuar como ferramentas de apoio à explicação de código em contextos educacionais, desde que o nível de detalhamento e a clareza

das explicações sejam cuidadosamente balanceados. Diferentemente da presente dissertação, a avaliação adotada pelos autores baseia-se majoritariamente em comparação direta com explicações humanas, ao passo que este trabalho investiga abordagens de avaliação automatizada, empregando métricas semânticas e o paradigma *LLM-as-a-Judge*.

### 3.7 O Impacto da Programação de Prompts na Geração de Código em Nível de Função

Avançando em relação aos estudos que analisam técnicas de PE de forma isolada ou em pequenos conjuntos, Khojah et al. (2025) investigam o impacto de técnicas de *prompt programming* na síntese de funções em Python, propondo o conjunto de dados CodePromptEval (221 tarefas  $\times$  32 combinações de técnicas de PE = 7.072 *prompts*).

Nesse estudo são avaliados os efeitos individuais e de interação entre cinco técnicas de *prompt*: *Few-Shot*, *Chain-of-Thought* (CoT), *Persona* ou *Role Prompting*, *Function Signature* — que explicita a assinatura da função (nome, parâmetros e, opcionalmente, tipo de retorno), fornecendo informações de interface — e *Packages* — definida como uma lista de bibliotecas e módulos disponíveis no ambiente, extraída dos *imports* do arquivo e utilizada para oferecer contexto sobre dependências potenciais. As duas últimas técnicas são tratadas pelos autores como “peças de contexto” facilmente acessíveis ao desenvolvedor e exploradas por sua relevância em tarefas de geração de código em nível de função. O desenho experimental é *full-factorial*, contemplando todos os subconjuntos possíveis das cinco técnicas, o que permite mensurar efeitos principais e interações entre os fatores analisados.

No que diz respeito a modelos e protocolos de execução, os autores executam as 7.072 variações de *prompts* em três LLMs de geração de código contemporâneos (GPT-4o, Llama3-70B-Instruct e Mistral-22B), reamostrando cada *prompt* três vezes para controlar a não-deterministicidade, com temperatura 0.2. Também coletaram dados de modelos de gerações anteriores (GPT-3.5, Llama2) para análises complementares. A infraestrutura incluiu execução local em cluster ( $\approx 2.800GPU - hours$ ) e uso da API OpenAI para GPTs.

As avaliações cobrem três dimensões complementares: *correctness* (execução contra testes da benchmark CoderEval, medindo Pass@k, com distinção entre correção sintática e semântica), *similarity* (CrystalBLEU, que combina n-gramas e tenta refletir semelhança sintático-semântica ao *baseline* humano) e *quality* (manutenibilidade por Pylint — *code smells* — e complexidade: McCabe cyclomatic e cognitive complexity). A medição de erros é instrumentada (timeout, captura de exceções) para classificar falhas (AssertionError, TypeError, ImportError, etc.).

Entre os resultados mais relevantes observados no estudo, destacam-se quatro achados principais. Primeiro, a presença das técnicas *Signature* e *Few-Shot* mostrou-se o fator de maior impacto na correção funcional do código, sendo que a combinação *Signature + Few-Shot*

apresentou melhor corretude em Pass@1, alcançando 57,5% para o GPT-4o nas condições avaliadas. Em segundo lugar, o GPT-4o obteve os melhores índices agregados de *pass rate*, entendido como a taxa global de funções plausivelmente corretas produzidas ao longo dos experimentos, com aproximadamente 52,2%, seguido pelos modelos Llama3 e Mistral.

Além disso, a análise de similaridade entre o código gerado e o *baseline* humano, medida por meio do CrystalBLEU, indicou valores geralmente baixos — com variações entre aproximadamente 7,5% e 16,8% no caso do Llama3 —, sendo observado que o uso das técnicas *Signature* e *Persona* tende a aumentar esse alinhamento. Por fim, os autores identificam um *trade-off* relevante entre correção e manutenibilidade: enquanto prompts que combinam *Signature* e *Few-Shot* elevam as taxas de acerto, também aumentam a complexidade do código e o número de code smells, indicadores iniciais de possíveis problemas de qualidade de software. Em contrapartida, técnicas como CoT, *Persona* e *Packages* contribuem para a redução da complexidade e dos *code smells*, embora, em alguns casos, impliquem queda na taxa de acerto.

As lições aprendidas (sintetizadas pelos autores) incluem pontos que são de interesse metodológico para investigações em engenharia de *prompts*: (L1) o efeito global das técnicas é menor do que se poderia supor — a diferença entre melhores e piores combinações é, na média, da ordem de  $\sim 10$  pontos percentuais no Pass@1; (L2) fornecer informação de interface (*Signature* ou exemplos) é robustamente útil para correção, mas limita “criatividade” do modelo; (L3) há *trade-offs* entre correção e qualidade (manutenibilidade) e interações não-lineares entre técnicas (combinar técnicas nem sempre aumenta desempenho e pode gerar regressões); além disso, os autores ressaltam a sensibilidade de resultados ao formato/ordem do *prompt* e defendem refinamento manual de *templates* como prática experimental recomendada.

Em resumo, o trabalho demonstra que os efeitos das técnicas de PE são moderados, interdependentes e sensíveis à forma como são combinadas, reforçando a importância de análises experimentais controladas para compreender esses fenômenos. Embora o estudo se concentre na geração de código em nível de função e na avaliação de aspectos como correção e manutenibilidade, seus resultados fornecem evidências empíricas relevantes sobre combinações entre técnicas de *prompting*, oferecendo subsídios metodológicos que dialogam diretamente com investigações que ampliam esse escopo para considerar, de forma integrada, a explicação de código e abordagens automatizadas de avaliação, como as exploradas nesta dissertação.

### 3.8 Comparativo e Diferenciais deste Trabalho

Esta dissertação fundamenta-se em uma abordagem experimental que investiga o uso de LLMs na geração e explicação de código, com ênfase nas combinações de técnicas de PE. Os trabalhos relacionados seguem uma progressão que parte de aplicações voltadas à geração e/ou explicação automatizada de código, assim como análises técnicas sobre o comportamento dos modelos diante de diferentes técnicas de *prompting*.

Ao longo dos trabalhos relacionados analisados, observa-se a adoção, de forma direta ou indireta, das principais estratégias mais conhecidas de PE: *Role Prompt*, *Zero-shot*, *Few-shot* e *CoT*. Além disso, alguns estudos se destacam pela proposição de *frameworks* específicos (WANG; ZHOU; CHEN, 2024; JI et al., 2023), que estruturam investigações aplicadas à geração automatizada de código, à explicação de soluções algorítmicas e à avaliação de saídas produzidas por LLMs. Esses *frameworks* reforçam a importância de sistematizar abordagens de *prompting*, contribuindo para o controle de variáveis experimentais e para a mensuração rigorosa dos efeitos das diferentes estratégias utilizadas.

Entre os trabalhos descritos, destaca-se o estudo de Damke, Gregorini e Copetti (2024), que apresenta uma análise comparativa interessante entre algumas técnicas de PE, ao empregar métricas objetivas para avaliar a qualidade da geração de código. Contudo, sua limitação metodológica está na análise isolada das técnicas, sem explorar possíveis efeitos advindos de sua combinação. Essa lacuna é diretamente contemplada pela proposta deste trabalho, que busca examinar os impactos da combinação estruturada de múltiplas técnicas de *prompting* de modo a investigar potenciais sinergias no comportamento dos LLMs em tarefas de geração e explicação de código.

Nesse mesmo eixo de análise estrutural das técnicas, o estudo de Khojah et al. (2025) representa um avanço significativo ao realizar uma investigação sistemática do *prompt programming* por meio de mais de 7.000 combinações fatoriais envolvendo cinco técnicas: *Few-Shot*, *Chain-of-Thought*, *Persona*, além de duas estratégias menos difundidas — *Function Signature* e *Packages*. Essas duas últimas introduzem especificações formais da função e delimitações explícitas de bibliotecas permitidas, capturando dimensões que normalmente não aparecem em estudos tradicionais de *prompting*. Os autores também avaliam vários LLMs (GPT-4o, Llama3-70B e Mistral-22B) sob métricas como Pass@k e CrystalBLEU, demonstrando que certas combinações produzem ganhos expressivos, enquanto outras resultam em degradação sistemática de qualidade.

Já o estudo de Ji et al. (2023) contribui para a compreensão causal da influência dos *prompts*, aplicando métodos formais para descobrir relações estatisticamente significativas entre variações de *prompting* e desempenho do código. Embora esse estudo ofereça uma contribuição inovadora ao propor a criação automatizada de *templates* de *prompt*, sua ênfase está mais concentrada na inferência causal do que na análise empírica de combinações específicas, deixando espaço para explorações experimentais complementares, como as realizadas nesta dissertação.

No que se refere à explicação de código, os trabalhos de Lekshmi-Narayanan et al. (2024) e Ji et al. (2023) enfatizam a avaliação humana como componente central da análise qualitativa. Embora essa abordagem seja valiosa do ponto de vista interpretativo, observa-se um movimento crescente em direção a métodos de avaliação automatizada capazes de mensurar a qualidade explicativa de forma escalável em alguns cenários do mundo real, a exemplo de explicações de códigos. Métricas semânticas e abordagens baseadas no paradigma *LLM-as-a-Judge* emergem, assim, como alternativas promissoras, especialmente em cenários experimentais de maior porte.

Inserida nesse contexto, esta dissertação avança o estado da arte ao adotar uma abordagem estruturada, combinatória e multivariada para a Engenharia de *Prompts* aplicada à geração e explicação de código. São investigadas seis técnicas de PE — *Role Prompting*, RaR, *Zero-Shot-CoT*, *Skeleton-of-Thought*, *Self-Verification* e *Retrieval-Augmented Generation* (RAG) — organizadas em 13 cenários experimentais cuidadosamente delineados. Diferentemente da maior parte dos estudos revisados, que avaliam técnicas de forma isolada ou em poucas combinações, esta pesquisa explora interações estruturadas entre múltiplas estratégias, permitindo identificar sinergias, conflitos e padrões de comportamento dos modelos diante de variações controladas de *prompting*.

Outro diferencial metodológico reside na estratégia de avaliação adotada. Além da avaliação baseada em um *ground truth*, por meio da métrica BERTScore, a pesquisa incorpora julgamentos automatizados por meio de avaliadores *LLM-as-a-Judge* (Claude e Gemini), instruídos para avaliar tanto a correção do código quanto a qualidade da explicação gerada. Essa abordagem possibilita uma análise abrangente que contempla aspectos funcionais, semânticos e explicativos, ampliando o escopo de avaliação em relação aos trabalhos existentes.

Complementarmente, a investigação distingue-se pelo uso de um conjunto diversificado de modelos — GPT-4o-mini, Llama-3.1-405B, DeepSeek-V3, Gemma3-27B, Mixtral-8×22B-Instruct e Qwen2.5-72B —, o que permite analisar diferenças arquiteturais e avaliar a robustez das combinações de técnicas em cenários heterogêneos. A análise conjunta dessas dimensões fornece princípios gerais para o design de *prompts* combinados e diretrizes práticas para a seleção de técnicas conforme requisitos como qualidade, interpretabilidade e desempenho.

A Tabela 2 consolida uma comparação estruturada entre os principais trabalhos associados e esta pesquisa, evidenciando os modelos de linguagem utilizados, as estratégias de *prompting* aplicadas, as combinações exploradas, bem como as linguagens de programação e os meios e/ou métricas de avaliação adotados.

Tabela 2 – Tabela comparativa do trabalhos relacionados

Trabalho	LLMs	Técnicas de prompt	Combinação	Linguagem Programação	Avaliação
(WANG; ZHOU; CHEN, 2024)	GPT-4 GPT-4o Llama3-8b Mixtral8x-7b	Chain-of-Thoughts Role Prompting	1 Combinação: CoT e Role	Python	Human-in-the-loop
(DAMKE; GREGORINI; COPETTI, 2024)	Llama 3 8B Gemma 7B	Zero-Shot Few-Shot Chain-of-Thought	Não há	Python	Pass@K
(ZHENG et al., 2025)	Llama 3.0 e 3.1, 8B, 70B, 405B GPT-4o	Chain-of-Thought	Não há	Python	Pass@K Pass-n@K
(OLI et al., 2023)	GPT-3.5-Turbo-0613 ChatGPT-4 Llama2	Role Prompting Few-Shot	1 Combinação	Java Python C++	Lexical Density assertividade completude consistência especificidade
(JI et al., 2023)	GPT-Neo (2.7B) GPT-3.5-Turbo GPT-4	Role prompt	Não há	Python	BLEU CodeBLEU
(LEKSHMI-NARAYANAN et al., 2024)	GPT-3.5-Turbo	Role prompt	Não há	Java	Lexical Legibilidade Similaridade
(KHOJAH et al., 2025)	GPT-4o Llama3-70B-Instruct Mistral-Small-Instruct-2409 (22B).	Few-Shot Chain-of-Thought Persona Signature Packages	32 Combinações	Python	Benchmark CoderEval Pass@K CrystalBLEU Pylint Medições de erros
Este trabalho	GPT-4o-mini Llama-3.1-405b DeepSeek-v3 Gemma3-27b Mixtral-8x22b-instruct Qwen2.5-72B	Role Prompting RaR Zero-Shot-CoT Skeleton-of-Thought Self-Verification RAG	12 Combinações	Python	BERTScore  <i>LLM-as-a-Judge</i> com 22 critérios Claude-Sonnet-4-5-20250929 Gemini-2.5-flash-lite

## 4 METODOLOGIA

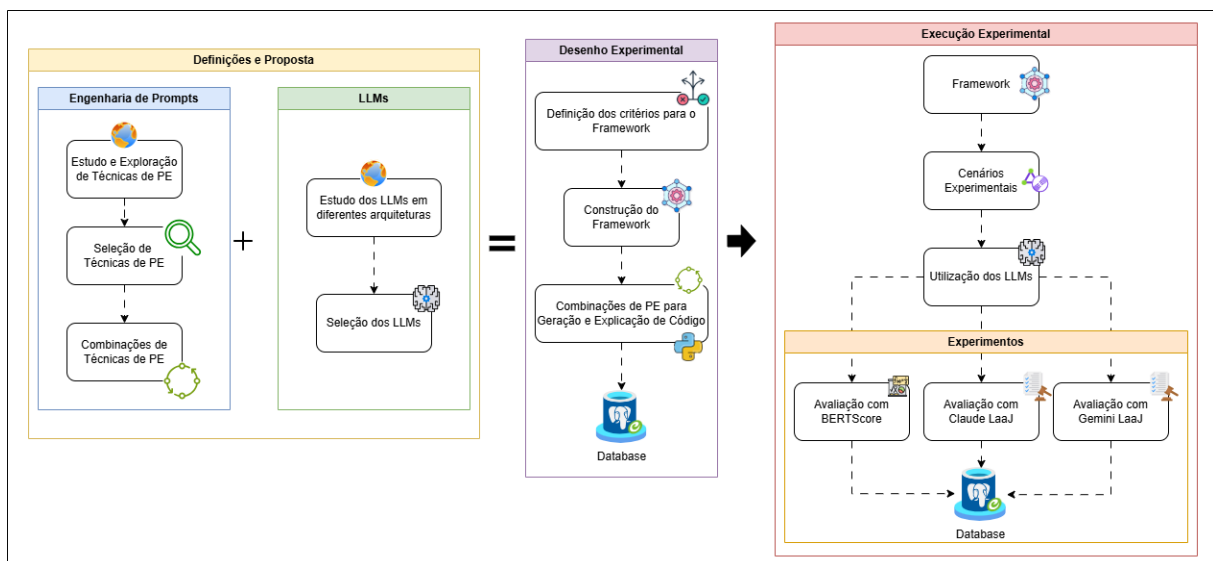
Este capítulo apresenta a abordagem metodológica adotada na investigação do impacto das combinações de técnicas de PE na geração e explicação de código por LLMs. Fundamentado nos referenciais teóricos discutidos no Capítulo 2, o capítulo está estruturado de modo a articular três dimensões complementares: a definição conceitual dos elementos que compõem o estudo, o delineamento do desenho experimental e a execução controlada dos experimentos.

### 4.1 Visão Geral da Abordagem Metodológica

Esta pesquisa adota uma abordagem metodológica mista, de caráter propositivo e experimental, na qual definições conceituais são inicialmente estabelecidas e, em seguida, operacionalizadas e avaliadas por meio de um desenho experimental controlado. O objetivo central é investigar, de forma sistemática, o impacto de combinações de técnicas de PE na geração e explicação de código nos LLMs.

A Figura 10 sintetiza o fluxo metodológico adotado, destacando os principais componentes envolvidos em cada macroetapa e evidenciando as relações entre as combinações de técnicas de PE, a seleção dos LLMs, o *framework* experimental e os procedimentos de avaliação, apresentando uma visão integrada do processo e enfatizando a articulação entre definição conceitual, operacionalização experimental e execução controlada ao longo da investigação.

Figura 10 – Visão integrada do fluxo metodológico da pesquisa



Elaborada pelo Autor

Na macroetapa de Definições e Proposta são apresentadas a exploração e seleção das técnicas de PE adotadas, a definição das combinações consideradas, as estratégias de avaliação

da geração e explicação de código em Python, a seleção dos LLMs empregados e a concepção do *framework* experimental. Trata-se de uma etapa estruturante, pois delimita o escopo da investigação e define os mecanismos que viabilizam sua implementação.

Na sequência, a macroetapa de Desenho Experimental trata da operacionalização das definições previamente estabelecidas, preservando a separação entre concepção metodológica e implementação empírica. Inicialmente, são retomadas as questões de pesquisa e explicitados os objetivos associados a cada experimento. Em seguida, apresenta-se o *framework* experimental enquanto infraestrutura responsável por integrar os componentes metodológicos definidos e padronizar as interações com as combinações de técnicas de PE e os LLMs. Posteriormente, descrevem-se a infraestrutura de dados utilizados e os cenários experimentais — entendidos como as instanciações práticas das combinações de técnicas de PE. Por fim, detalha-se a aplicação concreta das estratégias de avaliação previamente definidas, explicitando como a métrica BERTScore foi empregada na análise da geração de código e como a abordagem *LLM-as-a-Judge*, orientada pelos critérios estabelecidos, foi utilizada para avaliar conjuntamente a geração e a explicação de código.

A macroetapa de Execução Experimental corresponde à aplicação sistemática dos cenários experimentais aos modelos de linguagem, conforme o desenho metodológico previamente estabelecido. Nessa fase, o *framework* experimental orquestra as requisições aos LLMs, controla a execução dos cenários experimentais e os parâmetros de utilização, e registra de forma padronizada as saídas produzidas — compostas por código e respectivas explicações. Em seguida, as estratégias de avaliação são aplicadas sobre essas saídas, gerando os dados quantitativos e estruturados que subsidiam a análise empírica. A interpretação e discussão desses resultados são apresentadas no Capítulo 5.

## 4.2 Definições e Proposta

Nesta etapa, são descritos os elementos que estruturam a condução da investigação experimental, incluindo as decisões metodológicas relacionadas à seleção de técnicas, definição de combinações, critérios de avaliação, escolha dos modelos e organização do *framework* proposto.

### 4.2.1 Exploração e Seleção de Técnicas de PE

A exploração das técnicas de PE que fundamentam esta pesquisa foi apresentada na Seção 2.3. A partir dessa análise, realizou-se a seleção das técnicas de PE a serem utilizadas neste trabalho, considerando critérios como dificuldade de implementação prática, frequência de uso na literatura, adequação às tarefas de geração e explicação de código e potencial de articulação entre as técnicas. Com base nesses critérios, foram escolhidas as técnicas consideradas mais relevantes e com maior potencial de influência no comportamento dos modelos, as quais serviram

de base para a construção de combinações de técnicas de PE definidas na subseção seguinte.

A técnica **P1 – Role Prompting** foi empregada neste trabalho por meio da definição explícita de um papel fixo atribuído ao modelo de linguagem, especificado diretamente no *prompt*. Em particular, o LLM é instruído a assumir o papel de **tutor de programação**, o que orienta a forma de apresentação das respostas, o nível de detalhamento do código gerado e a estrutura da explicação associada. Essa configuração foi mantida de forma padronizada em todos os *prompts* que utilizaram essa técnica, por meio da definição explícita do mesmo papel atribuído ao modelo.

Na **P2 – RaR**, inspirada em Deng et al. (2023), a abordagem adotada consiste em induzir o modelo a reformular a pergunta original antes de gerar a resposta, com o objetivo de ampliar a compreensão semântica da tarefa e reduzir ambiguidades na interpretação do enunciado.

Para a técnica **P3 – ZS-CoT**, optou-se por empregar uma instrução consagrada na literatura, conforme proposto por Kojima et al. (2022), com o objetivo de induzir o modelo a realizar um raciocínio encadeado e estruturado durante a resolução da tarefa.

Na técnica **P4 – SoT** a abordagem adotada baseia-se na decomposição da tarefa em três linhas de raciocínio, permitindo que o modelo processe diferentes aspectos da pergunta de forma paralela e consolide esses raciocínios em uma resposta final estruturada.

Na técnica **P5 – SV**, o objetivo é induzir o modelo a reavaliar iterativamente suas próprias respostas, promovendo um mecanismo de verificação interna que busca reduzir inconsistências e aprimorar a qualidade final da saída. Essa abordagem fundamenta-se na reformulação sucessiva da pergunta e da resposta, conforme proposto por Weng et al. (2022), permitindo que o modelo refine seu raciocínio ao longo das iterações.

Por fim, a técnica **P6 – RAG** tem como finalidade enriquecer o *prompt* com contexto externo relevante, recuperado a partir de uma base de dados ou de um conjunto de dados, construída especificamente para o domínio da tarefa. A adoção de uma base vetorial local visa garantir maior controle sobre as informações utilizadas e favorecer a reprodutibilidade dos experimentos.

Os detalhes operacionais, exemplos de *prompts* e fluxos de execução associados a cada técnica são apresentados no Desenho Experimental, na Subseção 4.3.4.

#### 4.2.2 Combinações de Técnicas de PE

As técnicas de PE selecionadas nesta pesquisa foram organizadas em diferentes combinações, apresentadas na Tabela 3, com o objetivo de compor cenários metodologicamente distintos para investigação experimental. Cada combinação define um cenário experimental específico, resultando em configurações que variam tanto em complexidade quanto no grau de direcionamento imposto ao modelo de linguagem. Essa organização permite estruturar o espaço experimental de forma sistemática, estabelecendo uma progressão controlada entre estratégias de *prompting* mais simples e abordagens mais elaboradas.

Tabela 3 – Tabela das Técnicas de PE e Cenários de Combinações

Técnicas de PE	Cenários de Combinações
P <sub>1</sub> - Role Prompting	S <sub>0</sub> : P <sub>1</sub>
P <sub>2</sub> - Rephrase and Respond (RaR)	S <sub>1</sub> : P <sub>1</sub> + P <sub>6</sub>
P <sub>3</sub> - Zero-Shot-CoT (ZS-CoT)	S <sub>2</sub> : P <sub>1</sub> + P <sub>2</sub> + P <sub>6</sub>
P <sub>4</sub> - Skeleton-of-Thoughts (SoT)	S <sub>3</sub> : P <sub>1</sub> + P <sub>3</sub>
P <sub>5</sub> - Self-Verification (SV)	S <sub>4</sub> : P <sub>1</sub> + P <sub>3</sub> + P <sub>6</sub>
P <sub>6</sub> - Retrieval Augmented Generation (RAG)	S <sub>5</sub> : P <sub>1</sub> + P <sub>4</sub> + P <sub>6</sub>
	S <sub>6</sub> : P <sub>1</sub> + P <sub>5</sub> + P <sub>6</sub>
	S <sub>7</sub> : P <sub>1</sub> + P <sub>2</sub> + P <sub>4</sub> + P <sub>6</sub>
	S <sub>8</sub> : P <sub>1</sub> + P <sub>2</sub> + P <sub>5</sub> + P <sub>6</sub>
	S <sub>9</sub> : P <sub>1</sub> + P <sub>3</sub> + P <sub>4</sub> + P <sub>6</sub>
	S <sub>10</sub> : P <sub>1</sub> + P <sub>3</sub> + P <sub>5</sub> + P <sub>6</sub>
	S <sub>11</sub> : P <sub>1</sub> + P <sub>2</sub> + P <sub>4</sub> + P <sub>5</sub> + P <sub>6</sub>
	S <sub>12</sub> : P <sub>1</sub> + P <sub>3</sub> + P <sub>4</sub> + P <sub>5</sub> + P <sub>6</sub>

Observa-se que a técnica **P1 — Role Prompting** é comum a todas as combinações definidas, atuando como elemento estruturante dos prompts. A atribuição explícita de um papel ao modelo fornece um contexto estável para a interação, sobre o qual as demais técnicas são progressivamente articuladas. As demais estratégias de PE são então combinadas de maneira incremental, compondo diferentes arranjos que refletem hipóteses distintas sobre o impacto da interação entre técnicas no comportamento dos modelos.

A lógica que orienta essas combinações baseia-se na hipótese de que a associação entre técnicas amplamente consolidadas na literatura — como **P1 (Role Prompting)**, **P3 (Zero-Shot Chain-of-Thought)** e **P6 (Retrieval Augmented Generation)** — e técnicas relativamente menos exploradas em conjunto, como **P2 (Rephrase and Respond)**, **P4 (Skeleton-of-Thoughts)** e **P5 (Self-Verification)**, pode ampliar as possibilidades de construção de *prompts* reutilizáveis e metodologicamente robustos. Parte-se do pressuposto de que tais associações podem explorar sinergias complementares entre estratégias de reformulação, decomposição, verificação e contextualização, indo além da aplicação isolada de cada técnica.

Cabe destacar que as técnicas P2 e P3 não são empregadas conjuntamente em nenhum dos cenários propostos. Essa decisão fundamenta-se no entendimento de que ambas atuam sobre a reestruturação cognitiva da entrada do usuário, ainda que por mecanismos distintos. Enquanto a P2 enfatiza a reformulação explícita da pergunta como estratégia para aprimorar a compreensão semântica do problema, a P3 induz o modelo a realizar um raciocínio estruturado passo a passo durante a resolução da tarefa. Dessa forma, optou-se por tratá-las como estratégias alternativas, com finalidades semelhantes, porém aplicadas de maneira independente, evitando sobreposição funcional nas combinações avaliadas.

Busca-se, adicionalmente, analisar como o aumento gradual da complexidade das combinações — por meio da incorporação progressiva de técnicas — impacta aspectos como a

precisão, a consistência e a confiabilidade das respostas produzidas pelos LLMs. Ao organizar as técnicas em diferentes arranjos conceituais, esta pesquisa estabelece a base para uma análise comparativa dos cenários definidos, permitindo investigar de que forma distintas articulações entre estratégias de *prompting* influenciaram o desempenho dos LLMs nas tarefas de geração e explicação de código em Python.

Essas combinações de técnicas de PE constituem, portanto, o arcabouço conceitual a partir do qual os cenários experimentais são definidos, estabelecendo uma organização sistemática das estratégias de *prompting* a serem investigadas.

De forma geral, o cenário  $S_0$  é definido como *baseline* da investigação, sendo utilizado como referência para a comparação com os demais cenários experimentais. A adoção de uma *baseline* própria mostrou-se necessária, uma vez que a literatura não apresenta um padrão consolidado para avaliação de combinações de técnicas de PE em tarefas específicas como geração e explicação de código. Assim, optou-se por definir um cenário de referência caracterizado pela ausência de estratégias explícitas de indução de raciocínio ou enriquecimento contextual, permitindo avaliar a capacidade inerente dos LLMs em compreender o problema e gerar respostas com mínimo direcionamento externo. Essa definição está alinhada a práticas adotadas na literatura de avaliação de LLMs e técnicas de *prompting*, nas quais a *baseline* é construída de forma contextualizada ao problema investigado, servindo como ponto de comparação para mensurar os ganhos obtidos por diferentes estratégias (WEI et al., 2022b; KOJIMA et al., 2022).

Os cenários  $S_1$  e  $S_2$  combinam a técnica *Role* com RAG e, adicionalmente, com RaR, explorando o efeito da reformulação da pergunta aliada à recuperação de contexto externo. Os cenários  $S_3$  e  $S_4$  empregam a técnica ZS-CoT, sendo que o primeiro a aplica de forma isolada, enquanto o segundo a combina com RAG, buscando ampliar a compreensão contextual dos LLMs.

No  $S_5$ , a técnica SoT foi aplicada para gerar múltiplas interpretações paralelas da pergunta, cujas respostas foram concatenadas e reutilizadas para a geração final. No cenário  $S_6$ , combinou-se RAG com SV, permitindo que o modelo incorpore contexto externo e, em seguida, reavaliasse iterativamente sua própria resposta.

Os cenários  $S_7$  e  $S_8$  integraram RaR, SoT, SV e RAG em fluxos estruturados em múltiplas etapas, explorando diferentes ordens de aplicação das técnicas. Os cenários  $S_9$  e  $S_{10}$  combinaram ZS-CoT com SoT, SV e RAG, buscando investigar o impacto da integração entre raciocínio encadeado, decomposição paralela e verificação iterativa.

Por fim, os cenários  $S_{11}$  e  $S_{12}$  representam as combinações mais completas, integrando até cinco técnicas de PE, com o objetivo de avaliar os efeitos do aumento da complexidade do *prompt* sobre a qualidade, consistência e confiabilidade das respostas geradas.

A operacionalização prática dessas combinações — incluindo a instanciação dos *prompts*, os fluxos de execução e os procedimentos adotados em cada cenário — é apresentada no Desenho

Experimental, na Subseção 4.3.4.

### 4.2.3 Avaliação da Geração e Explicação de Código

A métrica BERTScore foi adotada neste trabalho como instrumento para avaliar a similaridade semântica entre o código gerado pelos modelos de linguagem e o código de referência (*ground-truth*).

Para a aplicação do BERTScore na tarefa de geração de código, foi selecionado o modelo *microsoft/codebert-base*<sup>15</sup>, um modelo pré-treinado especificamente voltado à representação de linguagens de programação e linguagem natural. O CodeBERT apresenta ampla cobertura de linguagens, incluindo Python, Java, JavaScript, PHP, Ruby e Go, além de consolidada adoção em tarefas de análise semântica de código. Sua escolha fundamenta-se na adequação ao domínio da programação e na robustez demonstrada em estudos prévios.

Reconhece-se, entretanto, a existência de outros modelos que também poderiam ser empregados para tarefas de similaridade entre códigos, como o *microsoft/graphcodebert-base*<sup>15</sup>, que incorpora informações estruturais por meio de grafos sintáticos; o *Salesforce/codeT5-base*<sup>16</sup>, baseado em arquitetura *encoder-decoder* voltada à geração e tradução de código; e o *microsoft/unixcoder-base*<sup>15</sup>, projetado para tarefas multitarefa envolvendo linguagem natural e código. Ainda assim, optou-se pelo CodeBERT por apresentar equilíbrio entre desempenho, maturidade e compatibilidade com o escopo experimental desta pesquisa.

No âmbito da implementação adotada, decidiu-se empregar a biblioteca *torch* para o cálculo da similaridade por cosseno entre os *embeddings* gerados. Embora existam implementações do BERTScore que retornem múltiplas métricas — como precisão, revocação e *F1-score* —, adotou-se uma configuração que produz um único valor consolidado de similaridade, expresso no intervalo  $[-1, 1]$ , o qual se mostrou mais adequado ao fluxo automatizado de avaliação empregado nesta pesquisa.

A avaliação complementar adotada nesta pesquisa baseia-se no paradigma *LLM-as-a-Judge* (LaaJ), no qual um modelo de linguagem é empregado como avaliador automático das respostas produzidas por outros modelos. Diferentemente do BERTScore — que se concentra exclusivamente na similaridade semântica do código — o paradigma LaaJ permite avaliar conjuntamente a geração de código e a qualidade da explicação associada, considerando critérios qualitativos explicitamente definidos.

Para viabilizar essa avaliação, foi definido um conjunto sistematizado de critérios, apresentados na Tabela 4, estruturado em um *prompt template* específico para o modelo julgador. Embora exista um volume significativo de estudos que empreguem LLMs como avaliadores de código, o levantamento realizado nesta pesquisa não identificou trabalhos que proponham

<sup>15</sup> GitHub - CodeBERT

<sup>16</sup> Hugging Face - CodeT5

explicitamente critérios organizados e formalizados para a avaliação simultânea da geração e da explicação de código nesse domínio específico. Essa lacuna motivou a estruturação criteriosa adotada neste trabalho, conferindo caráter inovador, até o momento, à proposta no contexto investigado.

No que se refere à avaliação da geração de código, adotou-se como base o *prompt* proposto por Tong e Zhang (2024), disponibilizado em repositório do GitHub<sup>17</sup>, o qual apresenta uma categorização sistemática de inconsistências. Foram incorporados 11 critérios (itens 0 – 10) relacionados à geração de código, preservando a estrutura conceitual proposta pelos autores. Entretanto, a dimensão referente à avaliação da qualidade das explicações não pôde ser derivada diretamente da literatura. Assim, foram desenvolvidos 11 critérios adicionais (itens 11 – 21), com o objetivo de complementar a avaliação no contexto do paradigma LaaJ.

O *prompt template* — apresentado no *Prompt 1* — foi organizado de maneira modular, abrangendo cinco componentes principais: (1) definição de entrada; (2) escala de severidade (do inglês, *Scale of Value*) para mensuração das inconsistências; (3) catálogo de inconsistências (do inglês, *Inconsistency Catalog*) que descreve detalhadamente os tipos de desvios possíveis; (4) especificação do formato de saída, responsável por garantir a padronização estrutural dos resultados retornados; e (5) protocolo de avaliação (do inglês, *Evaluation Steps*) que orienta o processo de julgamento. Essa organização contribui para um processo avaliativo transparente, reproduzível e metodologicamente robusto.

### **Definição de Entrada**

O *prompt* utilizado no processo de avaliação recebe três elementos principais: o enunciado do problema (*PROBLEM*), uma solução de referência (*CODE1*) e o *snippet* a ser avaliado (*CODE2*). A solução de referência é empregada como artefato auxiliar para a compreensão do problema, não sendo utilizada para fins de comparação direta com o código avaliado. O *CODE2* corresponde à geração do código e à respectiva explicação produzidas pelos LLMs no contexto dos experimentos.

### **Escala de Severidade**

Foi definida uma escala quantitativa de cinco níveis para a classificação das inconsistências identificadas: Insignificante (0), Pequena ou Menor (-1), Moderada (-2), Grave (-3) e Fatal (-4). Essa escala permite mensurar, de forma sistemática, o impacto associado a cada inconsistência, possibilitando uma avaliação mais granular das saídas analisadas, além de julgamentos binários. Sua adoção baseia-se na categorização de inconsistências proposta por Tong e Zhang (2024). Importa destacar que os valores negativos atribuídos às categorias de severidade constituem uma convenção operacional adotada para permitir agregação quantitativa dos resultados, viabilizando o cálculo de uma métrica sintética representativa da qualidade global do par código-explicação avaliado. Ressalta-se que tais valores não configuram uma escala contínua de

<sup>17</sup> GitHub - CodeJudge

magnitude, mas categorias discretas associadas aos tipos de inconsistências definidos no catálogo elaborado.

Considerando a natureza acumulativa da métrica, valores mais próximos de zero indicam menor ocorrência de inconsistências e, conseqüentemente, maior qualidade das respostas. Por outro lado, valores mais negativos indicam maior quantidade e/ou severidade de inconsistências. Embora não se estabeleçam limiares rígidos de classificação, valores próximos de zero podem ser interpretados como indicativos de alta qualidade, enquanto valores intermediários sinalizam inconsistências moderadas, e valores mais negativos indicam respostas com falhas significativas.

### **Catálogo de Inconsistências**

O *prompt template* incorpora um catálogo composto por 22 critérios, organizados em dois eixos complementares — geração e explicação de código —, tomando como unidade analítica a noção de inconsistência, entendida como desvio em relação aos critérios esperados para a tarefa avaliada, conforme conceituado na Subseção 2.2.3.2. Os critérios relacionados à geração de código (itens 0–10) contemplam falhas como erros lógicos, erros sintáticos, omissão de funcionalidades, falhas no tratamento de casos extremos e referências indefinidas. Já os critérios voltados à explicação de código (itens 11–21), concebidos e formalizados neste trabalho, avaliam aspectos como correspondência entre a explicação e o comportamento do código, completude da descrição, clareza e legibilidade. Cada critério está associado a uma categoria de severidade definida na escala adotada nesta investigação. A Tabela 4 apresenta a descrição detalhada desses critérios, explicitando as inconsistências consideradas no processo de julgamento.

### **Formato de Saída**

No *prompt template*, o item “*Evaluation Form*” especifica a estrutura JSON de resposta, exigindo um *array* de objetos contendo os pares inconsistência-severidade identificados.

### **Protocolo de Avaliação**

O protocolo de avaliação descrito a seguir corresponde à formalização textual das instruções explicitamente definidas no *prompt template* apresentado no *Prompt 9*. Trata-se da explicitação metodológica das etapas que o modelo julgador deve executar conforme orientado no próprio *template*.

O processo de avaliação foi estruturado em cinco etapas sequenciais, executadas de forma sistemática pelo modelo julgador (Claude ou Gemini). Cada uma das etapas corresponde ao respectivo item (1-5) do “*Evaluation Steps*”.

Na primeira etapa – análise do problema e identificação das funcionalidades requeridas – o modelo é instruído a ler atentamente o enunciado do problema (*PROBLEM*) e a solução de referência (*CODE1*), utilizando esta última exclusivamente como suporte interpretativo. A identificação das funcionalidades requeridas não decorre de um mecanismo externo, mas da interpretação do próprio enunciado pelo modelo julgador, conforme orientado no *template*. A

solução de referência auxilia na compreensão do escopo do problema, sem servir como base para comparação direta com o código avaliado.

Na segunda etapa – verificação da geração de código – o modelo compara o *snippet* avaliado (*CODE2*) com as funcionalidades identificadas a partir do enunciado. A análise consiste em verificar se o código implementa adequadamente os requisitos descritos no problema, classificando eventuais inconsistências conforme o Catálogo de Inconsistências (itens 0–10), as quais são sempre avaliadas em relação ao enunciado do problema e aos critérios previamente definidos no catálogo, e não em relação direta à solução de referência.

Na terceira etapa – avaliação da explicação linha a linha – concentra-se na análise da explicação associada ao código gerado. O modelo verifica se a explicação cobre todas as linhas e funcionalidades do *snippet* e classifica eventuais inconsistências conforme os critérios 11–21 do catálogo. Dessa forma, os aspectos avaliados não são arbitrários, mas explicitamente definidos no catálogo incorporado ao *template*.

Na quarta etapa – regra de não duplicação – o modelo é instruído a não registrar a mesma inconsistência múltiplas vezes. Essa regra evita inflacionamento artificial da penalização total, garantindo que cada desvio identificado — em relação aos critérios do catálogo — seja contabilizado uma única vez. Tal restrição preserva a consistência do cálculo agregado da severidade, impedindo que um mesmo erro impacte desproporcionalmente o valor final da avaliação.

Na última etapa – estruturação padronizada da saída – o modelo deve retornar exclusivamente um objeto JSON estruturado como uma lista de inconsistências identificadas. Cada inconsistência é representada por um par atributo-valor contendo o tipo de inconsistência detectado e sua respectiva severidade (“*inconsistency*”: “<*inconsistency1*>”, “*severity*”: “<*severity1*>”), conforme o catálogo predefinido. Casos especiais são tratados de forma específica: quando nenhuma inconsistência é encontrada, o sistema retorna um objeto único indicando {“*inconsistency*”: “None”, “*severity*”: “Negligible”}. Adicionalmente, a estrutura JSON deve obrigatoriamente incluir um campo final denominado “*final\_answer*”, que agrega numericamente todas as penalizações identificadas ao longo da análise, produzindo uma métrica quantitativa única que representa a qualidade geral do par código-explicação avaliado. Esta especificação rígida do formato de saída assegura a padronização dos resultados e viabiliza o processamento automatizado subsequente dos dados coletados.

Tabela 4 – Critérios de Avaliação dos LaaJ

Avaliação de Geração de Código				Avaliação de Explicação de Código			
ID	Descrição	Severidade	Escala	ID	Descrição	Severidade	Escala
0	Utilização de métodos ou algoritmos diferentes para resolver o problema	Insignificante	0	11	Determina se a explicação descreve corretamente o que o código realmente faz	Fatal	-4
1	Falta de declarações de dependência utilizadas no programa	Insignificante	0	12	Avalia se todas as partes relevantes do trecho (variáveis, lógica, condições, importações, funções) estão cobertas	Grave	-3
2	Ausência de tratamento de exceções para eventos inesperados, tais como entradas inválidas	Insignificante	0	13	Avalia se a explicação é compreensível e legível pelo público-alvo (desenvolvedores, revisores ou alunos)	Grave	-3
3	Inclusão de instruções ineficientes ou desnecessárias	Insignificante	0	14	Verifica se a explicação vai além de descrever o que acontece para incluir por que e como isso acontece, e o raciocínio por trás das decisões de design	Grave	-3
4	Incapacidade de tratar casos extremos	Pequeno	-1	15	Avalia se os termos e conceitos de programação corretos (por exemplo, “método”, “iterador”, “mutável”) são usados de forma precisa e consistente	Moderado	-2
5	Erros lógicos que levam a resultados incorretos	Grave	-3	16	Avalia se a explicação está bem organizada (ordem lógica, recuo, marcadores, separação de ideias)	Moderado	-2
6	Erros de sintaxe ou erros de compilação	Fatal	-4	17	Determina se a explicação relaciona o trecho com seu objetivo mais amplo (por exemplo, sua função em um sistema ou por que foi escrito dessa maneira)	Moderado	-2
7	Lógica não implementada, apenas placeholders presentes	Fatal	-4	18	Verifica se a explicação reconhece possíveis limitações, tratamento de erros ou casos de entrada inesperados	Menor	-1
8	Erros de tempo limite ou execução contínua do programa sem encerramento	Fatal	-4	19	Avalia a ortografia, a gramática e a fluência das frases	Menor	-1
9	Referência a variáveis ou funções indefinidas	Fatal	-4	20	Verifica se a explicação está alinhada com os comentários embutidos no código, as convenções de nomenclatura e o tom geral	Menor	-1
10	Outros erros	Pequeno	-1	21	Avalia se a explicação ajuda outras pessoas a aprender com ela — um desenvolvedor júnior ou alguém que não seja o autor consegue entendê-la facilmente?	Moderado	-2

**Prompt 1: Template do Laaj****Versão no código**

You will be provided with a problem statement, a code snippet and line by line explanation that supposedly addresses the problem, a reference solution, and a catalog of code and explanation inconsistencies.

**Scale of Value:**

“Negligible” = 0

“Small” or “Minor” = -1

“Moderate” = -2

“Major” = -3

“Fatal” = -4

**Evaluation Steps:**

1. Read the problem statement and reference solution carefully and identify the required functionalities for the implementation. The reference solution is used to help you understand the problem, do not directly compare the code snippet to the reference code.
2. Read the code snippet and compare it to the problem statement. Check if the code snippet covers the required functionalities.
3. Read the line by line explanation and code snippet checking if the explanation covers all lines and functionalities of the code snippet.
4. DO NOT duplicate inconsistencies.
5. Output Requirements (JSON Only). You MUST output your answer as a JSON list containing exactly one object with the following structure:
  - a) If the code snippet is correct, output: [{"inconsistency": "None", "severity": "Negligible"}].
  - b) If the code snippet is incorrect, output the identified inconsistencies and their severity according to the catalog of code and line by line explanation inconsistencies. For example: [{"inconsistency": "<inconsistency1>", "severity": "<severity1>"}, {"inconsistency": "<inconsistency2>", "severity": "<severity2>"}, ...]
  - c) If the line by line explanation covers all lines and functionalities of the code snippet, output: [{"inconsistency": "None", "severity": "Negligible"}].
  - d) If the line by line explanation is incorrect, output the identified inconsistencies and their severity according to the catalog of code and line by line explanation inconsistencies. For example: [{"inconsistency": "<inconsistency1>", "severity": "<severity1>"}, {"inconsistency": "<inconsistency2>", "severity": "<severity2>"}, ...]
  - e) In the end of JSON list MUST contain a final item been a “final\_answer”based on the Scale of Value.
  - f) DO NOT split the “Catalog of Code and line by line explanation Inconsistencies” into separate JSON schemas. They MUST be combined into a SINGLE JSON schema.
  - g) The “final\_answer” attribute MUST aggregate all results found across the entire analysis. Therefore, there MUST be ONLY ONE “final\_answer” field in the JSON schema (not multiple instances).
  - h) JSON schema [{"inconsistency": "<inconsistency1>", "severity": "<severity1>"}, {"inconsistency": "<inconsistency2>", "severity": "<severity2>"}, ..., "final\_answer": <numeric sum only>}]

**Problem:**

«PROBLEM»

Reference:

«CODE1»

Code Snippet:

«CODE2»

Catalog of Code and line by line explanation Inconsistencies:

0. Using different methods or algorithms to solve the problem: Negligible
1. Missing dependency declarations that are used in the program: Negligible
2. No exception handling for unexpected events, such as invalid inputs: Negligible
3. Including inefficient or unnecessary statements: Negligible
4. Failing to handle corner cases: Small
5. Logical errors leading to incorrect outputs: Major
6. Syntax errors or compilation errors: Fatal
7. Logic not implemented, only placeholders present: Fatal
8. Timeout errors or continuous program execution without termination: Fatal
9. Referencing undefined variables or functions: Fatal
10. Other mistakes: Small
11. Determines whether the explanation correctly describes what the code actually does: Fatal
12. Evaluates if all relevant parts of the snippet (variables, logic, conditions, imports, functions) are covered: Major
13. Assesses whether the explanation is understandable and readable by its intended audience (developers, reviewers, or learners): Major
14. Checks if the explanation goes beyond describing what happens to include why and how it happens, and the reasoning behind design decisions: Major
15. Evaluates whether correct programming terms and concepts (e.g., “method,” “iterator,” “mutable”) are used accurately and consistently: Moderate
16. Measures whether the explanation is well-organized (logical order, indentation, bullet points, separation of ideas): Moderate
17. Determines if the explanation relates the snippet to its broader purpose (e.g., its function in a system or why it was written this way): Moderate
18. Checks if the explanation acknowledges potential limitations, error handling, or unexpected input cases: Minor
19. Evaluates spelling, grammar, and sentence fluency: Minor
20. Verifies that the explanation aligns with the code’s inline comments, naming conventions, and overall tone: Minor
21. Assesses whether the explanation helps others learn from it — can a junior developer or non-author understand it easily?: Moderate

Evaluation Form:

JSON output (a JSON list only):

#### 4.2.4 Seleção dos LLMs

A seleção dos LLMs empregados neste trabalho é orientada por critérios metodológicos que visam garantir adequação ao desenho experimental, diversidade arquitetural e viabilidade de integração à execução dos experimentos. Consideraram-se, para isso, aspectos como capacidade de raciocínio, flexibilidade na construção e aplicação de *prompts*, suporte a diferentes formatos de entrada/saída e análise do desempenho reportado na literatura em tarefas de geração e explicação de código. Esses critérios não têm como objetivo avaliar os modelos em si, mas assegurar que os LLMs utilizados sejam capazes de sustentar a investigação proposta sobre combinações de técnicas de PE.

Ao longo da pesquisa, foram selecionados oito LLMs distintos, incorporados de forma progressiva à abordagem metodológica. Desse total, seis modelos foram utilizados nas tarefas de geração e explicação de código, constituindo a base empírica para a análise dos efeitos das diferentes combinações de técnicas de PE, enquanto dois modelos adicionais foram empregados exclusivamente no papel de avaliadores automáticos, no contexto da abordagem LaaJ. Essa estratégia permitiu ampliar o escopo da investigação de forma controlada, preservando a consistência metodológica e evitando que os próprios modelos responsáveis pela geração das respostas atuassem também como julgadores.

Entre os modelos selecionados, o GPT-4o-mini destaca-se pela eficiência computacional e ampla validação na literatura recente para tarefas de geração de código (PALLA; SLABY, 2025). O Llama-3.1-405b, com seus 405 bilhões de parâmetros, representa modelos de escala massiva otimizados para raciocínio complexo, oferecendo a oportunidade de investigar se a magnitude arquitetural se traduz em desempenho superior em domínios específicos (DEROY; MAITY, 2025). Além disso, a seleção desses modelos baseou-se em critérios<sup>18</sup> de equivalência em termos de porte, capacidade de raciocínio, flexibilidade na formulação de *prompts* e suporte a múltiplos formatos de entrada e saída, de modo a minimizar interferências externas na análise das combinações de PE.

Com o objetivo de ampliar a diversidade arquitetural considerada, também foram selecionados modelos adicionais, fundamentando-se em critérios de representatividade arquitetural e diversidade de estratégias de pré-treinamento e ajuste fino (do inglês, *fine-tuning*). O DeepSeek-v3<sup>19</sup> foi incluído por suas otimizações voltadas para tarefas técnicas e programação (SADIK; GOVIND, 2025). O Gemma3-27b representa uma abordagem intermediária em termos de escala, permitindo avaliar *trade-offs* entre tamanho e eficácia (TEAMGEMMA et al., 2025). O Mixtral-8x22b-instruct<sup>19</sup>, baseado em uma arquitetura *mixture-of-experts* (MoE), oferece uma perspectiva sobre a eficiência de modelos especializados (ZHUO et al., 2025). Finalmente, o Qwen2.5-72B foi selecionado por sua recente proeminência em benchmarks de código e disponibilidade para pesquisa acadêmica (TEAMQWEN et al., 2024; YANG et al., 2024). Essa

<sup>18</sup> Comparativo do Artificial Analysis

<sup>19</sup> DeepSeek-AI

ampliação teve como objetivo verificar a consistência do comportamento das combinações de técnicas de PE em diferentes arquiteturas de LLMs, sem deslocar o foco da análise para a comparação entre modelos.

Para viabilizar a integração desse conjunto heterogêneo de modelos sem demandar refatorações significativas no código experimental, analisou-se previamente a compatibilidade dos LLMs com um formato de entrada estruturado em JSON, detalhado na subseção seguinte. Essa compatibilidade permitiu a adoção de um mecanismo de integração mais flexível e agnóstico a provedores específicos, contribuindo para a escalabilidade e a reprodutibilidade da abordagem metodológica proposta.

Além dos modelos empregados na geração e explicação de código, foram selecionados dois LLMs distintos para atuar como avaliadores automáticos no paradigma LaaJ, viabilizando a análise conjunta dessas duas dimensões da resposta. A avaliação foi realizada por meio de duas execuções independentes, utilizando, respectivamente, os modelos Claude-Sonnet-4-5-20250929 e Gemini-2.5-Flash-Lite no papel de julgadores. A adoção de dois LLMs distintos como avaliadores teve como objetivo promover uma dupla verificação das respostas geradas, reduzindo vieses associados ao uso de um único julgador e evitando que os próprios modelos responsáveis pela geração e explicação do código avaliassem suas próprias saídas. Dessa forma, a análise permanece centrada no comportamento das combinações de técnicas de PE, preservando a imparcialidade e a robustez do processo avaliativo.

#### 4.2.5 *Framework*

A necessidade de concepção do *framework* emergiu a partir da etapa inicial da pesquisa, cujos resultados foram publicados em Girardi, Souza e Rêgo (2025). A experiência acumulada nessa fase evidenciou limitações práticas relacionadas à escalabilidade, à reutilização de código e à gestão de múltiplas configurações experimentais, o que motivou o desenvolvimento de uma infraestrutura mais flexível e estruturada.

No contexto desta pesquisa, o *framework* é concebido como uma estrutura de software, desenvolvida em Python, destinada a organizar e operacionalizar, de forma sistemática e padronizada, as etapas associadas à investigação experimental sobre combinações de técnicas de PE. Trata-se de um artefato metodológico cujo propósito central é viabilizar a execução controlada dos experimentos, assegurando consistência, rastreabilidade e reprodutibilidade ao longo de todo o processo investigativo.

A concepção do *framework* é motivada pela complexidade do desenho experimental a ser adotado, que envolve múltiplos cenários de combinação de técnicas de PE, a utilização de diferentes LLMs e a aplicação de distintos mecanismos de avaliação. Diante desse contexto, abordagens baseadas em *scripts* isolados ou soluções *ad hoc* mostram-se inadequadas, tanto do ponto de vista da manutenção quanto do controle metodológico. Ao centralizar funcionalidades como o carregamento e o pré-processamento dos dados, a aplicação das técnicas de *prompting*, a

integração com APIs de LLMs e o registro estruturado dos resultados, o *framework* contribui para a redução de redundâncias, a padronização das execuções e o fortalecimento da confiabilidade experimental — aspectos essenciais em estudos empíricos na área de inteligência artificial.

Com o objetivo de assegurar que o *framework* desenvolvido seja reutilizável, escalável e de fácil manutenção, sua especificação e implementação seguiram princípios consolidados da engenharia de software. Em particular, adotou-se a *Clean Architecture*<sup>20</sup>, que promove a separação explícita de responsabilidades entre camadas e o desacoplamento entre regras de negócio e detalhes de infraestrutura, favorecendo a independência em relação a *frameworks*, bibliotecas externas e provedores de LLMs. Complementarmente, empregou-se o padrão *Command Query Responsibility Segregation (CQRS)*<sup>21</sup>, permitindo distinguir de forma clara as operações de leitura e escrita, o que se mostrou especialmente adequado para o controle dos fluxos experimentais, do armazenamento de resultados e da recuperação de dados para análise.

O *framework* foi concebido em conformidade com os princípios do SOLID<sup>22</sup>, que orientam a construção de sistemas modulares, extensíveis e de fácil evolução. Esses princípios contribuíram para que novas técnicas de PE, modelos de linguagem ou estratégias de avaliação pudessem ser incorporados ao *framework* sem impactar negativamente componentes já existentes. Em conjunto, essas decisões arquiteturais reforçam a robustez do *framework* e garantem sua adequação como infraestrutura metodológica para a condução de experimentos complexos.

Dessa forma, o *framework* desenvolvido nesta dissertação representa uma evolução metodológica natural do trabalho, sendo posteriormente detalhado quanto à sua organização e funcionamento no âmbito do desenho experimental.

### 4.3 Desenho Experimental

Nesta seção, apresenta-se o desenho experimental adotado para operacionalizar as definições estabelecidas na Seção 4.2.

#### 4.3.1 Questões de Pesquisa e Objetivos dos Experimentos

As questões de pesquisa apresentadas na Introdução orientam diretamente a estrutura e a organização do desenho experimental adotado neste trabalho.

A QP1 norteia a investigação sobre como diferentes combinações de técnicas de PE influenciam a qualidade da geração de código quando avaliadas por métrica de similaridade textual. Para responder a essa questão, é estruturado o Experimento 1, etapa inicial da investigação, no qual são utilizados dois LLMs e treze combinações de técnicas de PE (cenários

---

<sup>20</sup> Clean Architecture

<sup>21</sup> CQRS

<sup>22</sup> SOLID Design Principles

experimentais). A avaliação concentra-se exclusivamente na geração de código em Python, empregando o BERTScore como métrica de análise.

A QP2 busca verificar se os efeitos observados nas combinações de técnicas de PE mantêm-se consistentes quando avaliados em um conjunto ampliado e heterogêneo de modelos. Com esse propósito, é delineado o Experimento 2, que mantém os treze cenários experimentais, porém amplia o número de LLMs para seis modelos distintos. Assim como no Experimento 1, a análise é conduzida sobre a geração de código, utilizando o BERTScore como métrica de avaliação, permitindo examinar a robustez dos resultados em diferentes arquiteturas.

A QP3 amplia a investigação ao considerar não apenas a geração, mas também a explicação de código, avaliadas por meio de *LLM-as-a-Judge*. Para responder a essa questão, é conduzido o Experimento 3, no qual são empregados os mesmos treze cenários experimentais e os seis LLMs utilizados na etapa anterior. Nesse experimento, adota-se o paradigma LaaJ, estruturado a partir de um conjunto de 22 critérios de avaliação. O Experimento 3 é subdividido em duas execuções independentes: a primeira utilizando o modelo Claude como julgador automatizado (Experimento 3.1) e a segunda utilizando o modelo Gemini no papel de julgador (Experimento 3.2), permitindo analisar a consistência das avaliações sob diferentes avaliadores automáticos.

A QP4, por sua vez, assume caráter transversal na pesquisa, uma vez que investiga o papel das técnicas menos exploradas quando combinadas a técnicas amplamente investigadas e recorrentes na literatura. Essa questão não está restrita a um único experimento, sendo examinada ao longo de todas as etapas experimentais, à medida que as diferentes combinações de técnicas de PE são analisadas sob distintos conjuntos de modelos e critérios de avaliação.

#### 4.3.2 *Framework* Experimental

Esta subseção apresenta os aspectos operacionais do *framework* experimental, contemplando a padronização das chamadas às APIs, a organização das estruturas de entrada e saída em formato JSON, o controle dos parâmetros experimentais e a consolidação das execuções realizadas ao longo dos experimentos. Dessa forma, busca-se evidenciar como as definições previamente estabelecidas foram instanciadas em condições controladas, viabilizando a execução dos experimentos e o registro estruturado dos resultados analisados neste trabalho.

A operacionalização do *framework* inicia-se pela camada de integração com os provedores de modelos de linguagem, responsável por padronizar as requisições e assegurar equivalência nas condições de execução entre diferentes APIs. Considerando a heterogeneidade dos serviços utilizados, definiu-se uma interface comum de invocação, composta por parâmetros gerais e por adaptações específicas conforme o provedor.

Entre os parâmetros gerais adotados, destacam-se: *model*, que identifica o LLM instanciado; *messages*, que encapsula a estrutura textual do *prompt*; *api\_key*, utilizada para autenticação;

e, quando aplicável, *url*, empregada nas requisições HTTP diretas. O hiperparâmetro *temperature*, responsável por controlar o grau de aleatoriedade ou criatividade das respostas geradas pelo modelo, seguiu o valor padrão definido na documentação da OpenAI<sup>23</sup> (igual a **zero**), não sendo explicitamente declarado nas requisições em que esse valor já é assumido por *default*. Tal escolha visa reduzir variações estocásticas — isto é, flutuações aleatórias nas respostas do modelo decorrentes de processos probabilísticos de amostragem — e favorecer maior consistência entre execuções sucessivas.

No caso da API da OpenAI, utilizou-se a biblioteca oficial do provedor, seguindo a estrutura recomendada de requisição por meio do método *chat.completions.create*. Para os LLMs disponibilizados pelo provedor ApiLLM<sup>24</sup> — incluindo LLaMA, DeepSeek, Gemma, Mixtral e Qwen —, embora exista compatibilidade declarada com a biblioteca da OpenAI, essa funcionalidade encontra-se desabilitada. Em razão disso, o *framework* empregou requisições HTTP diretas por meio da biblioteca *requests*, utilizando o parâmetro *url* para definir o *endpoint* específico de cada modelo. Ainda assim, manteve-se o mesmo padrão de parâmetros gerais, assegurando uniformidade metodológica na execução.

No caso dos modelos julgadores utilizados no paradigma LaaJ, foram necessárias adaptações adicionais. Para o Claude<sup>25</sup>, empregou-se a biblioteca oficial do provedor Anthropic, sendo explicitamente definidos os hiperparâmetros *max\_tokens* e *temperature*. Já para o Gemini<sup>26</sup>, a requisição foi realizada por meio da biblioteca específica do provedor Google, cuja interface apresenta menor número de parâmetros explícitos, exigindo tratamento particular na camada de integração do *framework*.

Essa padronização da camada de integração permitiu abstrair diferenças técnicas entre provedores, mantendo consistência experimental e garantindo que as variações observadas nos resultados estejam associadas às combinações de técnicas de PE, e não a divergências nas configurações das APIs. Os trechos de código correspondentes encontram-se no Apêndice B.

As mensagens enviadas aos LLMs foram organizadas em formato JSON, conforme a estrutura exigida pelas APIs utilizadas neste estudo. O *framework* operou com uma representação padronizada composta pelos campos *role* e *content*, responsáveis por indicar, respectivamente, o participante da interação e o texto associado ao *prompt*.

De forma geral, a estrutura utilizada pode ser representada conforme o esquema seguinte:

---

<sup>23</sup> OpenAi - Responses API

<sup>24</sup> LLMAPI - Responses API (deactivated)

<sup>25</sup> Claude - Responses API

<sup>26</sup> Gemini - Responses API

### Estrutura em JSON

```
{  
  "role": [Tipo de papel],  
  "content": [Texto do prompt]  
}
```

O campo *role* assumiu valores distintos conforme o provedor e o contexto de uso. Nos modelos empregados para geração e explicação de código por meio das APIs compatíveis com OpenAI e ApiLLM, foram utilizados predominantemente os valores “*system*” e “*user*”. O valor “*system*” foi destinado às instruções estruturantes — como a definição do papel do modelo — enquanto “*user*” armazenou a entrada textual ou o *prompt* fornecido pelo usuário. Em implementações específicas, como determinados modelos da família LLaMA, foi necessário utilizar o valor “*assistant*”, conforme exigência da respectiva API.

O campo *content*, por sua vez, armazenou o texto efetivo do *prompt* associado ao papel indicado. Esse conteúdo variou conforme o cenário experimental, podendo incluir apenas a definição do papel do modelo ou incorporar instruções adicionais decorrentes das combinações de técnicas de PE adotadas.

No cenário de referência ( $S_0$ ), essa estrutura foi mantida, diferenciando-se apenas pelo conteúdo inserido no campo *content* associado ao papel do modelo, que incluiu exclusivamente a sentença “*You are a programming tutor in Python.*”. Nos demais cenários ( $S_1$ – $S_{12}$ ), o mesmo formato foi preservado, porém com a inserção da versão expandida do *Role Prompting*. Assim, a distinção entre os cenários reside no conteúdo do *prompt*, e não na estrutura da requisição.

No contexto das execuções que envolveram LLMs utilizados como julgadores automáticos no paradigma LaaJ, observou-se uma particularidade estrutural. Para o Claude, manteve-se o envio em formato JSON com *role* definido como “*user*”. Já o Gemini não adota essa organização baseada em *role* e *content*, exigindo o envio direto da mensagem textual ao método de geração da resposta avaliativa. Essa diferença representou a única exceção à estrutura adotada, sendo tratada internamente pelo framework para assegurar uniformidade no armazenamento e no processamento das saídas obtidas.

Além da padronização das mensagens de entrada, o *framework* também controlou a organização das respostas retornadas pelos modelos. Considerando que as saídas produzidas envolvem tanto trechos de código em Python quanto explicações em linguagem natural, adotou-se uma estrutura de resposta em formato JSON, conforme discutido por Hada et al. (2024).

Nos experimentos em que se aplicou a métrica BERTScore, os *prompts* continham instruções explícitas para que os LLMs estruturassem suas respostas segundo um *schema* previamente definido. Essa estrutura separava intencionalmente: a resposta completa, contendo código e explicação (*all\_answer*); e o código-fonte isolado (*program\_created*), destinado à comparação com o *ground-truth*, conforme o *Prompt 2*.

**Prompt 2: Saída do LLM****Versão no código**

You must return a JSON object only, in the following format:

```
{
  "all_answer": {full result},
  "program_created": {code generated and
  line-by-line explanation}
}
```

The “all\_answer” field must contain your full response with all relevant information and the code included.

The “program\_created” field must contain only the Python source code.

You must not provide any output or text outside of this JSON structure.

**Versão traduzida**

Você deve retornar apenas um objeto JSON, no seguinte formato:

```
{
  "all_answer": {resultado completo},
  "program_created": {código gerado e expli-
  cação linha por linha}
}
```

O campo “all\_answer” deve conter sua resposta completa com todas as informações relevantes e o código incluído.

O campo “program\_created” deve conter apenas o código-fonte Python.

Você não deve fornecer nenhuma saída ou texto fora dessa estrutura JSON.

Durante a execução, o *framework* realizou a extração automática do campo “*program\_created*”, isolando exclusivamente o código-fonte Python para posterior processamento no módulo de avaliação quantitativa.

No experimento que utilizou o paradigma LaaJ, a instrução de saída incorporada ao *prompt* foi modificada pelo *framework* — *Prompt 3* —, resultando em uma formatação distinta da adotada nos experimentos anteriores. Essa alteração decorreu da mudança no objetivo avaliativo: enquanto nos experimentos com BERTScore buscava-se comparar o código gerado com um *ground-truth* específico, no paradigma LaaJ passou-se a avaliar conjuntamente a geração e a explicação de código, considerando critérios qualitativos definidos no próprio *prompt* de julgamento.

Em razão disso, o modelo deixou de ser instruído a produzir uma resposta estruturada em JSON com campos separados e passou a receber uma orientação explícita para gerar exatamente um bloco de código Python, no qual cada linha deveria conter, imediatamente ao lado, um comentário explicativo em formato *inline*. Essa configuração permitiu que o conteúdo produzido fosse posteriormente submetido ao modelo julgador, preservando a integração entre código e explicação em uma única representação textual.

**Prompt 3: Saída do LLM****Versão no código**

Your response must follow these rules:

- Output exactly one Python code block.
- Inside the code block, each line of Python code must be immediately followed by a comment (# ...) explaining that line.
- The explanation must stay on the same line, as an inline comment.
- Explanations **MUST** be clear and precise.

**Versão traduzida**

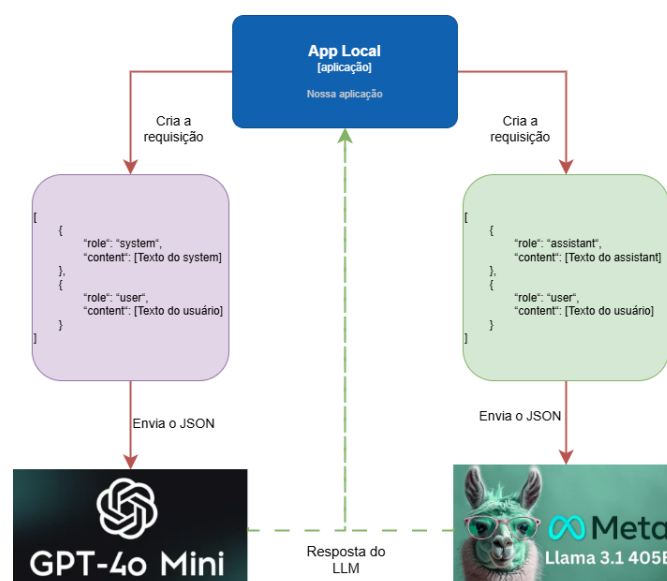
Sua resposta deve seguir estas regras:

- Produza exatamente um único bloco de código em Python.
- Dentro do bloco de código, cada linha de código Python deve ser imediatamente seguida por um comentário (# ...) explicando essa linha.
- A explicação deve permanecer na mesma linha, como um comentário inline.
- As explicações **DEVEM** ser claras e precisas.

No âmbito da orquestração experimental, o Experimento 1 corresponde à etapa inicial da investigação empírica e foi executado antes da consolidação do *framework* experimental descrito nesta subseção. Seus resultados foram publicados por Girardi, Souza e Rêgo (2025), com código-fonte disponibilizado no GitHub<sup>27</sup>.

Nessa fase, a execução limitou-se a dois LLMs, cujas integrações foram implementadas de forma específica para cada API. As requisições foram estruturadas individualmente em formato JSON, sem a utilização de uma camada comum de abstração entre os modelos. A Figura 11 apresenta o *workflow* correspondente, evidenciando o fluxo operacional adotado nessa etapa.

Figura 11 – *Workflow* do Experimento 1



Elaborada pelo Autor

<sup>27</sup> GitHub - Artigo

A avaliação concentrou-se na geração de código por meio da métrica BERTScore, cuja aplicação prática é detalhada na Subseção 4.3.5. As pontuações obtidas na comparação entre o código gerado e o *ground-truth* foram registradas em arquivos CSV, possibilitando a consolidação e a análise quantitativa dos resultados.

Embora funcional, essa abordagem apresentava limitações relacionadas à escalabilidade, à reutilização de código e ao gerenciamento de múltiplas configurações experimentais, especialmente diante da perspectiva de ampliação do número de modelos. Essas restrições evidenciaram a necessidade de uma infraestrutura mais padronizada para sustentar as etapas subsequentes da investigação, conduzindo à implementação do *framework* experimental descrito anteriormente.

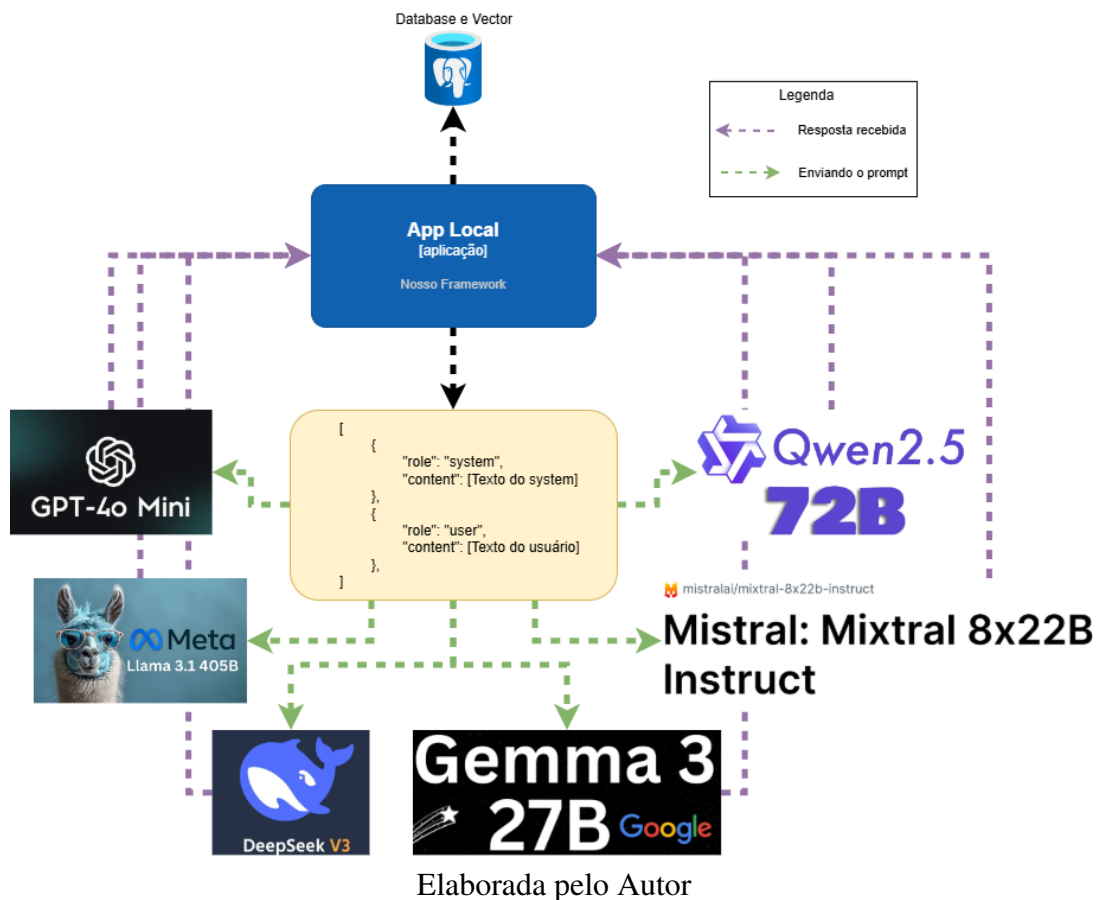
O Experimento 2 foi conduzido já sob a infraestrutura consolidada do *framework* experimental, representando a primeira aplicação integral da camada de abstração implementada. Nessa etapa, as requisições aos modelos passaram a ser intermediadas por uma interface comum, responsável por padronizar a construção das mensagens, o envio às APIs e o tratamento das respostas retornadas. O conjunto de modelos foi ampliado para seis LLMs, mantendo-se constantes o conjunto de questões, as combinações de técnicas de PE e o protocolo de execução. A padronização promovida pelo *framework* possibilitou a integração de diferentes provedores por meio de uma estrutura unificada de entrada em JSON. A partir dessa consolidação, o campo *role* passou a ser tratado de forma uniforme para os modelos de geração, permitindo um único formato de requisição no âmbito do experimento.

Essa estrutura foi aplicada exclusivamente à tarefa de geração de código, no Experimento 2. O fluxo operacional contemplou a construção dinâmica dos prompts conforme cada cenário, o envio padronizado às APIs e a extração do campo correspondente ao código gerado para posterior processamento pela métrica BERTScore. A ativação desse fluxo ocorreu por meio dos parâmetros de configuração do *framework*, com `use_bertscore = true`, direcionando a execução para o eixo quantitativo de avaliação da geração.

Outro avanço operacional relevante foi a integração de uma camada de persistência relacional (PostgreSQL), utilizada para armazenar os conjuntos de dados experimentais, as respostas geradas e os resultados das avaliações, garantindo rastreabilidade e organização sistemática das saídas.

A Figura 12 ilustra o *workflow* correspondente ao Experimento 2, evidenciando a sequência de construção do *prompt*, chamada às APIs, processamento da métrica e persistência dos resultados.

Figura 12 – Workflow do Framework do Experimento 2



O Experimento 3 manteve a mesma infraestrutura operacional estabelecida no Experimento 2, porém estendeu o fluxo para contemplar a avaliação conjunta da geração e da explicação de código por meio do paradigma LaaJ. Nessa etapa, os seis modelos de geração permaneceram inalterados, e a unificação do campo *role* foi mantida nas requisições destinadas à geração das respostas. A diferença operacional residiu na inclusão de um módulo adicional de avaliação automática.

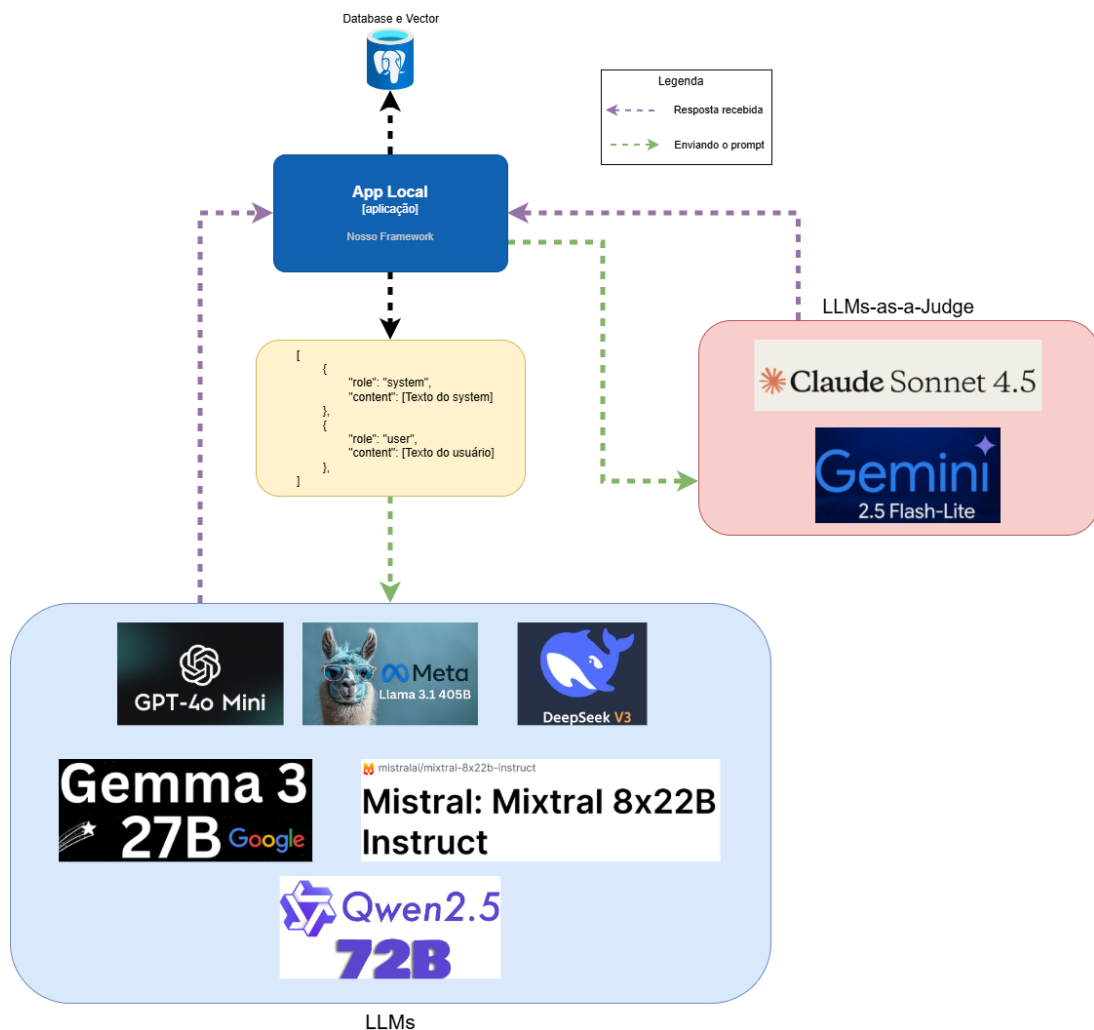
A execução foi controlada por parâmetros específicos do *framework*. Quando configurado com *use\_laaj = true*, o sistema direcionava o fluxo para o módulo de julgamento automático. O parâmetro *model\_laaj* determinava o avaliador utilizado, assumindo o valor “**claude**” ou “**gemini**” conforme a rodada executada.

O Experimento 3 foi subdividido em duas etapas operacionais independentes. Na etapa 3.1, utilizando o Claude como avaliador, o *framework* realizou a geração das respostas (código e explicação) e persistiu na camada de armazenamento relacional tanto as saídas parciais quanto os resultados finais da avaliação. Na etapa 3.2, com o Gemini como avaliador, o *framework* reutilizou as respostas previamente armazenadas, submetendo-as ao novo julgador sem necessidade de regeneração do código e da explicação. Essa estratégia garantiu consistência entre as avaliações e eliminou variações decorrentes de múltiplas gerações.

Durante essa etapa, o *framework* registrou de forma estruturada as respostas geradas pelos modelos e os pareceres emitidos pelos avaliadores automáticos, possibilitando o rastreamento completo do fluxo de geração e julgamento ao longo das execuções.

A Figura 13 apresenta o *workflow* correspondente ao Experimento 3, evidenciando o fluxo integrado de geração, persistência, encaminhamento para avaliação e consolidação dos resultados sob a infraestrutura do *framework*.

Figura 13 – *Workflow* do *Framework* do Experimento 3



Elaborada pelo Autor

### 4.3.3 Infraestrutura de Dados Experimental

O conjunto de dados adotado neste trabalho foi extraído do estudo conduzido por Gouveia et al. (2023), composto por uma coleção de perguntas e respostas relacionadas a problemas de programação em Python, desenvolvidas por alunos do PPGTI (GOUVEIA; ALBUQUERQUE, 2021). Essa base foi construída a partir de interações em uma ferramenta educacional voltada ao treinamento de estudantes para competições de programação, como a Maratona de Programação

da Sociedade Brasileira de Computação (SBC), cuja metodologia é fortemente baseada na resolução prática de problemas (*hands-on*).

O banco de dados original foi disponibilizado em formato JSON, contendo 457 chaves e 9.575 instâncias documentais organizadas de forma hierárquica (dados aninhados). Para adequá-lo aos objetivos deste trabalho — centrados na geração e explicação de código — foi necessário realizar um processo de seleção e filtragem dos dados, envolvendo: a identificação das entidades relevantes, a eliminação de dados sensíveis ou irrelevantes, a redução do número de colunas e a organização das instâncias pertinentes. Como resultado, foram selecionadas as entidades “*account.user*”, “*judge.problem*” e “*judge.submission*”, conforme apresentado na Tabela 5, que concentram informações essenciais sobre usuários, problemas de programação e submissões realizadas. Após esse refinamento, o número de colunas foi reduzido de 457 para 26.

Tabela 5 – Entidades selecionadas

Entidade	Descrição
“ <i>account.user</i> ”	Usuários
“ <i>judge.problem</i> ”	Perguntas
“ <i>judge.submission</i> ”	Respostas

Na entidade *judge.submission*, restringiu-se a seleção às soluções escritas exclusivamente em Python, reduzindo o total de instâncias de 9.575 para 2.943. Posteriormente, realizou-se a limpeza dos dados, removendo registros com valores nulos e selecionando apenas submissões com veredito “*Accepted*”, identificado no campo *fields.judge\_result.judge\_verdict*.

Na entidade “*account.user*”, foi conduzida uma análise para identificar os usuários com maior volume de submissões corretas. Destacaram-se dois perfis: o usuário 1 (superusuário) – identificado pelo campo “*fields.is\_superuser = True*”, e o usuário 51 – responsável pelo maior número de submissões corretas. A partir dessa identificação, foi realizada a junção entre as entidades *account.user* e *judge.submission*, permitindo associar submissões corretas aos respectivos usuários.

Com base nessa associação, os dados foram organizados em dois conjuntos metodologicamente distintos: (i) o conjunto de referência (*ground-truth*), composto por 70 instâncias selecionadas segundo critérios de validação e confiabilidade; e (ii) o conjunto de teste, formado por 74 instâncias independentes, destinado à execução dos experimentos. Embora ambos os conjuntos sejam derivados da mesma base original, eles não correspondem ao mesmo subconjunto de registros, tendo sido construídos com finalidades experimentais distintas.

Durante essa etapa, foram identificados casos de duplicidade, nos quais uma mesma questão apresentava múltiplas respostas corretas. Para evitar ambiguidade, manteve-se apenas uma instância por questão, removendo-se as duplicidades de forma aleatória. Adicionalmente, visando reduzir potenciais vieses experimentais, adotou-se a seguinte estratégia: as 67 instâncias

do usuário 51 foram utilizadas para geração das respostas pelos LLMs, enquanto as 49 instâncias do usuário 1 foram empregadas na construção da base vetorial utilizada na técnica RAG.

O conjunto final utilizado nos experimentos é composto pelas seguintes colunas:

Tabela 6 – Dicionário de dados

Nome da coluna	Descrição
“title”	Título da questão
“desc”	Descrição da questão
“input_desc”	Parâmetros relativos à entrada de dados da questão
“output_desc”	Saída esperada
“dicas”	Dicas da questão
“program”	Resposta correta

A configuração de todos os experimentos foi conduzida a partir de 10 instâncias de perguntas e respostas selecionadas aleatoriamente do conjunto de dados de referência, sem considerar o nível de dificuldade. Ainda assim, a distribuição foi equilibrada: 20% básicas, 40% intermediárias e 40% avançadas, garantindo diversidade e imparcialidade para compor cada rodada experimental.

No Experimento 1, o conjunto de dados foi manipulado por meio de arquivos CSV, utilizados simultaneamente como base de referência (*ground-truth*) e como conjunto de questões de teste. Nessa etapa inicial, não havia integração com um Sistema Gerenciador de Banco de Dados (SGBD) e a persistência dos resultados ocorreu de forma simplificada, por meio de arquivos estruturados. Para a técnica RAG, foi empregada uma base vetorial (VectorDB), utilizada na recuperação de contexto semanticamente relacionado às consultas submetidas aos modelos.

Com a consolidação do *framework* experimental, nos Experimentos 2 e 3, adota-se o PostgreSQL como Sistema Gerenciador de Banco de Dados (SGBD) relacional para centralizar o armazenamento e o gerenciamento das informações experimentais. O SGBD passou a ser utilizado como camada de persistência responsável por armazenar as instâncias do *ground-truth*, as questões de teste, as respostas geradas pelos modelos, os resultados das avaliações automáticas e os respectivos metadados experimentais associados a cada execução — como modelo utilizado, cenário, parâmetros ativados e informações temporais —, bem como por oferecer suporte à base vetorial empregada na técnica RAG. Essa integração permite maior rastreabilidade das execuções, controle das configurações adotadas e organização sistemática dos dados produzidos.

No âmbito dessa infraestrutura, foi definido um esquema relacional composto por tabelas específicas destinadas à organização e persistência dos dados experimentais. A modelagem adotada assegura separação entre dados de referência, instâncias submetidas aos modelos, respostas geradas, métricas de avaliação e informações temporais de execução. A Tabela 7 apresenta uma visão sintética das principais tabelas que compõem a camada de dados utilizada

nos experimentos.

Tabela 7 – Estrutura relacional da infraestrutura de dados experimental

Tabela	Descrição
“master_rag”	<i>Ground-Truth</i> e <i>embeddings</i> utilizados na técnica RAG
“master_quest”	Questões de Testes
“answers”	Respostas originais e Resultados finais
“metrics”	Resultados das Avaliações
“scenarios_time”	Informações temporais das execuções

A tabela “*master\_rag*” concentra os dados de referência, incluindo enunciado, descrição, entradas, saídas esperadas, resposta correta e os vetores semânticos associados. Para viabilizar a recuperação semântica necessária à técnica RAG, foi instalada a extensão *pgvector* no PostgreSQL, permitindo o armazenamento de *embeddings* em colunas do tipo *vector(1536)*. Complementarmente, foram empregados campos do tipo *TSVECTOR* para suporte a mecanismos de busca textual completa (do inglês, *full-text search*), integrando recuperação lexical e vetorial em uma única infraestrutura relacional.

A tabela “*master\_quest*” armazena as instâncias efetivamente utilizadas nos experimentos como entradas para os modelos. Já a tabela “*answers*” registra tanto a resposta original retornada pelo modelo quanto o conteúdo processado utilizado nas avaliações. A tabela “*metrics*” consolida os valores numéricos das métricas aplicadas, incluindo as pontuações de similaridade do BERTScore e os resultados agregados do LaaJ. Por fim, a tabela “*scenarios\_time*” armazena informações temporais associadas às execuções de cada cenário experimental em relação aos LLMs utilizados, permitindo monitoramento e análise de desempenho.

A integração entre o *framework* experimental e o PostgreSQL ocorreu por meio de uma camada de persistência responsável por registrar automaticamente as execuções realizadas, armazenar respostas parciais e finais, recuperar instâncias para processamento e disponibilizar os *embeddings* para a técnica RAG. No Experimento 2, foram persistidos os resultados finais da geração de código, associados às respectivas combinações de técnicas e modelos utilizados. No Experimento 3, passaram a ser armazenadas tanto as respostas geradas quanto os resultados intermediários e finais das avaliações conduzidas no paradigma LaaJ. Essa arquitetura garantiu consistência operacional, reprodutibilidade dos experimentos e separação clara entre dados de entrada, saídas geradas e avaliações automatizadas, sustentando de forma robusta a condução empírica desta pesquisa.

#### 4.3.4 Cenários Experimentais

Esta subseção descreve os procedimentos operacionais adotados para a execução dos cenários experimentais, bem como as configurações de *prompts* resultantes das combinações de técnicas de PE definidas na Subseção 4.2.2. Cada cenário corresponde a uma instanciação

prática dessas combinações, aplicada de forma sistemática no contexto da geração e explicação de código. Os cenários foram estruturados de modo a refletir diferentes níveis de complexidade e de direcionamento ao modelo, possibilitando a análise comparativa dos efeitos das estratégias propostas.

No cenário de referência ( $S_0$ ), a técnica  $P_1$  — *Prompt 1* — foi instanciada de forma mínima, com o objetivo de indicar explicitamente o papel do modelo como **tutor de programação**, sem induzir qualquer outro tipo de assistência ou direcionamento adicional ao raciocínio. Esse cenário foi utilizado como *baseline*, servindo como referência para a comparação com os demais cenários experimentais.

Os exemplos de *prompts* apresentados a seguir foram elaborados no contexto desta dissertação e desenvolvidos especificamente para a tarefa de geração e explicação de código em Python. Cada *prompt* é apresentado, quando aplicável, em sua redação original em inglês, acompanhada da respectiva tradução para o português (pt-BR), com o objetivo de garantir clareza, reprodutibilidade e acessibilidade.

*Prompt 4: Baseline ( $S_0$ )*

**Versão no código**

You are a programming tutor in python.

**Versão traduzida**

Você é tutor de programação em python.

Nos demais cenários de combinações ( $S_1 - S_{12}$ ), a técnica  $P_1$  — *Prompt 2* — foi instanciada de forma ampliada. Além da definição explícita do papel de **tutor de programação**, foi adicionado um complemento descritivo a esse papel, seguindo a proposta definida por Reynolds e McDonell (2021). Esse complemento descreve a função atribuída ao modelo e apresenta instruções adicionais com o objetivo de orientar seu raciocínio durante a análise da questão proposta pelo usuário, favorecendo respostas mais contextualizadas e precisas.

A sentença inicial define o papel profissional assumido pelo modelo, enquanto o restante do *prompt* especifica responsabilidades como a análise cuidadosa da pergunta, a verificação de código-fonte quando presente, a organização e execução do código e a geração de explicações claras.

**Prompt 5: Cenários ( $S_1 \dots S_{12}$ )****Versão no código**

You are a Python programming tutor. Your job is to analyze the user's question carefully and, if the question includes source code, check it for syntax errors, organize and structure the code properly, execute it, and provide the output along with a clear explanation of how the code works. You must also detect the language used in the user's question and always respond in that same language.

**Versão traduzida**

Você é um tutor de programação Python. Sua função é analisar cuidadosamente a pergunta do usuário e, se a pergunta incluir código-fonte, verificar se há erros de sintaxe, organizar e estruturar o código adequadamente, executá-lo e fornecer o resultado junto com uma explicação clara de como o código funciona. Você também deve detectar o idioma usado na pergunta do usuário e sempre responder nesse mesmo idioma.

Para a técnica **P<sub>2</sub>** — *Prompt 3* —, o prompt foi enriquecido com a instrução explícita que teve como objetivo induzir o modelo a reformular e expandir semanticamente a pergunta antes da geração da resposta, favorecendo uma interpretação mais precisa da tarefa.

**Prompt 6: RaR****Versão no código**

Rephrase and expand the question, and respond.

**Versão traduzida**

Reformule e expanda a pergunta, e responda.

Para a técnica **P<sub>3</sub>** — *Prompt 4* —, foi adicionada ao prompt essa instrução, com o intuito de estimular um raciocínio encadeado durante a resolução da tarefa.

**Prompt 7: ZS-CoT****Versão no código**

Let's think step by step.

**Versão traduzida**

Pense passo a passo

Entretanto, no caso específico dos modelos da OpenAI<sup>28</sup>, descritos na Subseção 4.2.4, observa-se uma orientação distinta. A documentação oficial recomenda evitar instruções que estimulem explicitamente a geração de cadeias de pensamento, uma vez que esses modelos já realizam o raciocínio internamente. Assim, expressões como “*think step by step*” ou “*explain your reasoning*” são desencorajadas, pois podem levar o modelo a expor conteúdos intermediários indesejados ou incompatíveis com as melhores práticas de segurança e eficiência adotadas pela plataforma. Em contrapartida, os demais LLMs selecionados para esta pesquisa não apresentam essa peculiaridade.

<sup>28</sup> OpenAI doc – How to prompt reasoning models effectively

Na técnica **P<sub>4</sub>**, a pergunta do usuário foi decomposta em três linhas de raciocínio distintas, permitindo que os LLMs processassem essas etapas em paralelo e gerassem respostas intermediárias diferentes. Os resultados intermediários foram então concatenados, compondo uma resposta final única e estruturada.

O primeiro raciocínio — *Prompt 5* — consistiu em extrair e organizar as informações principais da pergunta do usuário. Essa etapa teve como objetivo identificar os elementos essenciais da tarefa apresentada. Os itens considerados incluíram:

- **Extraia a pergunta principal:** nos casos em que o usuário incluiu múltiplas perguntas em um único *prompt*, o LLM foi instruído a identificar a questão central a ser respondida;
- **Identifique quaisquer dicas rotuladas como “Dicas”:** o termo “Dicas”, refere-se aos dados extraídos do banco de dados de (GOUVEIA et al., 2023), após a análise realizada nesta dissertação. Caso o usuário forneça dicas auxiliares, o LLM deve considerá-las como para enriquecer sua compreensão contextual;
- **Detecte se há dados de entrada que o usuário deve incluir na pergunta:** se houver dados de entrada específicos informados pelo usuário, o modelo precisa reconhecê-los e utilizá-los como base para o raciocínio;
- **Determine que tipo de pergunta é essa.:** embora seja um critério mais subjetivo, essa etapa visa categorizar a natureza da questão (por exemplo, se é sobre listas, laços de repetição, funções etc.), o que auxilia na adequação da resposta.
- **Identifique qual é o resultado esperado.:** o LLM deve inferir o formato da resposta desejada com base nas instruções do usuário (por exemplo, imprimir uma string, retornar um valor etc.).
- **Defina o objetivo principal da pergunta.:** essa etapa reforça a anterior, buscando alinhar o raciocínio do LLM com a real intenção da pergunta.

**Prompt 8: SoT – Primeiro raciocínio****Versão no código**

Regarding the user's question:

- Extract the main question.
- Identify any tips labeled as "Dicas."
- Detect if there are input data that the user must include in the question.
- Determine what type of question it is.
- Identify what output is expected.
- Define the main goal of the question.

**Versão traduzida**

Em relação à pergunta do usuário:

- Extraia a pergunta principal.
- Identifique quaisquer dicas rotuladas como "Dicas".
- Detecte se há dados de entrada que o usuário deve incluir na pergunta.
- Determine que tipo de pergunta é essa.
- Identifique qual é o resultado esperado.
- Defina o objetivo principal da pergunta.

O segundo raciocínio — *Prompt 6* — consistiu na geração de um código em Python com base na pergunta do usuário, incorporando comentários explicativos diretamente no código. Nesse ponto, a pergunta original foi reintroduzida no processo, uma vez que a técnica SoT baseou-se na execução paralela de diferentes linhas de raciocínio para compor uma única resposta consolidada. Para isso, a variável *self.question* foi utilizada para armazenar e reutilizar o *prompt* do usuário em cada etapa da geração, garantindo que o contexto fosse preservado ao longo do processo.

**Prompt 9: SoT – Segundo raciocínio****Versão no código**

Generate a Python program that performs the following task: {self.question}.

Use all available information to answer the user's question accurately.

Include clear, line-by-line comments in the code to explain its functionality.

**Versão traduzida**

Crie um programa Python que execute a seguinte tarefa: {self.question}.

Use todas as informações disponíveis para responder com precisão à pergunta do usuário.

Inclua comentários claros, linha por linha, no código para explicar sua funcionalidade.

O terceiro raciocínio — *Prompt 7* — envolveu a geração de um novo código nos casos em que o usuário forneceu um código de exemplo. Nessa situação, o modelo foi orientado a analisar o código existente, compreender seu funcionamento e, com base na pergunta original, produzir uma versão alternativa ou aprimorada, mantendo a lógica do problema. Sempre que possível, essa etapa incluiu a adição de explicações adicionais na forma de comentários no código gerado.

**Prompt 10: SoT – Terceiro raciocínio****Versão no código**

If sample data is provided, your task is to evaluate the samples, generate the appropriate Python code to handle them, and include line-by-line explanations for clarity.

**Versão traduzida**

Se forem fornecidos dados de amostra, sua tarefa é avaliar as amostras, gerar o código Python apropriado para lidar com elas e incluir explicações linha por linha para maior clareza.

Para a técnica **P<sub>5</sub>** — *Prompt 8* — o prompt foi reenviado ao modelo três vezes. Essa estratégia visou induzir o modelo a reavaliar iterativamente sua própria resposta, refinando o conteúdo gerado a cada nova iteração. As respostas produzidas em cada uma das três iterações foram posteriormente avaliadas por meio das estratégias de avaliação definidas neste trabalho, sendo selecionada como resultado final aquela considerada de maior qualidade.

**Prompt 11: SV****Versão no código**

Please rewrite the question and answer to provide a better response.

**Versão traduzida**

Reescreva a pergunta e a resposta para fornecer uma resposta melhor.

No caso da técnica **P<sub>6</sub>**, foi utilizada uma base vetorial construída a partir de um conjunto de dados armazenado localmente. Considerando que os modelos empregados não possuíam conhecimento prévio sobre as perguntas e respostas específicas desse conjunto, foram gerados *embeddings* a partir do *prompt* do usuário, os quais foram utilizados para realizar a busca por similaridade semântica na base vetorial. A base vetorial foi parametrizada para retornar os três pares de pergunta–resposta semanticamente mais semelhantes, os quais foram incorporados ao *prompt* original antes da geração da resposta pelo modelo.

Com base nessas instâncias operacionais, foram operacionalizados os cenários experimentais **S<sub>1</sub>** a **S<sub>12</sub>**, descritos a seguir, os quais diferem entre si pelas combinações de técnicas empregadas e pelo grau de complexidade imposto ao processo de geração e explicação de código.

No cenário **S<sub>1</sub>**, combinam-se a técnica *Role Prompt* com o RAG. A construção do *prompt* envolve a inclusão do papel do modelo, a busca de perguntas e respostas similares ao texto elaborado pelo usuário na base vetorial, a inserção do *prompt* fornecido pelo usuário e a estrutura da resposta esperada conforme o comportamento do LLM. Todos esses elementos são organizados dentro de uma lista, respeitando o formato de entrada exigido pelas APIs, garantindo que o modelo processe as informações de forma integrada.

**S<sub>1</sub> - Role e RAG**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do RAG] + [Saida em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario]
  }
]
```

No cenário **S<sub>2</sub>**, são utilizadas três técnicas de *prompt* em conjunto: *Role Prompting*, RaR e RAG. A construção do *prompt* inclui a definição do papel do modelo, a reformulação e expansão da pergunta do usuário conforme a técnica RaR, além da adição de contexto por meio da recuperação de informações relevantes da base vetorial (RAG). O *prompt* final integra esses elementos juntamente com o enunciado original do usuário e o tipo de resposta esperada.

**S<sub>2</sub> - Role Prompting, RaR e RAG**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do RAG] + [Saida em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [Conteúdo do RaR]
  }
]
```

Os cenários **S<sub>3</sub>** e **S<sub>4</sub>** empregam a técnica ZS-CoT, porém com abordagens distintas. O cenário **S<sub>3</sub>** aplica a técnica de forma direta, sem adicionar contexto adicional, utilizando apenas a definição do papel do modelo (*Role*), a instrução sobre o tipo de resposta esperada e a pergunta do usuário. Por outro lado, o cenário **S<sub>4</sub>** busca aprimorar a compreensão contextual ao combinar a técnica ZS-CoT e RAG, estimulando o modelo a realizar um raciocínio mais estruturado e progressivo.

S<sub>3</sub> e S<sub>4</sub>**Cenário S<sub>3</sub>:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saída em
      ↪ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [Conteúdo
      ↪ do ZS-CoT]
  }
]
```

**Cenário S<sub>4</sub>:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do
      ↪ RAG] + [Saída em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [Conteúdo
      ↪ do ZS-CoT]
  }
]
```

A partir deste ponto, os *prompts* adotados nos cenários tornaram-se mais complexos, exigindo a criação de *workflows* ilustrativos para representar graficamente como cada cenário interage com os LLMs. Esses fluxos, disponibilizados no Apêndice C, contribuem para a visualização das etapas envolvidas na construção e no processamento dos *prompts*.

No cenário S<sub>5</sub>, foi empregada a técnica SoT, com o objetivo de obter diferentes respostas do modelo com base em distintas interpretações da pergunta feita pelo usuário. Para isso, a questão é decomposta em múltiplas linhas de raciocínio, permitindo que o modelo gere respostas paralelas a partir de cada uma dessas perspectivas. Em seguida, tais respostas são concatenadas e reenviadas ao modelo juntamente com a pergunta original, promovendo um refinamento na geração da resposta final.

S<sub>5</sub> - Role, SoT e RAG**Parte 1:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saída em
      ↪ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [
      ↪ Raciocínio (1 ou 2 ou 3)]
  }
]
```

**Parte 2:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do
      ↪ RAG] + [Saída em JSON]
  },
  {
    "role": "user",
    "content": [Todas as respostas
      ↪ concatenadas]
  }
]
```

No cenário S<sub>6</sub>, foi realizada uma combinação do papel de sistema com a técnica de RAG, na qual tanto a resposta proveniente do RAG quanto a pergunta original do usuário foram inseridas como entrada no LLM. Com o objetivo de assegurar maior precisão na resposta gerada, implementou-se um mecanismo de SV, detalhado na subseção anterior. Nesse processo, o modelo produziu e avaliou sua própria resposta ao longo de três iterações consecutivas. Ao

final, a resposta com a maior pontuação, segundo a métrica BERTScore ou LaaJ, foi selecionada como resultado final.

#### S<sub>6</sub> - Role, RAG e SV

##### Prompt sem SV parte 1:

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do
      ↪ RAG] + [Saida em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario]
  }
]
```

##### As respostas submetido ao SV parte 2:

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saida em
      ↪ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [Conteúdo
      ↪ do Self-Verification]
  }
]
```

No cenário S<sub>7</sub>, foi adotada uma combinação integrada das técnicas *Role Prompting*, RaR, SoT e RAG, organizadas em um processo estruturado em três etapas principais:

1. Primeiramente, são enviados ao modelo os componentes correspondentes ao papel, à técnica RaR e à pergunta do usuário;
2. Em seguida, como o SoT opera de forma paralela, são enviados três raciocínios distintos, cada um derivado da resposta obtida na etapa anterior;
3. Por fim, todas as respostas geradas são concatenadas e enriquecidas com o conteúdo recuperado via RAG, formando a base para a geração da resposta final, estruturada em formato JSON.

#### S<sub>7</sub> - Role, RaR, SoT e RAG

##### Prompt com RAR parte 1:

```
[
  {
    "role": ["system" ou "assistant"],
    "content": [Tipo de papel]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [Conteúdo
      ↪ do RaR]
  }
]
```

##### Prompt com SoT parte 2:

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saida em
      ↪ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [
      ↪ Raciocinio (1 ou 2ou 3)]
  }
]
```

**Prompt Final:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do RAG] + [Saída em JSON]
  },
  {
    "role": "user",
    "content": [Todas as respostas concatenadas]
  }
]
```

No cenário **S<sub>8</sub>**, são combinadas as técnicas de *Role Prompting*, RAG, RaR e SV. A execução é organizada em duas etapas principais:

1. Na primeira etapa, são enviados ao modelo os elementos correspondentes ao papel atribuído (*Role*), à técnica de recuperação de informações relevantes (RAG) e à estratégia de reformulação e resposta (RaR), compondo o *prompt* inicial que será inferido pelo modelo de linguagem;
2. Na segunda etapa, aplica-se a técnica de SV, que solicita ao modelo a reescrita e reavaliação de suas próprias respostas. Esse processo é repetido três vezes, permitindo a seleção da melhor resposta com base na pontuação mais alta atribuída pela métrica BERTScore ou LaaJ.

**S<sub>8</sub> - Role, RaR, RAG e SV****Prompt sem SV parte 1:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do
      ↪ RAG] + [Saída em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [Conteúdo
      ↪ do RaR]
  }
]
```

**As respostas submetido ao SV parte 2:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saída em
      ↪ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [Conteúdo
      ↪ do Self-Verification]
  }
]
```

No cenário **S<sub>9</sub>**, foi realizada a combinação das técnicas *Role Prompting*, ZS-CoT, SoT e RAG. O principal objetivo deste cenário consiste em integrar as abordagens ZS-CoT e SoT, enviando os *prompts* correspondentes de forma paralela ao modelo de linguagem. Após a obtenção das respostas individuais geradas por cada linha de raciocínio, realiza-se a unificação dessas saídas, que, por sua vez, compõem um novo *prompt*. Este novo *prompt* é então submetido ao modelo utilizando a técnica de RAG, com o intuito de gerar a resposta final de maneira mais contextualizada.

S<sub>9</sub> - Role, ZS-CoT, SoT e RAG**Prompt Role, SoT e ZS-CoT parte 1:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saida em
      ↔ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [
      ↔ Raciocinio (1 ou 2ou 3)] + [
      ↔ Conteudo do ZS-CoT]
  }
]
```

**Adicionando o RAG parte 2:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteudo do
      ↔ RAG] + [Saida em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario]
  }
]
```

No cenário S<sub>10</sub>, ocorre a combinação das técnicas *Role Prompting*, ZS-CoT, SV e RAG. Inicialmente, é enviada uma requisição ao modelo contendo o papel atribuído (*Role*), o raciocínio em cadeia do tipo ZS-CoT e a pergunta do usuário. Na etapa seguinte, aplica-se a técnica de SV e RAG para adicionar mais contexto, na qual o modelo é instanciado três vezes para reavaliar sua própria resposta, com o objetivo de selecionar a saída com a melhor pontuação, medida por meio da métrica BERTScore ou LaaJ. Esse processo busca aprimorar a precisão e coerência da resposta final.

S<sub>10</sub> - Role, ZS-CoT, SV e RAG**Prompt Role e ZS-CoT parte 1:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saida em
      ↔ JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [Conteudo
      ↔ do ZS-CoT]
  }
]
```

**As respostas submetidas ao SV parte 2:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteudo do
      ↔ RAG] + [Saida em JSON]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [Conteudo
      ↔ do Self-Verification]
  }
]
```

O cenário S<sub>11</sub> combina as técnicas *Role Prompting*, RaR, SoT, SV e RAG. Esse cenário foi estruturado para operar em quatro etapas distintas:

1. Agrupamento das informações de *Role*, RaR, RAG e a pergunta do usuário;
2. Considerando que a técnica de SoT atua de forma paralela para garantir múltiplas interpretações, cria-se um novo *prompt* contendo apenas *Role*, os esboços de raciocínio (do inglês, *thoughts*) e a pergunta do usuário;

3. Fusão das respostas obtidas nas partes 1 e 2, solicitando ao modelo a geração da resposta final em formato JSON;
4. Aplicação da técnica de SV, por meio da geração de três respostas distintas, selecionando-se aquela com a maior pontuação segundo a métrica BERTScore ou LaaJ.

#### S<sub>11</sub> - Role, RaR, SoT, SV e RAG

##### **Prompt com Role, RaR e RAG parte 1:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteúdo do
      ↪ RAG]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [Conteúdo
      ↪ do RaR]
  }
]
```

##### **Prompt com SoT parte 2:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [
      ↪ Raciocínio (1 ou 2 ou 3)]
  }
]
```

##### **Prompt que unifica respostas parte 3:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saída em
      ↪ JSON]
  },
  {
    "role": "user",
    "content": [Texto concatenado]
  }
]
```

##### **Prompt com SV parte 4:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saída em
      ↪ JSON] + [Texto concatenado]
  },
  {
    "role": "user",
    "content": [Texto do usuário] + [Conteúdo
      ↪ do Self-Verification]
  }
]
```

No cenário S<sub>12</sub>, as técnicas *Role Prompting*, ZS-CoT, SoT, SV e RAG foram combinadas para compor a proposta experimental. O objetivo principal foi preservar a integridade e a contribuição individual de cada técnica, como já observado no cenário S<sub>9</sub>, porém com a intenção de elevar a qualidade da resposta sem aumentar a complexidade do fluxo de execução. O processo está estruturado em três etapas:

1. Execução paralela das requisições, de acordo com a técnica SoT, integrando as informações de *Role Prompting* e ZS-CoT;
2. Agregação de todas as respostas geradas, inclusão da técnica RAG e formatação do conteúdo resultante em formato JSON;
3. Aplicação da técnica de SV, gerando três versões alternativas da resposta final e selecionando aquela com a maior pontuação, com base na métrica BERTScore ou LaaJ.

S<sub>12</sub> - Role, ZS-CoT, SoT, SV e RAG**Prompt parte 1:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [
      ↪ Raciocinio (1 ou 2ou 3)] + [
      ↪ Conteudo do ZS-CoT]
  }
]
```

**Prompt com RAG e unifica parte 2:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Conteudo do
      ↪ RAG] + [Saida em JSON]
  },
  {
    "role": "user",
    "content": [Texto concatenado]
  }
]
```

**Prompt Final:**

```
[
  {
    "role": ["system"],
    "content": [Tipo de papel] + [Saida em JSON] + [Texto concatenado]
  },
  {
    "role": "user",
    "content": [Texto do usuario] + [Conteudo do Self-Verification]
  }
]
```

## 4.3.5 Avaliação BERTScore e LaaJ

Nesta subseção, descreve-se a operacionalização das estratégias de avaliação adotadas nos experimentos, detalhando o fluxo de execução das métricas BERTScore e do paradigma LaaJ no âmbito do *framework* experimental.

A avaliação da geração de código foi operacionalizada por meio da métrica BERTScore, empregada para medir a similaridade semântica entre o código produzido pelos modelos e o código de referência (*ground-truth*). O procedimento consistiu na conversão de ambos os códigos em *embeddings* vetoriais e por meio do cálculo de similaridade de cosseno entre os vetores correspondentes.

Para essa finalidade, foi utilizado o modelo *microsoft/codebert-base*, especializado em representações semânticas de código-fonte, em conjunto com a biblioteca *torch*, responsável pelo cálculo da similaridade por cosseno entre os *embeddings* gerados, resultando em um único valor escalar de similaridade para cada par de códigos comparados.

Operacionalmente, a avaliação envolveu três etapas principais: (i) extração automática do código-fonte gerado a partir da estrutura de saída padronizada em JSON, especificamente do campo “*program\_created*”, conforme descrito na Subseção 4.3.2; (ii) recuperação do código correspondente no conjunto *ground-truth*, armazenado na tabela “*master\_rag*” do SGBD PostgreSQL; (iii) aplicação da métrica de similaridade e registro estruturado das pontuações obtidas,

com persistência dos resultados na tabela *metrics* do mesmo SGBD.

No Experimento 1, as pontuações calculadas foram igualmente registradas em arquivos CSV, sem integração com sistema gerenciador de banco de dados. A partir do Experimento 2, com a consolidação da infraestrutura de dados descrita anteriormente, a avaliação passou a integrar o fluxo automatizado do *framework*, utilizando o PostgreSQL como mecanismo central de persistência para armazenamento das instâncias de referência e das métricas calculadas.

No Experimento 3, o paradigma LaaJ foi integrado ao *framework* experimental como um módulo adicional de avaliação automática, acionado por meio de parâmetros específicos de configuração. Quando o parâmetro *use\_laa\_j = true* era ativado, o fluxo de execução direcionava as respostas geradas pelos modelos ao LLM julgador definido no parâmetro *model\_laa\_j*, assumindo os valores “claude” ou “gemini”, conforme a rodada experimental executada.

O processo avaliativo compreendeu as seguintes etapas operacionais: (i) recuperação do enunciado do problema (*PROBLEM*), da solução de referência (*CODE1*) e da resposta gerada pelo modelo (*CODE2*), contendo código e explicação; (ii) composição da requisição segundo o *prompt template* descrito na Subseção 4.2.3 e envio ao LLM julgador; (iii) recebimento da saída estruturada exclusivamente em formato JSON.

A saída exigida do julgador consistiu em um *array* de objetos contendo pares (“*inconsistency*”, “*severity*”) para cada inconsistência identificada, além de um campo final obrigatório denominado “*final\_answer*”, correspondente ao somatório numérico das severidades atribuídas.

O campo “*final\_answer*” representa a soma das penalizações associadas às inconsistências identificadas, conforme a escala de severidade descrita na Subseção 4.2.3. Trata-se de uma métrica de natureza cumulativa, assumindo valores no intervalo  $[-\infty, 0]$ . O valor 0 indica ausência de inconsistências, enquanto valores progressivamente mais negativos refletem maior quantidade e/ou gravidade de desvios identificados no par código-explicação avaliado. Não há valores positivos nessa métrica, uma vez que ela expressa exclusivamente penalizações acumuladas. O valor final é calculado diretamente pelo modelo julgador, mediante a agregação das penalizações atribuídas durante o processo de avaliação.

Nos cenários experimentais que empregaram a técnica SV, três versões alternativas de resposta eram geradas pelo modelo. Cada uma dessas versões era submetida independentemente ao julgador LaaJ, resultando em três valores distintos no campo “*final\_answer*”. O *framework* realizava então a seleção automática da versão com melhor desempenho — isto é, aquela com menor penalização acumulada (valor mais próximo de zero) — registrando tanto os resultados parciais quanto o valor final consolidado.

Os resultados da avaliação — incluindo inconsistências individuais e valores agregados — foram registrados na infraestrutura de dados experimental, possibilitando análises comparativas entre modelos, cenários experimentais e avaliadores.

Com os fundamentos conceituais estabelecidos e a operacionalização experimental

devidamente estruturada, encontram-se consolidadas as condições necessárias para a investigação empírica proposta. A articulação entre técnicas de PE, modelos de linguagem e estratégias de avaliação cria um ambiente controlado que permite observar, de forma comparável e sistemática, os efeitos das diferentes combinações adotadas. No capítulo seguinte, os resultados obtidos são apresentados e analisados à luz das questões de pesquisa, buscando evidenciar padrões, contrastes e implicações decorrentes das escolhas metodológicas realizadas.

#### 4.4 Execução Experimental

Ao todo, foram conduzidos três experimentos ao longo desta pesquisa, estruturados de modo a investigar, de forma progressiva, os efeitos das combinações de técnicas de engenharia de prompts em tarefas de geração e explicação de código, em condições controladas de execução experimental.

O Experimento 1 produziu 480 resultados, resultantes da combinação de 10 questões, 2 iterações, 12 cenários e 2 LLMs ( $10 \times 2 \times 12 \times 2$ ), às quais se somam 40 resultados adicionais correspondentes ao cenário de referência (*baseline*), totalizando 520 saídas. No Experimento 2, foram incorporados quatro modelos adicionais, ampliando o conjunto experimental para 1.560 saídas, obtidas a partir de 10 questões, 2 iterações, 13 cenários e 6 LLMs, incluindo a *baseline* ( $10 \times 2 \times 13 \times 6$ ).

O Experimento 3 foi subdividido em duas etapas complementares: o Experimento 3.1, no qual se empregou o modelo Claude no paradigma de LaaJ, e o Experimento 3.2, em que o modelo avaliador foi substituído pelo Gemini. Em ambas as etapas, foram consideradas as mesmas configurações experimentais, sendo que, no Experimento 3.2, foram reutilizadas as respostas previamente geradas no Experimento 3.1, assegurando comparabilidade direta entre os julgadores automáticos. Assim como no Experimento 2, cada uma dessas etapas totalizou 1.560 resultados.

A Tabela 8 sintetiza a configuração geral dos experimentos conduzidos, incluindo os modelos de geração utilizados, os avaliadores adotados, as tarefas executadas e o volume de saídas produzido em cada etapa.

Tabela 8 – Configuração dos Experimentos

Experimento	LLMs (Geração)	LLM Avaliador	Tarefa	Avaliação	Cenários	Iterações	Infraestrutura	Saídas
Exp. 1	GPT-4o-mini, Llama-3.1-405B	–	Geração de código	BERTScore	13	2	Integração específica por provedor	520
Exp. 2	GPT-4o-mini, Llama-3.1-405B, DeepSeek-V3, Gemma3-27B, Mixtral-8x22B-instruct, Qwen2.5-72B	–	Geração de código	BERTScore	13	2	<i>Framework</i> + persistência relacional	1560
Exp. 3.1	Mesmos modelos do Exp. 2	Claude-Sonnet-4-5-20250929	Geração + explicação de código	Prompt Template + LaaJ	13	2	<i>Framework</i> + persistência relacional	1560
Exp. 3.2	Respostas do Exp. 3.1 (sem nova geração)	Gemini-2.5-Flash-Lite	Reavaliação (geração + explicação)	Prompt Template + LaaJ	13	2	<i>Framework</i> + persistência relacional	1560

A execução dos experimentos seguiu condições padronizadas, com o objetivo de assegurar consistência e comparabilidade entre cenários e modelos. Em todos os casos, adotou-se o hiperparâmetro temperatura (do inglês, *temperature*) igual a zero, reduzindo a variabilidade nas respostas geradas e minimizando efeitos decorrentes do caráter estocástico dos LLMs — isto é, flutuações aleatórias inerentes ao processo probabilístico de geração textual.

A decisão de executar duas iterações por cenário fundamenta-se na necessidade de obter múltiplas respostas para uma mesma entrada, possibilitando uma análise quantitativa mais robusta. Essa estratégia permite avaliar a consistência das saídas geradas e reduzir o impacto de variações estocásticas, capturando diferenças semânticas entre respostas e oferecendo uma base mais consistente para a aplicação do BERTScore e do paradigma LaaJ.

A execução foi realizada em ambiente computacional local, utilizando um equipamento Lenovo ThinkPad T14, com sistema operacional Windows 11, processador Intel Core i7 vPro e 16 GB de memória RAM. Embora as inferências tenham sido realizadas por meio de APIs externas dos provedores de LLMs, o ambiente local foi responsável pela orquestração das execuções, integração com o *framework* experimental, persistência dos dados e aplicação dos métodos de avaliação.

## 5 RESULTADOS E DISCUSSÕES

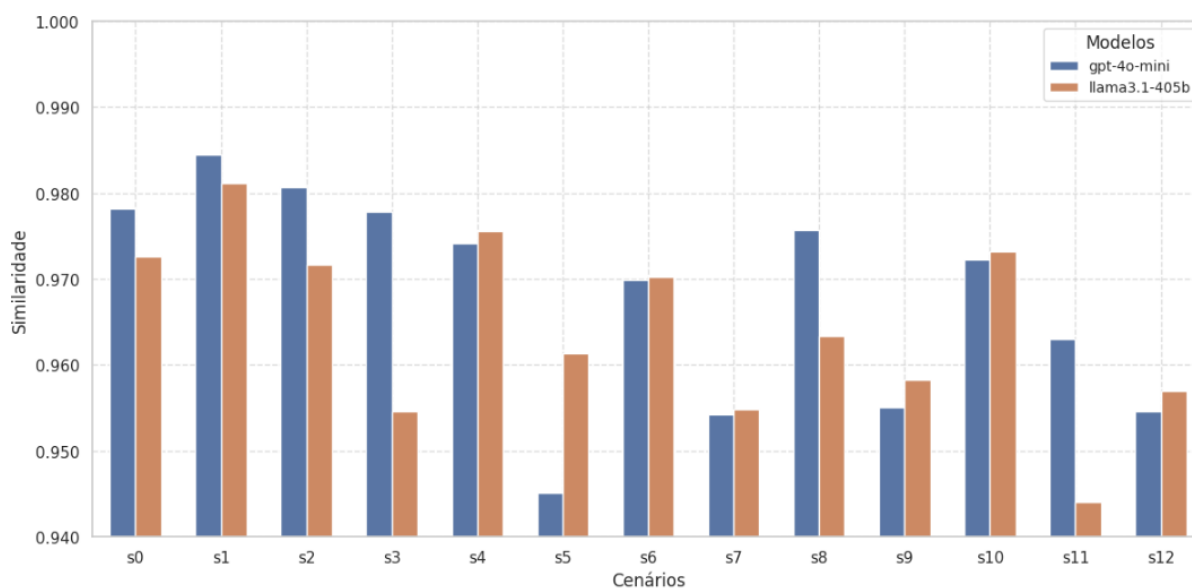
Este capítulo apresenta e discute os resultados obtidos nos três experimentos conduzidos nesta pesquisa, estruturados de forma progressiva para analisar os efeitos das combinações de técnicas de PE na geração e explicação de código. No Experimento 1, investigou-se o impacto inicial das combinações propostas sobre a geração de código, utilizando dois LLMs e avaliação baseada na métrica BERTScore. O Experimento 2 manteve as mesmas combinações e protocolo de execução, ampliando o conjunto de modelos avaliados para examinar a consistência dos resultados em diferentes arquiteturas. O Experimento 3 incorporou julgadores automáticos no paradigma LaaJ, permitindo avaliar conjuntamente geração e explicação de código.

### 5.1 Experimento 1

O Experimento 1 foi conduzido com o objetivo de responder à **QP1**, analisando como diferentes combinações de técnicas de PE influenciam a qualidade da geração de código quando avaliadas por meio do BERTScore. Foram analisados 13 cenários experimentais — incluindo um baseline como referência — aplicados a dois LLMs. Essa etapa estabelece a base empírica da investigação, permitindo observar o comportamento inicial das combinações propostas em um conjunto controlado de modelos.

A Figura 14 apresenta os resultados da avaliação de similaridade entre os códigos gerados pelos LLMs e o conjunto de referência, considerando diferentes combinações de técnicas de PE aplicadas aos modelos de linguagem GPT-4o-mini e Llama-3.1-405B. Ressalta-se que cada cenário experimental representa uma combinação específica de técnicas de PE, conforme definido na Subseção 4.3.4, e o desempenho foi mensurado por meio da métrica BERTScore. De forma geral, os modelos usados apresentaram altos níveis de similaridade na geração de código, com score mínimo de 0,940 em todos os cenários testados. Para a representação gráfica dos resultados, foi utilizada a média harmônica entre as duas saídas geradas por questão, conforme proposto por Rao, Shi e Wu (2014), por se tratar de uma medida menos sensível a valores extremos do que a média aritmética, proporcionando uma estimativa mais estável e interpretável, próxima ao comportamento da mediana.

Figura 14 – BERTScore - Comparação por Cenários x Modelos



Elaborada pelo Autor

Como as medidas estão expressas em milhares, também foi calculada a média harmônica do desvio padrão ( $sd$ ) entre todos os cenários ( $sd = 0,016$ ), fornecendo uma referência unificada para comparação. Para melhor visualização das diferenças de desempenho, o eixo Y foi ajustado para iniciar em 0,940, de maneira a perceber visualmente as variações entre os cenários.

O cenário  $S_1$  apresentou o melhor desempenho para ambos os modelos, alcançando 0,985 para o GPT-4o-mini e 0,981 para o Llama. No cenário  $S_0$  (*baseline*), que utiliza exclusivamente a técnica *Role Prompt*, já se observam resultados elevados, com 0,980 para o GPT-4o-mini e 0,964 para o Llama. Para o Llama, destacam-se ainda os cenários  $S_4$  (0,976) e  $S_{10}$  (0,973), que também apresentaram maiores scores de similaridade. No caso do GPT-4o-mini, entretanto,  $S_1$  permaneceu como o cenário de maior similaridade, superando significativamente os demais cenários avaliados.

No cenário  $S_2$  (*Role*, *RaR* e *RAG*), observou-se uma leve redução na similaridade no modelo Llama (0,971), enquanto o GPT-4o-mini manteve desempenho elevado (0,980). Esses resultados indicam que a combinação de múltiplas técnicas de forma extensiva nem sempre é benéfica e que estruturas de *prompt* mais enxutas podem favorecer o raciocínio dos modelos.

Nos cenários  $S_3$  e  $S_4$ , foi avaliado o impacto da adição de contexto semântico por meio das técnicas *ZS-CoT* e *RAG*. O cenário  $S_3$  (*Role* e *ZS-CoT*) resultou em desempenho inferior ao  $S_4$  (*Role*, *ZS-CoT* e *RAG*), especialmente no GPT-4o-mini. Com a introdução da técnica *RAG* no  $S_4$  — que forneceu um exemplo de saída esperada — o modelo Llama apresentou ganho expressivo, alcançando 0,975, resultado ligeiramente superior ao GPT-4o-mini (0,974). Esses dados sugerem que o modelo Llama se beneficia de contextos semânticos mais ricos para gerar respostas mais aderentes. Por outro lado, a técnica *ZS-CoT* demonstrou ser mais eficaz para o

modelo GPT-4o-mini do que para o Llama, conforme evidenciado nos resultados de similaridade.

No cenário  $S_5$  (*Role*, SoT e RAG), observou-se a maior queda de desempenho entre todos os cenários neste experimento, com valores de 0,961 para o Llama e 0,945 para o GPT-4o-mini. Essa redução pode ser atribuída ao uso da técnica SoT, que envia múltiplas requisições paralelas à API. Ambos os modelos foram executados sob condições de *Free Tier*, com recursos computacionais limitados; ainda assim, o Llama superou o GPT-4o-mini, sugerindo que a técnica SoT favoreceu seu desempenho em tarefas de raciocínio mais complexas.

No cenário  $S_6$  (*Role*, SV e RAG), os resultados foram praticamente equivalentes entre os dois modelos: Llama = 0,970 e GPT-4o-mini = 0,969. Já nos cenários  $S_7$  (*Role*, RaR, SoT e RAG) e  $S_9$  (*Role*, ZS-CoT, SoT e RAG), ambos os modelos apresentaram nova queda de desempenho, possivelmente associada ao mecanismo da técnica SoT. Nessa configuração, o problema é decomposto em múltiplas subconsultas enviadas paralelamente à API, cujas respostas são posteriormente recompostas em uma única saída. Esse processo aumenta o número de requisições, o consumo de *tokens* e a variabilidade das respostas intermediárias, podendo introduzir ruído na solução final — especialmente sob restrições computacionais de ambiente *Free Tier*.

No entanto, no cenário  $S_8$  (*Role*, RaR, SV e RAG), o GPT-4o-mini voltou a se destacar (0,975) em relação ao Llama (0,963), reforçando a hipótese de que o modelo da OpenAI responde melhor à combinação de técnicas baseadas em verificação (SV) e refinamento (RaR).

No cenário  $S_{10}$  (*Role*, ZS-CoT, SV e RAG), o modelo Llama obteve 0,973, superando o GPT-4o-mini (0,972) e também seu próprio desempenho no cenário *baseline*. Uma análise mais detalhada mostra que a presença da técnica de SV contribuiu para uma maior proximidade entre os códigos gerados e a implementação de referência, sugerindo que o uso dessa técnica tende a melhorar a correção dos resultados. Esses achados indicam que a incorporação de SV, de modo geral, favorece o alinhamento semântico e funcional entre a saída do modelo e a resposta de referência.

A comparação entre  $S_{11}$  e  $S_{12}$  evidencia que, mantendo-se constante a base composta por *Role*, SoT, SV e RAG, a técnica complementar — RaR no cenário  $S_{11}$  e ZS-CoT no cenário  $S_{12}$  — exerce impacto distinto sobre os modelos avaliados. No cenário  $S_{11}$ , o GPT-4o-mini alcançou 0,963, enquanto o Llama registrou 0,944 — seu menor desempenho em todo o experimento. Já no cenário  $S_{12}$ , os resultados mostraram-se mais equilibrados, com 0,954 para o GPT-4o-mini e 0,957 para o Llama. Esse comportamento sugere que a introdução de RaR sobre essa base estrutural pode aumentar a complexidade instrucional ou intensificar a interação entre estratégias de decomposição (SoT), verificação (SV) e recuperação (RAG), afetando de maneira diferenciada o desempenho dos modelos. Por outro lado, a substituição de RaR por ZS-CoT mostrou-se relativamente mais estável para o Llama, indicando que os efeitos das técnicas não são meramente aditivos, mas dependem da forma como cada modelo responde às interações estabelecidas entre as estratégias de PE empregadas.

A avaliação dos resultados contemplou a análise da similaridade do código gerado em relação ao *ground-truth*, bem como a verificação da significância estatística das variações observadas entre os cenários experimentais. Inicialmente, foi aplicado o teste de Kolmogorov-Smirnov (DODGE, 2008) para verificar a normalidade das distribuições dos valores de BERTScore obtidos pelos modelos GPT-4o-mini e Llama. Os resultados indicaram que ambas as distribuições não seguem comportamento normal ( $p < 0,05$ ), justificando a adoção de testes estatísticos não paramétricos nas comparações subsequentes.

Os gráficos que ilustram a distribuição dos valores de BERTScore para cada modelo, evidenciando visualmente o comportamento não normal identificado, encontram-se no Apêndice D.

Para comparar cada cenário ( $S_1 \dots S_{12}$ ) com o *baseline* ( $S_0$ ), foi aplicado o teste não paramétrico de Wilcoxon signed-rank (WOOLSON, 2005) para amostras pareadas. As hipóteses consideradas foram: (i) hipótese nula – inexistência de diferença significativa entre o *baseline* e o cenário comparado; e (ii) hipótese alternativa – existência de diferença estatisticamente significativa entre eles.

Para o GPT-4o-mini, observaram-se diferenças estatisticamente significativas ( $p < 0,05$ ) nos cenários  $S_5$ ,  $S_7$ ,  $S_{11}$  e  $S_{12}$ , indicando variações relevantes em relação ao *baseline*. Nos cenários  $S_1$ ,  $S_2$ ,  $S_3$ ,  $S_4$ ,  $S_8$  e  $S_{10}$  não foram identificadas diferenças significativas, sugerindo desempenho estatisticamente alinhado ao cenário de referência.

No caso do Llama, os cenários  $S_1$ ,  $S_3$ ,  $S_5$ ,  $S_7$ ,  $S_9$ ,  $S_{11}$  e  $S_{12}$  apresentaram diferenças significativas ( $p < 0,05$ ) em comparação ao *baseline*, enquanto  $S_2$ ,  $S_4$ ,  $S_6$ ,  $S_8$  e  $S_{10}$  não evidenciaram diferenças estatisticamente relevantes.

Parte desses resultados, referentes ao Experimento 1, foi previamente reportada em Girardi, Souza e Rêgo (2025), sendo aqui reapresentada de forma integrada ao conjunto completo das análises desenvolvidas neste trabalho.

Os resultados obtidos no Experimento 1 indicam que certas combinações de prompts podem, de fato, contribuir significativamente para a melhoria na qualidade das respostas dos modelos de linguagem, especialmente na geração de código em Python. Observou-se por exemplo, que o cenário  $S_1$  apresentou o maior índice de similaridade com o código de referência, mesmo sendo uma das combinações mais simples de PE avaliadas, sugerindo que os LLMs possuem capacidade de raciocínio suficiente para resolver problemas específicos sem depender de estruturas complexas de prompting. O cenário  $S_2$  também manteve alta similaridade, mesmo com a inclusão da técnica RaR. Contudo, observou-se queda de desempenho do modelo Llama em cenários onde o RaR foi novamente aplicado — comportamento não observado no GPT-4o-mini, que manteve desempenho estável. Esse contraste reforça que a interação entre técnicas pode variar conforme a arquitetura do modelo, motivando uma análise mais aprofundada sobre como essas combinações afetam cada LLM de maneira distinta.

Apesar de os cenários de maior complexidade ( $S_{11}$  e  $S_{12}$ ) apresentarem resultados numéricos menos expressivos, eles trouxeram uma perspectiva importante ao evidenciar a possibilidade de combinar múltiplas técnicas de PE. No entanto, o planejamento e a implementação desses cenários experimentais exigiram especial atenção, pois as combinações de técnicas podem introduzir ruídos ou interações inesperadas, comprometendo a clareza da instrução fornecida ao modelo.

Nos cenários intermediários ( $S_4 \dots S_{10}$ ) do experimento, identificou-se que diferentes combinações de técnicas resultam em variações significativas na similaridade do código gerado, a depender do LLM utilizado. Por exemplo, o GPT-4o-mini obteve melhor desempenho no cenário  $S_8$ , mas apresentou queda relevante em  $S_5$ , no qual a aplicação da técnica SoT elevou a complexidade operacional da execução, com maior número de requisições e consumo de recursos computacionais. Mesmo com acesso pago à API, ambos os modelos foram afetados por políticas de uso da camada *Free Tier*, exigindo controle no volume de requisições e até pausas nos testes para evitar erros.

Diante do exposto, em resposta à **QP1**, conclui-se que diferentes combinações de técnicas de PE influenciam de forma mensurável e diferenciada a qualidade da geração de código, impactando os níveis de similaridade semântica, conforme evidenciado pelos valores de BERTScore observados nos distintos cenários experimentais. Tais combinações podem produzir ganhos significativos, estabilidade ou queda de desempenho, dependendo tanto da estrutura do *prompt* quanto da arquitetura do modelo avaliado.

## 5.2 Experimento 2

O Experimento 2 foi conduzido com o objetivo de responder à **QP2**, investigando se os efeitos observados nas combinações de técnicas de PE na geração de código mantêm-se consistentes quando avaliados em um conjunto ampliado e heterogêneo de LLMs. Para isso, foram mantidos os 13 cenários experimentais e o mesmo conjunto de questões utilizado no experimento anterior, ampliando-se, contudo, o número de LLMs avaliados para 6 modelos distintos. A avaliação permaneceu centrada na geração de código em Python, utilizando o BERTScore como métrica automática de similaridade em relação ao *ground-truth*.

A Figura 15 apresenta os resultados obtidos por meio da métrica BERTScore nos diferentes cenários experimentais em relação aos LLMs avaliados. Observa-se que os modelos apresentaram desempenho consistentemente elevado, com scores variando entre 0,892 e 0,973. O modelo DeepSeek-v3 destacou-se como o melhor modelo independentemente do cenário de combinações, alcançando média geral de BERTScore = 0,964, seguido de perto pelo Mixtral-8x22b-instruct.

A melhor combinação individual foi observada no cenário  $S_6$  com o modelo Mixtral-8x22b-instruct, que obteve BERTScore médio de 0,973, com desvio padrão (*sd*) de  $\pm 0.0109$ ,

representando o maior valor registrado nesta etapa experimental. Por outro lado, o Llama3.1-405b apresentou os menores scores em comparação aos demais modelos, embora mantendo resultados acima de 0,892.

Como as medidas estão expressas em milhares, também foi calculada a média do desvio padrão entre todos os cenários ( $sd = 0,015$ ), fornecendo uma referência unificada para comparação da dispersão dos resultados.

Figura 15 – BERTScore - Média de Valores por Cenários x Modelos

Cenário	Qwen2.5-72B	deepseek-v3	gemma3-27b	gpt-4o-mini	llama3.1-405b	mixtral-8x22b-instruct
0	<b>0.965</b>	0.959	0.961	0.962	<b>0.932</b>	0.949
1	0.962	<b>0.966</b>	0.962	<b>0.966</b>	<b>0.941</b>	0.951
2	0.942	<b>0.969</b>	0.967	0.957	<b>0.925</b>	0.943
3	0.958	0.965	0.957	0.956	<b>0.927</b>	<b>0.968</b>
4	0.96	0.962	0.963	0.953	<b>0.925</b>	<b>0.969</b>
5	0.948	0.955	0.952	<b>0.956</b>	<b>0.926</b>	0.947
6	0.965	0.964	0.968	0.966	<b>0.935</b>	<b>0.973</b>
7	0.936	0.953	0.939	<b>0.954</b>	<b>0.929</b>	0.951
8	0.967	<b>0.971</b>	0.969	0.965	<b>0.944</b>	0.969
9	0.957	<b>0.961</b>	0.945	0.957	<b>0.892</b>	0.953
10	0.966	0.968	0.97	0.965	<b>0.943</b>	<b>0.971</b>
11	0.96	<b>0.971</b>	0.961	0.959	<b>0.937</b>	0.969
12	0.961	0.969	0.964	0.964	<b>0.936</b>	<b>0.97</b>

Legenda: verde = melhor score por cenário; vermelho = pior score por cenário.

Elaborada pelo Autor

A análise por cenário evidencia a variação leve no desempenho entre modelos, refletindo diferenças na sensibilidade às combinações de técnicas de PE. No cenário  $S_2$  (*Role*, RaR e RAG), o DeepSeek-v3 apresentou um de seus melhores resultados (0,969), enquanto o cenário  $S_6$  (*Role*, SV e RAG) favoreceu o Mixtral-8x22b-instruct, que alcançou seu maior score global (0,973). Em contraste, o cenário  $S_9$  (*Role*, ZS-CoT, SoT e RAG) resultou no menor desempenho observado para o llama3.1-405b (0,892), sugerindo maior sensibilidade desse modelo às características combinadas desse arranjo de técnicas.

O DeepSeek-v3 obteve o melhor desempenho em cinco cenários ( $S_1$ ,  $S_2$ ,  $S_8$ ,  $S_9$  e  $S_{11}$ ), indicando robustez consistente. O Mixtral-8x22b-instruct liderou em cinco cenários ( $S_3$ ,  $S_4$ ,  $S_6$ ,  $S_{10}$  e  $S_{12}$ ), enquanto GPT-4o-mini e Qwen2.5-72B apresentaram resultados tecnicamente

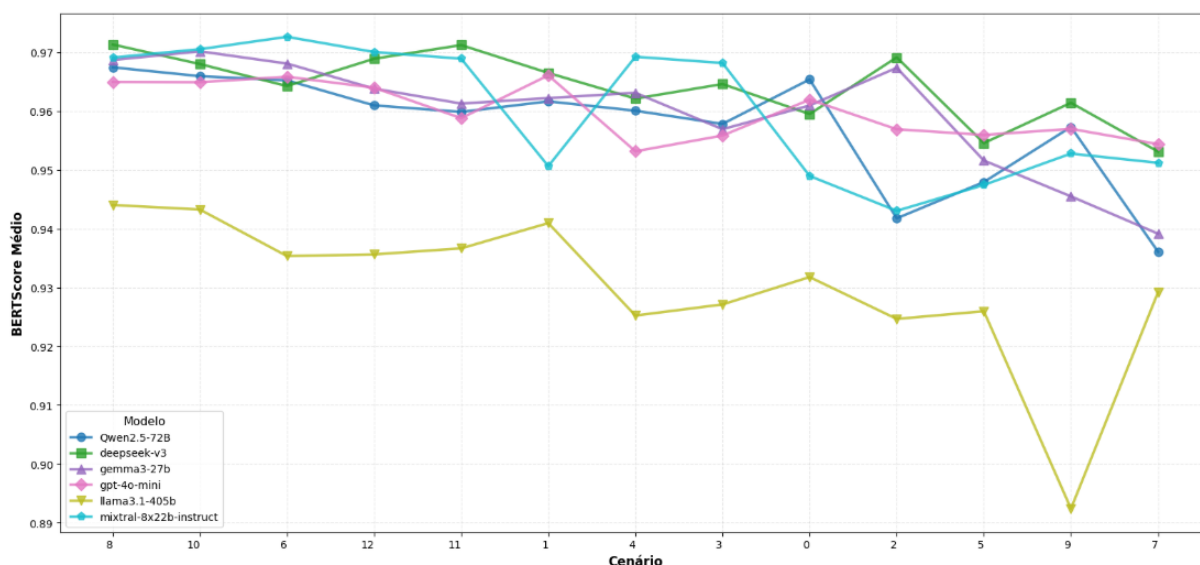
competitivos, frequentemente próximos aos valores máximos. O Gemma3-27b apresentou valores distribuídos de forma relativamente homogênea ao longo dos cenários, sem assumir a posição de maior score em nenhum cenário específico.

A variação dos scores de cada modelo ao longo dos diferentes cenários foi analisada a partir da amplitude entre seus valores mínimo e máximo. O DeepSeek-v3 apresentou scores entre 0,953 e 0,971, resultando em uma amplitude de 0,018. Já o Llama3.1-405b variou entre 0,892 e 0,944, com amplitude de 0,052. Essa diferença indica que o Llama3.1-405b apresentou maior variação de desempenho entre os cenários avaliados, enquanto o DeepSeek-v3 manteve resultados mais próximos entre si ao longo dos cenários. Tal comportamento sugere que os modelos respondem de maneira distinta às modificações estruturais dos *prompts*, com alguns apresentando maior sensibilidade às combinações empregadas.

De modo geral, os resultados confirmam competência ampla dos modelos na geração de código, porém com diferenças técnicas consistentes. DeepSeek-v3 e Mixtral-8x22b-instruct destacam-se como os modelos mais robustos, combinando altos valores absolutos com estabilidade entre cenários. A análise também indica que a seleção do modelo pode ser otimizada conforme o perfil da tarefa, dada a predominância distinta de diferentes arquiteturas em configurações específicas de PE.

A Figura 16 apresenta o comportamento do BERTScore médio dos seis modelos avaliados ao longo dos 13 cenários experimentais, organizados na sequência S<sub>8</sub>, S<sub>10</sub>, S<sub>6</sub>, S<sub>12</sub>, S<sub>11</sub>, S<sub>1</sub>, S<sub>4</sub>, S<sub>3</sub>, S<sub>0</sub>, S<sub>2</sub>, S<sub>5</sub>, S<sub>9</sub> e S<sub>7</sub>. A visualização permite observar como cada combinação de técnicas de PE impacta o desempenho dos modelos, evidenciando padrões de maior estabilidade em determinados cenários e maior variação em outros.

Figura 16 – BERTScore - Avaliação por Cenários x Modelos



Elaborada pelo Autor

Observa-se que DeepSeek-v3, Mixtral-8x22b-instruct e GPT-4o-mini apresentam curvas

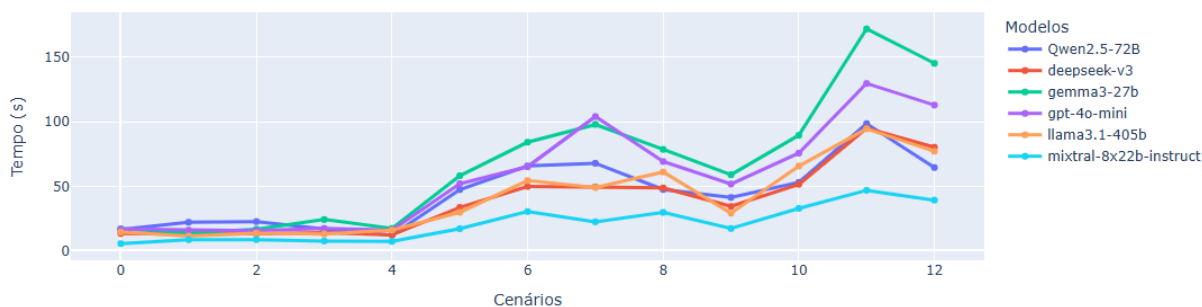
mais estáveis ao longo dos cenários, com variações discretas e desempenho consistentemente elevado. O GPT-4o-mini, em particular, mantém trajetória praticamente uniforme em todos os cenários experimentais, indicando baixa sensibilidade às combinações de técnicas de PE. De forma semelhante, o Gemma3-27b também exibe comportamento estável, com oscilações moderadas.

Em contraste, o Llama3.1-405b apresenta maior variabilidade, com quedas mais acentuadas em determinados cenários – especialmente em S<sub>9</sub> – e posterior recuperação parcial em S<sub>7</sub>, evidenciando maior sensibilidade às combinações de técnicas de PE. O Qwen2.5-72B também demonstra leve tendência de declínio nos cenários finais, embora sem oscilações tão abruptas quanto o Llama3.1-405b.

De modo geral, o gráfico evidencia que, embora todos os modelos apresentem desempenho elevado, há diferenças claras quanto à estabilidade ao longo das combinações de técnicas: alguns modelos mantêm comportamento consistente entre cenários, enquanto outros demonstram maior variação em função das configurações de PE empregadas.

Com respeito à análise dos tempos de execução, o gráfico de linhas da Figura 17, permite visualizar de forma intuitiva a dinâmica dos tempos de execução ao longo dos cenários, destacando tendências globais, pontos de inflexão e diferenças de comportamento entre os modelos.

Figura 17 – BERTScore - Tempo de Execução em (s) - Cenários x Modelos



Elaborada pelo Autor

A leitura geral do gráfico evidencia uma assimetria clara na eficiência temporal. O Mixtral-8x22b-instruct demonstra vantagem significativa em termos de velocidade, mantendo sua linha consistentemente abaixo das demais ao longo de praticamente todos os cenários. Sua trajetória apresenta inclinação suave e variação controlada, mesmo quando os cenários incorporam maior número de combinações de técnicas de PE. Esse comportamento indica alta estabilidade e uma ótima relação custo-tempo, reforçando seu potencial para aplicações que demandam baixa latência.

Em contraste, o Gemma3-27b apresenta o maior custo computacional, com uma curva acentuadamente crescente a partir do cenário S<sub>5</sub> e picos particularmente elevados nos cenários S<sub>11</sub> e S<sub>12</sub>, que concentram maior número de técnicas combinadas. O GPT-4o-mini segue tendência

semelhante, embora em menor magnitude, exibindo elevações mais pronunciadas nos cenários de maior densidade técnica. Qwen2.5-72B, DeepSeek-v3 e Llama3.1-405b ocupam uma faixa intermediária de desempenho temporal, com crescimento gradual do tempo de execução e menor volatilidade. Dentre esses, o DeepSeek-v3 demonstra maior estabilidade frente às variações de engenharia de *prompts*, sugerindo boa resiliência às diferentes configurações experimentais.

Um aspecto crucial para interpretar corretamente esses padrões é a presença da técnica de SV nos cenários S<sub>6</sub>, S<sub>8</sub>, S<sub>10</sub>, S<sub>11</sub> e S<sub>12</sub>. Essa técnica exige que os modelos executem uma etapa adicional de verificação da própria resposta, o que naturalmente aumenta o tempo de execução. Tal característica explica os “picos” observados de forma transversal nesses cenários, afetando todos os modelos — ainda que em diferentes magnitudes — e separando de maneira clara no gráfico as regiões de maior carga computacional.

A comparação conjunta entre tempo e qualidade revela que desempenho qualitativo e custo computacional não evoluem de forma diretamente proporcional. O Mixtral-8x22b-instruct, além de alcançar um dos melhores desempenhos em BERTScore, mostrou-se também o mais eficiente em tempo, registrando as menores médias entre todos os modelos. Já o Gemma3-27b, embora competitivo em termos de similaridade, exibiu o maior custo computacional. O DeepSeek-v3, por sua vez, combinou ótimo desempenho em BERTScore com posicionamento intermediário em tempo, indicando equilíbrio, embora sem atingir o mesmo nível de eficiência observado no Mixtral-8x22b-instruct. Esse panorama evidencia que modelos com alta similaridade semântica nem sempre são os mais rápidos, e que ganhos qualitativos podem vir acompanhados de maior custo temporal. Além disso, a presença de técnicas específicas — como a SV — influenciou transversalmente todos os modelos, ampliando significativamente o tempo de execução dos cenários em que foi aplicada.

A análise do BERTScore demonstra que os modelos apresentam competências relativamente semelhantes na geração de código, com diferenças que se tornam mais perceptíveis quando se observa a consistência ao longo dos cenários. Modelos como DeepSeek-v3 e Mixtral-8x22b-instruct mantiveram altos níveis de similaridade mesmo diante da variação das técnicas de PE, evidenciando maior robustez estrutural. Outros modelos, como Llama3.1-405b, apresentaram oscilações mais significativas ao longo dos cenários, sugerindo maior sensibilidade às mudanças estruturais introduzidas pelas diferentes combinações de técnicas de PE.

Como ilustração desse comportamento, apresenta-se no Apêndice E um exemplo de código extraído do cenário S<sub>6</sub>, que registrou o melhor resultado geral entre os modelos avaliados. Nesse contexto, observou-se reorganização estrutural da solução — como o uso de *list comprehension* e contagem posterior dos símbolos — sem prejuízo da lógica central do problema, evidenciando coerência estrutural e equivalência funcional em relação ao código de referência, mesmo diante de pequenas variações na formulação do prompt.

Dessa forma, os resultados indicam que o comportamento dos LLMs não pode ser interpretado exclusivamente a partir de métricas de similaridade. O impacto operacional, mensurado

pelo tempo de execução, constitui dimensão igualmente relevante para compreender como diferentes modelos respondem às variações estruturais dos *prompts* e às combinações de técnicas de PE empregadas.

Diante do exposto, em resposta à **QP2**, conclui-se que os efeitos das combinações de técnicas de PE observados na geração de código não se mantêm de forma homogênea quando avaliados em um conjunto ampliado e heterogêneo de LLMs. Ainda que seja possível identificar padrões recorrentes de desempenho em termos de similaridade semântica, tais padrões manifestam-se de maneira distinta conforme o modelo analisado e o cenário experimental considerado. Os resultados do Experimento 2 demonstram que algumas combinações de técnicas de PE favorecem, de maneira recorrente, níveis elevados de similaridade textual entre o código gerado e o *ground-truth*, como evidenciado pelo desempenho robusto e estável do DeepSeek-v3 e do Mixtral-8x22b-instruct ao longo da maioria dos cenários. Entretanto, a análise detalhada por modelo e por cenário evidencia que a magnitude desses ganhos em BERTScores varia substancialmente conforme as características arquiteturais de cada LLM, sendo particularmente sensível à complexidade das técnicas combinadas, como observado nos cenários que incorporam SV.

Além disso, as diferenças observadas nos tempos de execução reforçam que desempenho elevado em similaridade semântica não implica, necessariamente, eficiência computacional equivalente. Assim, embora determinadas combinações de PE apresentem resultados favoráveis em termos de BERTScore em múltiplos modelos, seus efeitos não se distribuem de maneira uniforme entre arquiteturas distintas, devendo ser analisados de forma conjunta com o modelo adotado e com as restrições operacionais do contexto de aplicação.

### 5.3 Experimento 3

O Experimento 3 foi delineado com o objetivo de responder à **QP3**, analisando como diferentes combinações de técnicas de PE impactam simultaneamente a qualidade da geração e da explicação de código quando avaliadas por LaaJ. Diferentemente dos experimentos anteriores — centrados na avaliação da geração de código por meio do BERTScore — esta etapa incorporou uma avaliação qualitativa estruturada, fundamentada em **22 critérios** previamente definidos na Subseção 4.2.3. Os mesmos seis LLMs responsáveis pela geração de código foram mantidos, enquanto dois modelos distintos atuaram como julgadores no paradigma LaaJ.

O experimento foi subdividido em duas execuções independentes: o Experimento 3.1, utilizando o modelo Claude como julgador, e o Experimento 3.2, utilizando o modelo Gemini, que avaliou as mesmas saídas previamente geradas, sem necessidade de nova geração e explicação de código. Essa organização permite não apenas responder à QP3 sob dois avaliadores distintos, mas também examinar comparativamente possíveis variações no julgamento automático, aspecto explorado na Subseção 5.3.3.

### 5.3.1 Experimento 3.1

A Figura 18 sintetiza o desempenho dos cenários experimentais quanto à geração e à explicação de código sob a perspectiva do modelo julgador Claude, denominado *Claude LaaJ* no contexto do paradigma LaaJ. A avaliação realizada pelo Claude LaaJ consistiu na identificação e quantificação das inconsistências previstas no catálogo descrito na Subseção 4.2.3.

Como os valores apresentados representam penalizações decorrentes dessas inconsistências, quanto mais próximos de zero estiverem, melhor é a qualidade global da resposta, indicando menor ocorrência de erros, contradições ou desalinhamentos entre o código produzido e sua respectiva explicação, de acordo com a Subseção 4.3.5.

Figura 18 – Claude LaaJ - Média de Valores por Cenários x Modelos

Cenário	Qwen2.5-72B	deepseek-v3	gemma3-27b	gpt-4o-mini	llama3.1-405b	mixtral-8x22b-instruct
0	-5.7	-6.25	-6.75	-7.3	-6.25	-7.2
1	-4.3	-5.05	-5.7	-4.45	-4.9	-4.95
2	-5.1	-4.1	-6.45	-4.3	-5.35	-4.95
3	-6.1	-7.45	-5.85	-6.75	-5.0	-6.85
4	-4.4	-4.5	-3.95	-5.75	-5.5	-4.75
5	-6.95	-7.45	-8.15	-6.45	-6.25	-7.0
6	-5.25	-4.85	-4.9	-7.35	-6.05	-6.95
7	-5.2	-6.75	-8.9	-8.5	-4.45	-9.55
8	-4.3	-3.15	-4.2	-3.35	-4.6	-3.9
9	-7.0	-6.8	-7.2	-7.4	-6.3	-8.25
10	-4.5	-2.9	-3.6	-3.6	-4.9	-4.55
11	-5.35	-3.2	-5.0	-4.9	-6.15	-5.0
12	-5.45	-3.1	-5.2	-6.05	-6.25	-3.45

Legenda: verde = melhor score por cenário; vermelho = pior score por cenário.

Elaborada pelo Autor

O modelo DeepSeek-v3 apresentou o melhor desempenho geral em termos de geração e explicação de código, destacando-se em 6 dos 13 cenários avaliados (cenários S<sub>2</sub>, S<sub>6</sub>, S<sub>8</sub>, S<sub>10</sub>, S<sub>11</sub> e S<sub>12</sub>). Seus valores de inconsistência variaram entre -2,9 no cenário S<sub>10</sub> (*Role*, *ZS-CoT*, *SV* e *RAG*) e -7,45 no cenário S<sub>3</sub> (*Role* e *ZS-CoT*), demonstrando boa capacidade de gerar código alinhado com explicações claras e coerentes na maioria das tarefas.

O modelo Llama3.1-405b obteve os melhores resultados em 4 cenários (S<sub>3</sub>, S<sub>5</sub>, S<sub>7</sub> e

S<sub>9</sub>), com destaque para o cenário S<sub>7</sub> (*Role*, RaR, SoT e RAG), onde alcançou o valor de -4,45, evidenciando sua robustez em cenários específicos que se mostraram mais desafiadores para outros modelos.

O Qwen2.5-72B destacou-se nos cenários S<sub>0</sub> (*Role*) e S<sub>1</sub> (*Role* e RAG), com valores de -5,7 e -4,3, respectivamente, apresentando desempenho competitivo nesses contextos iniciais. Já o Gemma3-27b obteve o melhor resultado apenas no cenário S<sub>4</sub> (*Role*, ZS-CoT e RAG), com o valor de -3,95, o melhor desempenho individual registrado em toda a avaliação.

Os modelos GPT-4o-mini e Mixtral-8x22b-instruct não obtiveram o melhor desempenho em nenhum cenário. O Mixtral-8x22b-instruct, em particular, apresentou consistentemente os piores resultados entre todos os modelos avaliados, com valores sistematicamente mais distantes de zero, como -9,55 no cenário S<sub>7</sub> e -8,25 no cenário S<sub>9</sub> (*Role*, ZS-CoT, SoT e RAG), indicando maior dificuldade em alinhar código e explicações de forma coerente.

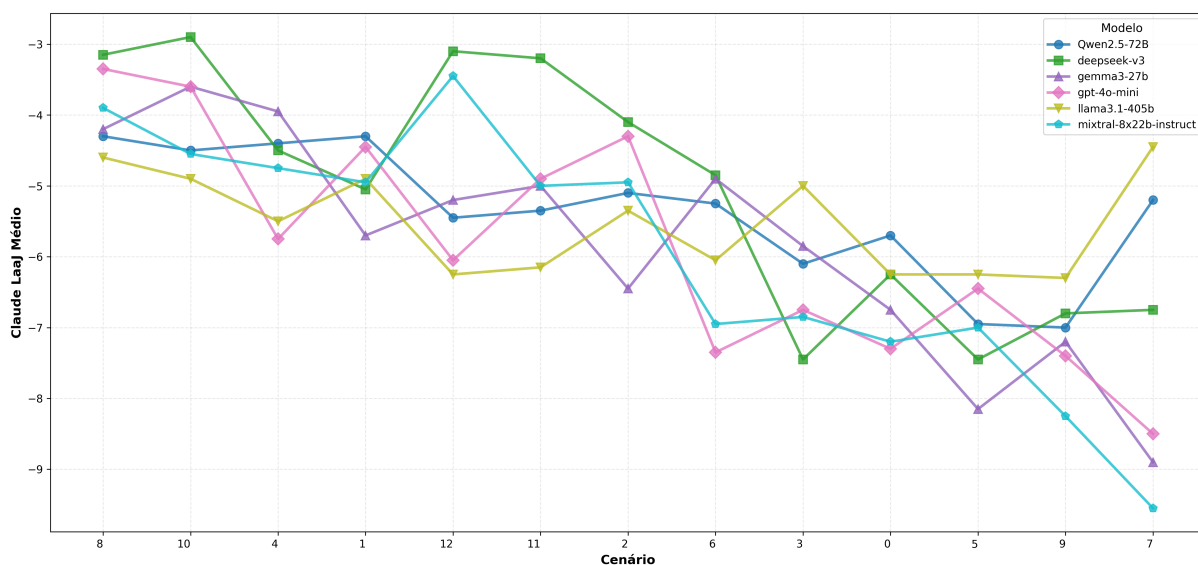
Fazendo uma observação por cenário, S<sub>10</sub> (*Role*, ZS-CoT, SV e RAG) apresentou os melhores resultados gerais, com todos os modelos alcançando valores entre -2,9 no DeepSeek-v3 e -4,9 no Llama3.1-405b, sugerindo que este cenário facilita a geração de código com explicações mais consistentes. O cenário S<sub>4</sub> (*Role*, ZS-CoT e RAG) também se destacou positivamente, com valores variando entre -3,95 no Gemma3-27b e -5,75 no GPT-4o-mini.

Em contrapartida, os cenários S<sub>5</sub> e S<sub>9</sub> registraram os piores desempenhos médios, com valores frequentemente superiores a -6,0, indicando maior complexidade ou ambiguidade que dificulta o alinhamento entre código e explicação. Contudo, o cenário S<sub>7</sub>, em particular, foi o mais desafiador para o Mixtral-8x22b-instruct (-9,55).

A variabilidade de desempenho entre cenários evidencia que a capacidade dos LLMs de gerar código consistente com suas explicações está fortemente atrelada às características específicas de cada tarefa.

Partindo para uma análise da evolução do desempenho dos modelos em termos de geração e explicação de código ao longo dos cenários, a Figura 19 apresenta os resultados dos seis modelos avaliados nos 13 cenários, ordenados do cenário com menor penalização média (valor de “final\_answer” mais próximo de **zero**, S<sub>8</sub>) ao cenário com maior acúmulo de penalizações (S<sub>7</sub>), segundo o julgamento do modelo Claude. Essa organização permite observar, de forma comparativa, como as diferentes combinações de técnicas de PE impactam o grau de inconsistências identificadas pelo julgador automático, evidenciando padrões de variação no desempenho dos modelos ao longo dos cenários analisados.

Figura 19 – Claude Laaj - Avaliação por Cenários x Modelos



Elaborada pelo Autor

Observa-se que, embora ambos os cenários compartilhem a mesma base estrutural — *Role*, *RaR* e *RAG* — a substituição da técnica *SV* (em  $S_8$ ) por *SoT* (em  $S_7$ ) produz efeitos substancialmente distintos no julgamento do modelo Claude. Em  $S_8$ , os valores de inconsistência permanecem relativamente próximos entre os modelos (aproximadamente entre -3,15 e -4,60), indicando maior uniformidade no desempenho. Já em  $S_7$ , verifica-se maior flutuação dos resultados, com variação de -4,45 no Llama3.1-405b a -9,55 no Mixtral-8x22b-instruct, evidenciando ampliação das diferenças entre as arquiteturas. A substituição de *SV* por *SoT* evidencia que nem todas as combinações de técnicas produzem resultados equivalentes e que a interação entre *RaR*, *SoT* e *RAG* pode criar conflitos ou sobrecarga de tokens que amplificam as diferenças de capacidade entre os modelos.

O DeepSeek-v3 demonstra melhor aderência nos cenários intermediários, particularmente em  $S_8$  (-3,15),  $S_{10}$  (*Role*, *ZS-CoT*, *SV* e *RAG*: -2,9),  $S_{11}$  (*Role*, *RaR*, *SoT*, *SV* e *RAG*: -3,2) e  $S_{12}$  (*Role*, *ZS-CoT*, *SoT*, *SV* e *RAG*: -3,1). Essa consistência sugere que o modelo responde bem a combinações que incluem *SV*, possivelmente porque essa técnica de autoverificação complementa suas capacidades internas. No entanto, observa-se declínio acentuado nos cenários  $S_0$  (-6,25),  $S_5$  (*Role*, *SoT* e *RAG*: -7,45) e  $S_9$  (*Role*, *ZS-CoT*, *SoT* e *RAG*: -6,8), indicando sensibilidade específica a configurações que utilizam *SoT* sem *SV* ou que carecem de técnicas adicionais de estruturação.

O Llama3.1-405b apresenta comportamento relativamente mais estável em comparação a alguns modelos, porém com um padrão consistente de sensibilidade às técnicas empregadas. Diferentemente dos demais, o cenário  $S_8$  (-4,6) figura entre os resultados menos favoráveis para esse modelo, comportamento que se repete em  $S_{10}$ ,  $S_{11}$  e  $S_{12}$  — todos envolvendo *SV* — indicando que a etapa de autoverificação não contribuiu para melhorar a qualidade conjunta da

geração e explicação nesse caso. Em contraste, seus melhores resultados concentram-se em  $S_3$  (-5,0),  $S_5$  (-6,25),  $S_7$  (-4,45) e  $S_9$  (-6,3), cenários associados a estratégias de raciocínio estruturado, o que sugere maior adequação do modelo a abordagens que organizam explicitamente o processo de resolução. Destaca-se especialmente  $S_7$  (-4,45), no qual o Llama mantém desempenho relativamente superior enquanto outros modelos apresentam declínio mais acentuado, indicando maior capacidade de lidar com a interação entre técnicas de reformulação e estruturação do pensamento quando essas orientam a construção da resposta desde o início, em vez de depender de mecanismos posteriores de autoverificação (SV).

O Qwen2.5-72B inicia com desempenho competitivo no cenário  $S_8$  (-4,3) e destaca-se particularmente nos cenários  $S_0$  (-5,7) e  $S_1$  (-4,3), demonstrando eficácia quando a combinação de técnicas é mais simples ou envolve apenas a adição de RAG ao *Role Prompting*. Contudo, apresenta oscilações em cenários intermediários, com declínios pontuais nos cenários  $S_{12}$  (-5,45),  $S_3$  (-6,1), e  $S_9$  (-7,0). Essa variabilidade sugere que o modelo é sensível à técnica ZS-CoT na forma como foi combinada.

O Gemma3-27b demonstra comportamento altamente volátil e imprevisível. Destaca-se o cenário  $S_4$  (*Role*, ZS-CoT e RAG: -3,95), que alcança o melhor resultado individual de toda a avaliação, o que indica forte sinergia nesta combinação para este modelo. No entanto, apresenta quedas abruptas em  $S_5$  (-8,15) e  $S_7$  (-8,9), sugerindo que a inclusão de SoT perturba significativamente seu desempenho, especialmente quando combinada com outras técnicas. Essa inconsistência indica possíveis limitações na capacidade de generalização ou na integração de múltiplas instruções estruturais simultâneas.

O GPT-4o-mini mantém desempenho intermediário e relativamente estável ao longo dos cenários, variando entre -3,35 e -8,5. Embora não se destaque positivamente em nenhum cenário específico, também não apresenta colapsos severos como outros modelos, sugerindo uma abordagem equilibrada no processamento de diferentes combinações de técnicas. Seu desempenho consistente pode indicar robustez generalista, embora sem especialização que permita vantagens significativas em configurações específicas.

O Mixtral-8x22b-instruct apresenta o pior desempenho geral, com declínio progressivo e sistemático conforme aumenta a complexidade das combinações. Inicia com valores moderados em  $S_8$  (-3,9) e  $S_{10}$  (-4,55), mas deteriora drasticamente em cenários como  $S_9$  (-8,25) e especialmente  $S_7$  (-9,55), o pior resultado individual registrado. A análise dos cenários revela que o modelo enfrenta dificuldades particulares quando SoT está presente, especialmente em  $S_7$  e  $S_9$ . Isso sugere limitações arquiteturais ou de treinamento que impedem o processamento eficaz de instruções que exigem decomposição estruturada do raciocínio combinada com outras técnicas.

A análise geral revela que a capacidade dos LLMs de manter coerência entre código e explicação, na utilização da avaliação do LaaJ, não depende linearmente do número de técnicas combinadas, mas sim de quais técnicas são utilizadas e como interagem entre si. A SV emerge como técnica estabilizadora essencial, enquanto SoT apresenta-se como fator de complexidade

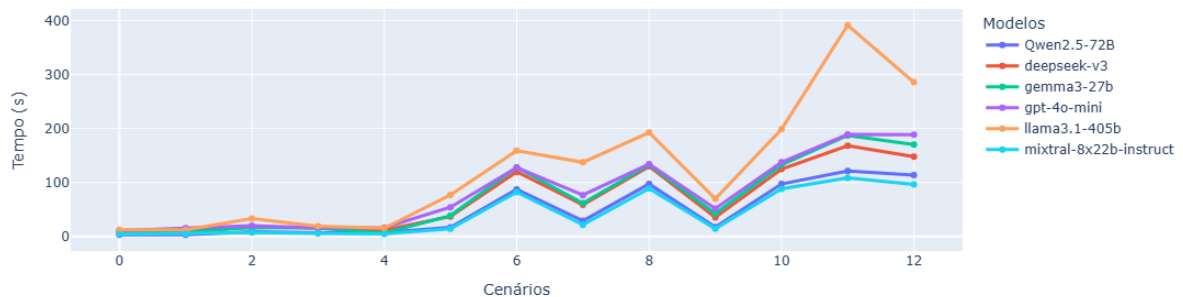
que exige capacidades específicas para ser processado adequadamente.

Estes resultados têm implicações práticas importantes: para maximizar desempenho, deve-se priorizar combinações incluindo SV; evitar SoT isolado sem mecanismos de validação; e considerar as capacidades específicas de cada modelo ao selecionar estratégias de *prompting*. A ordenação dos cenários valida que a engenharia de *prompts* eficaz requer compreensão não apenas de técnicas individuais, mas de suas interações e compatibilidades com as arquiteturas dos modelos específicos utilizados.

Contudo, essas recomendações precisam ser balanceadas com outra dimensão crítica: o custo computacional associado a cada estratégia. Afinal, técnicas que maximizam a qualidade das respostas podem incorrer em tempos de execução significativamente superiores, impactando a viabilidade prática de sua aplicação em ambientes de produção ou com restrições temporais. Nesse sentido, torna-se fundamental examinar como as diferentes combinações de técnicas de PE afetam a eficiência computacional dos modelos avaliados.

A Figura 20 apresenta a evolução dos tempos de execução ao longo dos 13 cenários para os seis modelos avaliados. O gráfico de linhas evidencia padrões distintos de comportamento entre os modelos, permitindo observar diferenças de escalabilidade, sensibilidade às técnicas empregadas e variações no custo computacional ao longo dos cenários experimentais.

Figura 20 – Claude LaaJ - Tempo de Execução em (s) - Cenários x Modelos



Elaborada pelo Autor

Os cenários  $S_0$  a  $S_4$ , que utilizam combinações mais simples de técnicas, apresentam os menores tempos de execução. Os cenários  $S_0$  e  $S_1$  demonstram desempenho particularmente eficiente entre os modelos avaliados, indicando que a adição do RAG não implica, necessariamente, aumento significativo no tempo de processamento. Da mesma forma, os cenários  $S_3$  e  $S_4$  mantêm-se em faixas temporais moderadas, sugerindo que a técnica de raciocínio em cadeia (ZS-CoT), quando aplicada isoladamente ou combinada apenas com RAG, não impõe sobrecarga computacional expressiva.

O gráfico revela dois pontos de inflexão significativos comuns a todos os modelos. O primeiro ocorre no cenário  $S_5$ , com a introdução da técnica SoT, resultando em elevação

moderada, porém consistente, dos tempos de execução. O segundo, mais pronunciado, ocorre no cenário  $S_6$ , marcando a entrada da técnica SV. A partir desse ponto, observa-se crescimento substancial dos tempos, especialmente nos cenários  $S_8$ ,  $S_{10}$ ,  $S_{11}$  e  $S_{12}$ , que combinam SV com outras técnicas iterativas. Esses resultados indicam que a técnica SV constitui o principal fator de impacto no custo temporal, devido aos ciclos adicionais de geração, verificação e possível refinamento que impõe ao processo.

O Qwen2.5-72B e o Mixtral-8x22b-instruct destacam-se por manterem tempos de execução consistentemente baixos ao longo de todos os cenários, com crescimento gradual e controlado. Mesmo nos cenários mais complexos ( $S_{11}$  e  $S_{12}$ ), estes modelos permanecem abaixo de 125s, demonstrando capacidade superior de processar técnicas iterativas sem declínio acentuado de desempenho. O Mixtral-8x22b-instruct, em particular, mantém-se como o mais eficiente em praticamente todos os pontos, validando sua adequação para aplicações que demandam respostas rápidas mesmo com *prompts* elaborados.

O Deepseek-v3, GPT-4o-mini e Gemma3-27b apresentam comportamento intermediário, com crescimento mais acentuado a partir do cenário  $S_5$ , quando técnicas iterativas como SoT são introduzidas. Estes modelos oscilam entre aproximadamente 15s nos cenários simples e 190s nos mais complexos, representando um compromisso razoável entre capacidade de processamento e custo temporal. A similaridade entre as trajetórias destes três modelos sugere comportamento computacional comparável frente às mesmas estratégias de *prompting*.

Em contraste, o Llama3.1-405b apresenta trajetória temporal marcadamente superior aos demais modelos, sobretudo nos cenários de maior complexidade. Sua curva ascendente culmina em um “pico” de 391s no cenário  $S_{11}$ , que combinam múltiplas técnicas iterativas, evidenciando custo computacional significativamente mais elevado. Este comportamento indica que, embora o modelo possa oferecer capacidades avançadas devido ao seu tamanho (405 bilhões de parâmetros), sua aplicação em cenários com múltiplas técnicas iterativas requer consideração cuidadosa das restrições temporais do contexto de uso.

Observa-se ainda que cenários que combinam técnicas iterativas — especialmente SV associada a SoT — produzem aumento substancial de custo computacional sem ganhos qualitativos proporcionais. Em contrapartida, técnicas como RAG, Role e ZS-CoT apresentam impacto relativamente reduzido na latência quando utilizadas isoladamente ou em combinações menos densas.

A análise conjunta entre tempo de execução e qualidade da geração revela que desempenho qualitativo e custo computacional não evoluem de forma diretamente proporcional. Modelos que alcançam boa coerência entre código e explicação nem sempre são os mais eficientes temporalmente. O Mixtral-8x22b-instruct exemplifica equilíbrio favorável entre qualidade e eficiência, enquanto o Llama3.1-405b evidencia alto custo temporal sem superioridade consistente no julgamento qualitativo. O DeepSeek-v3 posiciona-se como alternativa intermediária, combinando desempenho robusto com custo moderado.

Do ponto de vista prático, esses achados permitem identificar cenários mais adequados a diferentes níveis de exigência. O cenário  $S_4$  destaca-se pelo equilíbrio entre custo computacional e qualidade, sendo apropriado para aplicações em tempo real com restrições moderadas de latência. O cenário  $S_{10}$ , embora mais caro computacionalmente, oferece ganhos qualitativos que podem ser justificáveis em tarefas críticas. Por outro lado, cenários que combinam SV e SoT simultaneamente mostraram pior relação custo-benefício, sugerindo que devem ser evitados quando há limitação de recursos ou necessidade de desempenho estável.

De maneira geral, os resultados do Experimento 3.1 demonstram que a coerência entre código e explicação — avaliada pelo paradigma LaaJ sob o julgamento do modelo Claude — depende não apenas das capacidades intrínsecas de cada LLM, mas também da interação específica entre as técnicas de engenharia de *prompts* empregadas. A robustez observada não é uniforme entre os modelos, e os efeitos das combinações variam conforme a arquitetura e a forma como cada técnica é processada internamente.

Uma engenharia de *prompts* eficaz requer mais do que o conhecimento isolado das técnicas disponíveis: exige compreensão das interações entre elas e de como cada modelo responde a tais combinações. Assim, a escolha do cenário ideal deve considerar primeiro as restrições temporais da aplicação e, em seguida, a qualidade mínima desejada, equilibrando desempenho e latência por meio da seleção adequada de modelos — privilegiando arquiteturas mais eficientes em situações de baixa tolerância à espera e modelos mais robustos quando a qualidade final é o fator determinante.

Para ilustrar concretamente o comportamento observado no cenário  $S_{10}$  do modelo DeepSeek-v3, apresenta-se o Código 2, contendo o *ground-truth* e a saída gerada pelo modelo no Experimento 3.1. A comparação entre ambos evidencia elevada equivalência funcional: o código gerado mantém a lógica central do problema — leitura da matriz, classificação dos valores segundo os intervalos definidos, substituição pelos símbolos correspondentes e contabilização das ocorrências — preservando integralmente os requisitos estabelecidos no enunciado.

Embora haja reorganização estrutural, como o uso de *list comprehension* para leitura da matriz e a construção de uma nova matriz auxiliar (símbolos), a transformação não altera o comportamento esperado. A lógica condicional permanece semanticamente equivalente ao *ground-truth*, com correta distinção entre os três intervalos de valores ( $\leq 90$ , entre 91 e 100, e  $> 100$ ) e respectiva atualização dos contadores. Essa equivalência justifica os baixos níveis de inconsistência atribuídos pelo julgador automático no cenário  $S_{10}$ .

Além da geração do código, observa-se que a explicação associada apresenta descrição linha a linha, explicitando o papel de cada variável, das estruturas de repetição e das condições aplicadas. A correspondência entre código e explicação demonstra alinhamento conceitual adequado, sem contradições semânticas relevantes. Esse aspecto é fundamental no contexto do Experimento 3, pois o julgamento LaaJ considera não apenas a correção da implementação, mas também a coerência e a completude da explicação.

As inconsistências identificadas pelo modelo julgador — detalhadas no Apêndice F — são pontuais e não comprometem a lógica central da solução. Assim, o exemplo evidencia que, no cenário S10, a combinação de técnicas de PE foi capaz de atender simultaneamente às duas dimensões avaliadas: geração funcionalmente correta e explicação coerente com o comportamento do código. O caso ilustra, portanto, um cenário em que a engenharia de prompts contribui para reduzir desalinhamentos entre implementação e justificativa textual, atendendo plenamente aos objetivos do experimento.

### 5.3.2 Experimento 3.2

Antes de prosseguir, realizou-se uma releitura completa das instâncias de código e explicação persistidas após o Experimento 3.1, contendo as saídas geradas pelos seis LLMs. Essa verificação teve como objetivo assegurar a integridade, a consistência e a adequada estrutura dos registros antes de sua reutilização no Experimento 3.2.

A estratégia adotada consistiu no reaproveitamento integral dessas instâncias — isto é, os mesmos códigos e explicações produzidos no Experimento 3.1 foram utilizados como entrada para o novo julgamento. Com isso, evitou-se a necessidade de regeneração das respostas, eliminando possíveis variações estocásticas inerentes a novas execuções dos LLMs e preservando a comparabilidade entre os avaliadores.

Dessa forma, o Claude LaaJ foi substituído pelo Gemini, denominado *Gemini LaaJ* como modelo julgador, mantendo-se inalterados todos os demais elementos do experimento — cenários experimentais, modelos geradores e respectivas saídas. Esse procedimento assegura que quaisquer diferenças observadas entre os resultados do Experimento 3.1 e do Experimento 3.2 possam ser atribuídas exclusivamente à variação do modelo avaliador, permitindo analisar, de maneira controlada, a consistência e a estabilidade do julgamento automático quando se altera o LLM responsável pela avaliação.

A Figura 21 sintetiza o desempenho dos cenários experimentais quanto à geração e à explicação de código sob a perspectiva do modelo julgador Gemini. A avaliação realizada pelo Gemini LaaJ consistiu na identificação e quantificação das inconsistências previstas no catálogo descrito na Subseção 4.2.3.

Figura 21 – Gemini LaaJ - Média de Valores por Cenários x Modelos

Cenário	Qwen2.5-72B	deepseek-v3	gemma3-27b	gpt-4o-mini	llama3.1-405b	mixtral-8x22b-instruct
0	-14.462	-15.583	-22.583	-16.733	-18.333	-14.706
1	-20.714	-19.455	-19.923	-16.0	-15.824	-21.636
2	-14.923	-15.385	-22.714	-16.133	-19.5	-17.0
3	-17.812	-16.917	-21.231	-19.0	-19.667	-18.118
4	-18.467	-13.812	-22.0	-13.643	-21.3	-17.0
5	-18.385	-16.308	-22.294	-16.818	-17.909	-18.0
6	-14.3	-15.444	-14.45	-14.25	-16.421	-16.2
7	-17.167	-19.286	-17.933	-20.286	-22.273	-20.467
8	-19.111	-7.842	-10.421	-10.944	-13.526	-14.895
9	-22.267	-20.667	-18.214	-23.455	-18.545	-22.812
10	-14.15	-12.211	-13.85	-15.368	-10.833	-13.95
11	-12.765	-12.789	-13.737	-17.25	-15.211	-16.556
12	-15.15	-8.444	-12.944	-13.947	-14.35	-12.842

Legenda: verde = melhor score por cenário; vermelho = pior score por cenário.

Elaborada pelo Autor

A análise por cenários revela hierarquias de desempenho que divergem parcialmente das observadas no Experimento 3.1. O cenário  $S_8$  (*Role*, *RaR*, *SV* e *RAG*) apresenta as melhores avaliações, com destaque para o DeepSeek-v3 alcançando -7,842, correspondendo ao melhor resultado individual da tabela. O cenário  $S_{12}$  (*Role*, *ZS-CoT*, *SoT*, *SV* e *RAG*) também se destaca positivamente, especialmente com o DeepSeek-v3 obtendo -8,444, indicando desempenho superior consistente deste modelo em cenários de alta complexidade com *SV*.

O cenário  $S_{10}$  (*Role*, *ZS-CoT*, *SV* e *RAG*), também apresenta desempenho relevante, especialmente para o Llama3.1-405b, que alcança -10,833. O cenário  $S_6$  (*Role*, *SV* e *RAG*) também apresenta resultados positivos, com o GPT-4o-mini alcançando -14,25, sugerindo que o Gemini tende a atribuir avaliações mais favoráveis a configurações que incluem mecanismos explícitos de verificação.

Em contrapartida, cenários que combinam múltiplas técnicas sem *SV* ou com configurações inadequadas apresentam as avaliações mais desfavoráveis. O cenário  $S_9$  (*Role*, *ZS-CoT*, *SoT* e *RAG*) registra os piores scores, com o GPT-4o-mini atingindo -23,455, representando o valor mais baixo de toda a tabela. O cenário  $S_7$  (*Role*, *RaR*, *SoT* e *RAG*) também apresenta baixo score, com o Llama3.1-405b em -22,273, sugerindo que a combinação de *SoT* sem mecanismos

adicionais de verificação pode comprometer a avaliação da geração e explicação de código sob o julgamento do Gemini.

Os cenários *baseline* e de baixa complexidade ( $S_1$  a  $S_4$ ) apresentam desempenho intermediário, com scores geralmente entre -13 e -22, sugerindo que a ausência de técnicas de verificação limita o nível de desempenho alcançável, mesmo em configurações computacionalmente eficientes.

O DeepSeek-v3 demonstra desempenho particularmente robusto, obtendo os melhores scores em múltiplos cenários, com destaque para os cenários  $S_8$  (-7,842),  $S_{12}$  (-8,444),  $S_3$  (-16,917) e  $S_5$  (-16,308). O Qwen2.5-72B também apresenta desempenho sólido, especialmente nos cenários  $S_{11}$  (-12,765),  $S_6$  (-14,3),  $S_0$  (-14,462) e  $S_2$  (-14,923).

O Gemma3-27b apresenta avaliações consistentemente menos favoráveis, com resultados frequentemente entre os mais baixos em diversos cenários, incluindo -22,583 no cenário  $S_0$ , -22,714 no cenário  $S_2$ , -22,294 no cenário  $S_5$  e -22,0 no cenário  $S_4$ , indicando limitações recorrentes sob os critérios adotados pelo avaliador Gemini LaaJ. Entretanto, no cenário  $S_9$  (-18,214), apresenta o melhor score em relação aos demais modelos.

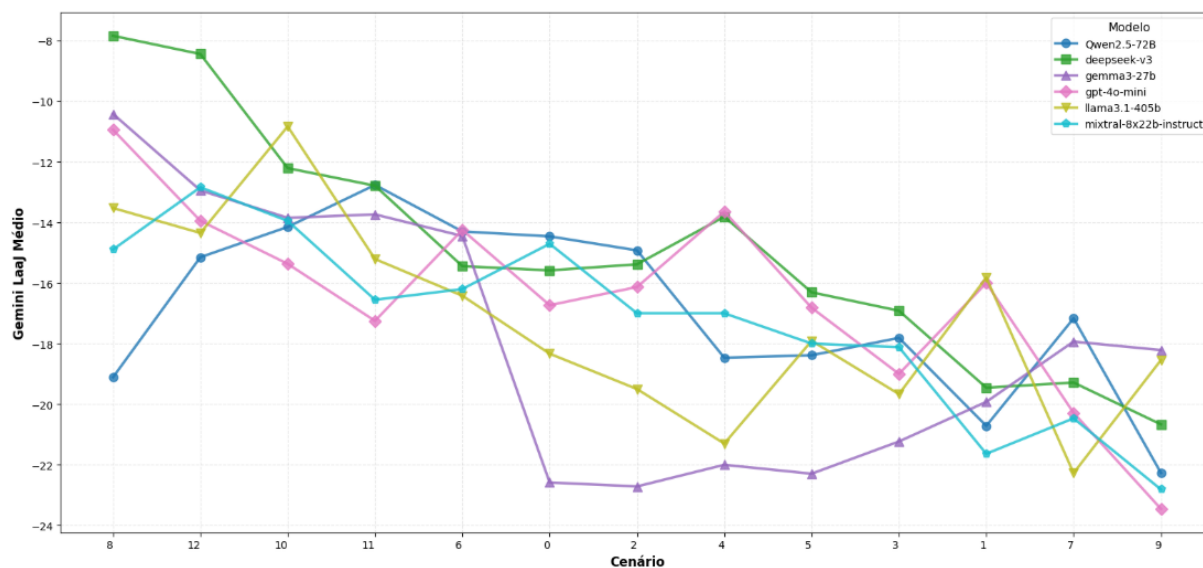
O Llama3.1-405b apresenta comportamento interessante, especialmente no cenário  $S_{10}$  com -10,833 e não complexos no cenário  $S_1$  com -15,824. Esse padrão sugere que sua arquitetura de grande escala pode favorecer determinadas combinações que incluem mecanismos estruturados de raciocínio e verificação, refletindo-se em resultados mais favoráveis em cenários específicos.

Observa-se que a presença de mecanismos de verificação tende a estar associada a avaliações mais favoráveis, enquanto combinações que ampliam a complexidade sem esse suporte resultam em maiores níveis de inconsistência.

A Figura 22 apresenta a evolução dos scores atribuídos pelo Gemini LaaJ ao longo dos 13 cenários para os seis modelos avaliados, permitindo visualizar de forma integrada os padrões de desempenho identificados na análise tabular anterior. O gráfico de linhas, com cenários ordenados de forma não sequencial conforme seus scores médios, evidencia comportamentos distintos tanto entre modelos quanto entre os cenários experimentais.

Uma característica marcante do gráfico é que praticamente todos os modelos apresentam trajetórias descendentes ao longo dos cenários, partindo de scores mais favoráveis à esquerda (cenários  $S_8$ ,  $S_{12}$  e  $S_{10}$ ) e convergindo para scores progressivamente mais desfavoráveis à direita (cenários  $S_7$  e  $S_9$ ). Esta tendência geral sugere que, sob a perspectiva do Gemini LaaJ, existe uma hierarquia clara de qualidade entre as diferentes combinações de técnicas de PE, com os cenários que incluem SV posicionados predominantemente à esquerda e aqueles sem mecanismos robustos de validação à direita.

Figura 22 – Gemini Laaj - Avaliação por Cenários x Modelos



Elaborada pelo Autor

O DeepSeek-v3 apresenta declínio constante ao longo da maioria dos cenários, iniciando no ponto mais alto do gráfico (cenário S<sub>8</sub> com -7,842) e descendo progressivamente. Entretanto, destaca-se uma melhora pontual no cenário S<sub>4</sub>, onde a linha apresenta elevação temporária antes de retomar a trajetória descendente nos cenários subsequentes. Este comportamento indica que, embora o modelo seja altamente sensível à degradação de qualidade em configurações menos adequadas, consegue responder positivamente a combinações específicas mesmo em regiões intermediárias do espectro de complexidade.

O Qwen2.5-72B apresenta comportamento distintamente mais estável em comparação aos demais modelos. Sua trajetória, embora também descendente, caracteriza-se por oscilações mais suaves e menor amplitude de variação, mantendo-se em uma faixa relativamente constante entre -14 e -20 ao longo dos cenários. Esta consistência sugere que o Qwen2.5-72B é menos sensível às variações nas combinações de técnicas de PE, mantendo desempenho homogêneo independentemente da complexidade ou combinação específica de técnicas empregadas. Esta característica pode ser interpretada tanto como robustez quanto como menor capacidade de resposta diferenciada a estratégias otimizadas.

O Gemma3-27b apresenta padrão particularmente problemático: declínio acentuado que persiste até o cenário S<sub>0</sub> (*baseline*), onde atinge um dos pontos mais baixos de sua trajetória com -22,583. Surpreendentemente, o modelo demonstra melhora nos cenários subsequentes à direita do *baseline*, sugerindo que, para este modelo específico, determinadas combinações de técnicas mais simples podem produzir resultados superiores ao próprio *baseline*. Esta inversão de expectativas indica possíveis dificuldades do Gemma3-27b em processar adequadamente o *Role* isolado, ou que a adição de técnicas específicas compensa limitações arquiteturais fundamentais.

O GPT-4o-mini e o Llama3.1-405b exibem trajetórias marcadas por alta volatilidade,

com oscilações pronunciadas entre cenários adjacentes. Ambos alternam entre desempenho relativamente favorável em pontos específicos e quedas abruptas em outros, resultando em linhas com múltiplos “picos e vales”. O GPT-4o-mini, em particular, atinge seu pior desempenho no cenário S<sub>9</sub> (-23,455), enquanto o Llama3.1-405b, apesar de sua arquitetura robusta, apresenta inconsistência similar, com variações que sugerem dependência acentuada da compatibilidade entre suas capacidades arquiteturais e as técnicas de PE específicas empregadas.

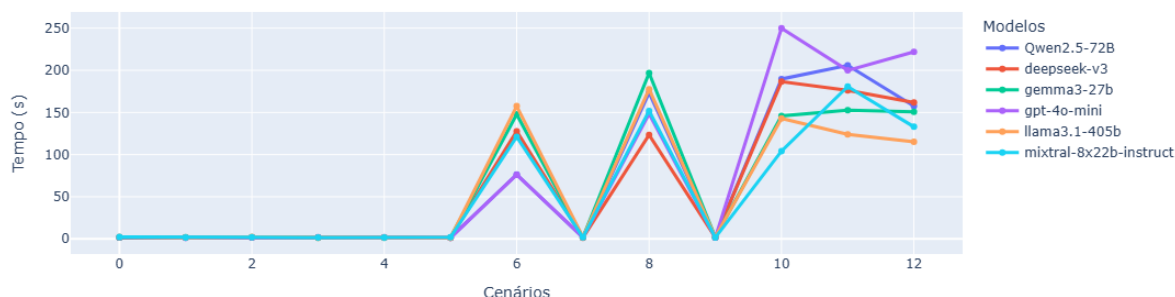
O Mixtral-8x22b-instruct apresenta declínio moderado e relativamente linear, sem os “picos” de recuperação observados em outros modelos. Sua trajetória descendente consistente, mantendo-se em faixas intermediárias do gráfico, sugere limitações estruturais que impedem tanto desempenhos superiores em relação aos demais modelos quanto recuperações significativas, independentemente das combinações de técnicas de PE utilizadas.

Um padrão visual importante é a convergência de todas as linhas nos cenários à extrema direita do gráfico (S<sub>7</sub> e S<sub>9</sub>), onde praticamente todos os modelos atingem scores substancialmente desfavoráveis. Esta convergência reforça que determinadas combinações de técnicas — especificamente aquelas que incluem SoT sem SV — produzem resultados problemáticos, superando as diferenças arquiteturais entre os modelos. A concentração visual das trajetórias nesta região evidencia limitações fundamentais destas combinações de técnicas de PE.

A análise revela que, embora exista consenso geral sobre a superioridade dos cenários com SV e as limitações daqueles com SoT isolado, os modelos respondem de forma heterogênea a estas configurações. A tendência geral de declínio observada em todos os modelos valida a ordenação dos cenários proposta, mas as variações individuais de trajetória indicam que a escolha do modelo gerador permanece relevante mesmo quando as técnicas de PE são mantidas constantes. Esta constatação reforça a necessidade de considerar tanto a estratégia de *prompting* quanto as características arquiteturais do modelo ao projetar sistemas baseados em LLMs.

A Figura 23 apresenta a evolução dos tempos de execução do Gemini LaaJ ao longo dos 13 cenários para os seis modelos avaliados, permitindo identificar padrões claros de escalonamento não linear, com saltos abruptos de tempo em cenários específicos e aumento da variabilidade entre modelos à medida que se introduzem combinações técnicas mais complexas.

Figura 23 – Gemini LaaJ - Tempo de Execução em (s) - Cenários x Modelos



Elaborada pelo Autor

O gráfico apresenta uma estrutura visual marcante: uma região de estabilidade próxima a zero nos cenários  $S_0$  a  $S_5$ , seguida por elevações abruptas nos cenários  $S_6$ ,  $S_8$  e  $S_{10}$ , com quedas intermediárias em  $S_7$  e  $S_9$ . Essa conformação em “picos e vales” revela padrão claramente segmentado de comportamento temporal.

Nos cenários  $S_0$  a  $S_5$ , as linhas praticamente se sobrepõem, formando uma base única próxima ao eixo horizontal. Essa convergência indica que, na ausência da técnica SV, o Gemini LaaJ apresenta eficiência temporal homogênea e variações mínimas entre modelos, independentemente da combinação de técnicas empregadas.

O primeiro pico ocorre no cenário  $S_6$  (*Role*, SV e RAG), quando a introdução da SV provoca salto abrupto nos tempos de execução. O segundo pico, mais pronunciado, manifesta-se no cenário  $S_8$  (*Role*, RaR, SV e RAG), e o terceiro no cenário  $S_{10}$  (*Role*, ZS-CoT, SV e RAG), onde se observa o maior tempo absoluto do gráfico. Em contraste, nos cenários  $S_7$  (*Role*, RaR, SoT e RAG) e  $S_9$  (*Role*, ZS-CoT, SoT e RAG) — que incluem SoT, mas não SV — os tempos de execução retornam a níveis semelhantes aos observados nos cenários iniciais. Essa alternância caracteriza um padrão em “dente de serra”, indicando que a presença ou ausência da SV atua como elemento determinante no custo computacional.

A partir dessa observação, evidencia-se que o Gemini LaaJ apresenta comportamento temporal estruturalmente bimodal: altamente eficiente quando avalia configurações sem SV e significativamente mais custoso quando a SV é introduzida. Técnicas como SoT, RaR e ZS-CoT, embora adicionem complexidade estrutural ao *prompt*, não impõem elevação temporal relevante quando desacompanhadas de mecanismos iterativos de verificação. Já a SV, ao envolver ciclos adicionais de validação e refinamento, constitui o principal fator de aumento no tempo de execução.

Nos cenários  $S_{11}$  e  $S_{12}$ , observa-se manutenção de patamar elevado de tempo, configurando uma espécie de platô. Nesses casos, a presença simultânea de SV e múltiplas técnicas adicionais mantém o processamento em níveis críticos, reforçando que a combinação de verifica-

ção iterativa com outras estratégias estruturadas sustenta a carga computacional elevada.

Nos picos, também se verifica maior dispersão vertical entre modelos, indicando que, em cenários de maior complexidade avaliativa, a eficiência passa a depender mais fortemente das características específicas do modelo analisado. Em contrapartida, nos cenários simples, as diferenças arquiteturais praticamente desaparecem sob o ponto de vista temporal.

A análise qualitativa revela hierarquia consistente entre cenários.  $S_8$  e  $S_{12}$  emergem como configurações de melhor avaliação, com destaque para o DeepSeek-v3.  $S_6$  e  $S_{10}$  também apresentam avaliações favoráveis, com desempenho relevante do Llama3.1-405b no  $S_{10}$ . Observa-se convergência em torno de cenários que incorporam SV como elemento central, indicando que essa técnica desempenha papel decisivo na redução de inconsistências entre código e explicação. Em contraste, configurações que combinam SoT sem SV — particularmente  $S_7$  e  $S_9$  — apresentaram os piores resultados. Essa recorrência sugere que a decomposição estruturada do raciocínio, quando não acompanhada de mecanismos de verificação, pode introduzir complexidade que compromete a coerência final das respostas. Esta convergência indica que as limitações destas configurações transcendem as diferenças arquiteturais entre modelos, representando incompatibilidades fundamentais entre as técnicas combinadas.

A integração das dimensões qualitativa e temporal evidencia tensão clara: os cenários de melhor avaliação são precisamente aqueles que impõem maiores custos computacionais. Configurações como  $S_8$ ,  $S_{10}$  e  $S_{12}$  apresentam desempenho avaliativo superior, mas demandam latências significativamente mais elevadas. Por outro lado, cenários simples oferecem extrema eficiência temporal, porém limitam o nível de consistência entre código e explicação alcançável. A ausência de mecanismos de verificação impõe teto qualitativo que não é superado apenas pela escolha do modelo gerador.

Apesar das variações entre cenários experimentais, alguns padrões se repetem. O DeepSeek-v3 apresenta estabilidade consistente em múltiplos cenários de alto desempenho qualitativo. O Gemma3-27b evidencia dificuldades recorrentes, especialmente em configurações mais exigentes. O GPT-4o-mini demonstra maior volatilidade temporal nos picos, enquanto o Llama3.1-405b apresenta desempenho relevante em configurações específicas, como  $S_{10}$ .

O Experimento 3.2 evidencia que o Gemini LaaJ possui perfil avaliativo distintivo: elevada eficiência em cenários simples, sensibilidade pronunciada à introdução da SV, comportamento temporal segmentado e forte correlação entre verificação iterativa e melhoria qualitativa.

Os resultados indicam que a eficácia da engenharia de *prompts* para avaliação automática depende não apenas das técnicas utilizadas, mas também das características específicas do avaliador. Não há avaliador universalmente superior; há perfis distintos de *trade-offs* entre consistência avaliativa e custo computacional. Assim, a escolha das combinações de técnicas de PE deve considerar os requisitos operacionais da aplicação, equilibrando latência aceitável e nível desejado de consistência entre código e explicação, sempre reconhecendo que o modelo

avaliador influencia diretamente os resultados observados.

Para ilustrar os resultados obtidos no Experimento 3.2, apresenta-se no Apêndice G, um código extraído do cenário  $S_8$ , gerado pelo modelo DeepSeek-v3, no qual se observa de forma concreta, a relação entre o código gerado, sua explicação e o julgamento realizado pelo Gemini LaaJ, bem como as inconsistências identificadas e o respectivo valor de “*final\_answer*”.

Ao comparar o *ground-truth* com o código gerado, observa-se elevada correspondência estrutural e funcional. A lógica de classificação das células e os critérios condicionais mantêm equivalência semântica com a solução de referência, ainda que organizados de forma distinta.

Entretanto, a análise dos comentários inseridos no código evidencia uma inconsistência explicativa relevante. A solução gerada introduz o elemento narrativo “*K-Oru*” como justificativa para o acréscimo de 10 unidades, ainda que tal referência não esteja explicitamente modelada no *ground-truth*. Embora essa contextualização não comprometa a execução do algoritmo, ela adiciona uma camada interpretativa que não é formalmente necessária para a resolução do problema.

Desse modo, verifica-se alinhamento funcional entre código e referência, mas não equivalência plena entre implementação e explicação. A presença de justificativas contextuais não previstas no enunciado original sugere leve desalinhamento semântico, aspecto refletido no julgamento automático do Gemini LaaJ.

### 5.3.3 Comparativo entre os Avaliadores LaaJ

Com os resultados dos Experimentos 3.1 e 3.2 estabelecidos, torna-se possível realizar uma análise comparativa entre os dois modelos julgadores empregados no Experimento 3. Essa comparação evidencia não apenas diferenças quantitativas nos scores atribuídos e tempos de execução, mas também distinções no padrão de utilização dos critérios avaliativos, o que têm implicações significativas para a aplicação prática do paradigma LaaJ.

Uma semelhança fundamental refere-se ao protocolo de avaliação adotado por ambos os julgadores. Tanto o Claude LaaJ quanto o Gemini LaaJ utilizaram o mesmo catálogo de 22 critérios de inconsistência, estruturado a partir da escala de severidade discreta (-4 a 0) definida na Subseção 4.2.3. Em ambos os casos, os valores atribuídos correspondem ao somatório das severidades identificadas, de modo que resultados mais próximos de zero indicam menor incidência de inconsistências na geração e explicação de código.

Apesar da base metodológica comum, observa-se diferença na amplitude dos valores finais atribuídos. O Claude LaaJ operou com variações menos acentuadas entre os cenários, enquanto o Gemini LaaJ apresentou maior dispersão dos resultados, com valores variando aproximadamente entre -7,842 e -23,455. Essa diferença não decorre de alteração na escala ou no catálogo — que permaneceram idênticos —, mas do modo como cada avaliador mobilizou os critérios durante o julgamento.

A análise da quantidade média de inconsistências registradas por avaliação revela padrão distinto entre os julgadores. O Gemini LaaJ tende a registrar maior número de ocorrências por resposta avaliada, acionando com maior frequência diferentes critérios do catálogo. Esse comportamento resulta em avaliações mais extensivas e detalhadas, refletindo aplicação sistemática do conjunto completo de critérios estabelecidos no protocolo.

O Claude LaaJ, por sua vez, apresenta número médio inferior de inconsistências registradas por avaliação. Em vez de acionar amplamente os critérios disponíveis, suas avaliações demonstram concentração em aspectos considerados centrais para a coerência entre código e explicação. Esse padrão sugere aplicação mais seletiva dos critérios, com menor volume de ocorrências explicitamente registradas.

Do ponto de vista prático, essa diferença é relevante: enquanto o Gemini LaaJ tende a produzir pareceres mais abrangentes em termos de cobertura criterial, o Claude LaaJ apresenta julgamentos mais concisos. Assim, a escolha do avaliador pode influenciar o grau de detalhamento do julgamento automático, especialmente em contextos que demandam maior granularidade na identificação de inconsistências.

Ambos os avaliadores concordaram quanto à importância central da SV como técnica estabilizadora. Cenários incorporando SV ( $S_6$ ,  $S_8$ ,  $S_{10}$ ,  $S_{11}$ ,  $S_{12}$ ) receberam avaliações consistentemente superiores de ambos os juízes, estabelecendo consenso robusto sobre o valor desta técnica. Similarmente, ambos identificaram os cenários  $S_7$  e  $S_9$  — que combinam SoT sem mecanismos de validação — como configurações particularmente problemáticas, registrando scores desfavoráveis independentemente do avaliador.

Entretanto, divergências significativas emergiram nas ordenações relativas atribuídas às respostas avaliadas. No caso do Claude LaaJ, as melhores avaliações concentraram-se em respostas geradas sob combinações de complexidade moderada, com destaque para os cenários  $S_4$  (*Role*, ZS-CoT e RAG) e  $S_{10}$  (*Role*, ZS-CoT, SV e RAG), posicionados como configurações mais equilibradas entre geração e explicação. Já o Gemini LaaJ atribuiu os melhores scores a respostas produzidas em cenários de maior complexidade estrutural, como  $S_8$  (*Role*, RaR, SV e RAG) e  $S_{12}$  (*Role*, ZS-CoT, SoT, SV e RAG). Essas diferenças indicam que cada julgador estabelece hierarquias distintas ao avaliar os pares modelo–cenário, refletindo variações na forma como as inconsistências são identificadas e ponderadas nas saídas analisadas.

Esta divergência sugere diferenças nos critérios implícitos de valorização: o Claude parece incorporar considerações sobre clareza comunicativa e eficiência estrutural, penalizando parcialmente cenários excessivamente complexos mesmo quando tecnicamente corretos. O Gemini, ao aplicar sistematicamente todos os 22 critérios, foca mais intensamente na completude e correção técnica, valorizando cenários que atendem exaustivamente todos os aspectos avaliativos, independentemente da complexidade requerida.

Ambos os avaliadores concordaram quanto à robustez do DeepSeek-v3, que obteve

desempenho consistentemente favorável sob ambas as perspectivas. Este consenso estabelece o DeepSeek-v3 como modelo de capacidades superiores, validado por critérios avaliativos independentes. Similarmente, ambos identificaram limitações sistemáticas do Gemma3-27b, que apresentou scores desfavoráveis independentemente do juiz utilizado.

Contudo, divergências emergiram na avaliação de outros modelos. O Llama3.1-405b recebeu avaliações relativamente mais favoráveis do Gemini LaaJ em cenários complexos, enquanto o Claude havia apresentado julgamentos mais moderados. Esta diferença pode refletir a abordagem exaustiva do Gemini: modelos com arquiteturas robustas que atendem múltiplos critérios técnicos são valorizados mais intensamente, mesmo se apresentarem limitações em aspectos comunicativos ou de clareza.

O Qwen2.5-72B demonstrou desempenho distintamente mais estável sob o Gemini LaaJ, com menor variação entre cenários, enquanto havia apresentado oscilações mais pronunciadas sob o Claude LaaJ. Esta diferença sugere que a aplicação sistemática de todos os critérios pelo Gemini pode atenuar variações contextuais, produzindo avaliações mais uniformes que valorizam consistência técnica em detrimento de excelências pontuais.

O Claude LaaJ apresentou maior poder discriminativo entre modelos, produzindo variância intra-cenário mais pronunciada. Esta característica permitiu identificar diferenças sutis de desempenho, estabelecendo ordenações claras mesmo em cenários onde múltiplos modelos apresentaram capacidades comparáveis. A dispersão vertical das linhas no gráfico do Claude evidencia visualmente esta capacidade diferenciadora.

O Gemini LaaJ, ao aplicar sistematicamente todos os 22 critérios, produziu avaliações com menor variação entre os modelos dentro de cada cenário. Observa-se, em diversos casos, intervalo mais estreito entre os valores de “*final\_answer*” atribuídos aos diferentes LLMs sob a mesma configuração experimental. Essa menor dispersão intra-cenário sugere que o Gemini é menos sensível às diferenças arquiteturais específicas dos modelos ou que sua aplicação exaustiva dos critérios tende a uniformizar a avaliação. Essa homogeneização possui implicações ambivalentes: por um lado, favorece consistência e previsibilidade; por outro, pode reduzir a capacidade discriminativa para capturar nuances qualitativas entre modelos e cenários.

A convergência visual das linhas no gráfico do Gemini, especialmente nos cenários simples, ilustra esta característica. Enquanto o Claude mantinha diferenciação visível mesmo em configurações básicas, o Gemini produziu avaliações praticamente indistinguíveis graficamente. Para aplicações que requerem seleção fina entre modelos similares, o Claude pode oferecer vantagens; para contextos que priorizam estabilidade avaliativa, o Gemini apresenta benefícios.

Ambos os avaliadores confirmaram a SV como fator dominante de qualidade, mas diferiram na intensidade desta valorização. O Gemini atribuiu aos cenários com SV vantagens qualitativas mais pronunciadas, possivelmente porque a verificação iterativa permite atender mais completamente aos 22 critérios avaliativos. O Claude, embora também valorizando SV,

apresentou diferenciação menos acentuada, sugerindo que outros aspectos — como clareza comunicativa e eficiência estrutural — recebem peso comparável em seus julgamentos.

O SoT recebeu avaliação consistentemente negativa de ambos os juízes quando aplicado sem mecanismos de validação, estabelecendo consenso robusto sobre as limitações desta configuração. Entretanto, quando SoT foi combinado com SV (cenário 12), os avaliadores divergiram: o Gemini posicionou este cenário entre os melhores, enquanto o Claude apresentou valorização mais moderada, possivelmente penalizando a complexidade estrutural adicional mesmo quando tecnicamente bem-executada.

O RAG e o ZS-CoT receberam tratamento relativamente consistente de ambos os avaliadores, sendo reconhecidos como técnicas que contribuem positivamente sem impor custos qualitativos ou temporais desproporcionais. Esta convergência sugere que há aspectos das técnicas de *prompting* cujo valor é capturado de forma relativamente universal, independentemente das peculiaridades do avaliador.

A escolha entre Claude LaaJ e Gemini LaaJ deve considerar os requisitos específicos da aplicação. Para sistemas que priorizam eficiência temporal em cenários de baixa a média complexidade, o Gemini oferece vantagens substanciais, com tempos de execução significativamente reduzidos. Sua aplicação sistemática de todos os critérios avaliativos também beneficia contextos que requerem cobertura exaustiva e justificativas detalhadas das avaliações.

Para aplicações que necessitam de maior poder discriminativo entre modelos similares ou que valorizam avaliações contextualmente adaptativas, o Claude LaaJ apresenta vantagens. Sua capacidade de diferenciar claramente entre modelos mesmo em cenários simples e sua tendência a equilibrar múltiplos aspectos qualitativos — incluindo clareza e eficiência comunicativa — podem ser preferíveis em contextos educacionais ou de desenvolvimento onde nuances qualitativas são críticas.

Em cenários de máxima complexidade que incorporam SV, as diferenças entre avaliadores se atenuam tanto em termos de custos temporais quanto de ordenações qualitativas. Esta convergência sugere que, para aplicações focadas em configurações complexas, a escolha do avaliador torna-se menos determinante, podendo-se priorizar outros fatores como integração técnica, custos de API, ou familiaridade da equipe com o modelo específico.

A comparação entre Claude LaaJ e Gemini LaaJ evidencia que a variação do modelo avaliador pode influenciar a magnitude dos scores e a ordenação relativa entre cenários constituindo decisão metodológica não-neutra. O Claude oferece maior poder discriminativo e avaliação contextualmente adaptativa, enquanto o Gemini proporciona cobertura criterial sistemática e eficiência temporal superior em cenários simples. Ambos convergem em aspectos fundamentais — validando a importância da SV e identificando limitações consistentes de determinadas configurações — mas divergem em nuances que podem ser determinantes conforme o contexto de aplicação.

Esta dualidade reforça que a abordagem LaaJ, embora promissora como ferramenta de avaliação automática escalável, não elimina a necessidade de julgamento metodológico criterioso. A seleção do avaliador, a interpretação de seus resultados, e o reconhecimento de suas limitações permanecem responsabilidades essenciais do pesquisador. Os resultados sugerem que, em vez de buscar o “melhor” avaliador universal, deve-se compreender os perfis de *trade-offs* de cada opção e selecionar aquela cujas características se alinham mais adequadamente aos objetivos e restrições do estudo específico.

Diante do exposto, em resposta à **QP3**, conclui-se que as diferentes combinações de técnicas de PE impactam de maneira significativa e multifacetada a qualidade da geração e da explicação de código, refletindo-se nos níveis de severidade das inconsistências identificadas, na adequação técnica das soluções geradas e na clareza e completude das explicações, quando avaliadas por *LLM-as-a-Judge*. Esse impacto é fortemente condicionado tanto pelas técnicas empregadas quanto pelo perfil avaliativo do julgador utilizado. Os resultados evidenciam consenso robusto quanto ao papel central da SV como fator estabilizador e promotor de qualidade, elevando consistentemente os níveis de adequação técnica e explicativa das respostas, independentemente do avaliador. Em contrapartida, combinações que incorporam SoT sem mecanismos de validação mostraram-se sistematicamente problemáticas, indicando limitações claras dessa configuração sob avaliação automatizada.

As divergências observadas entre Claude LaaJ e Gemini LaaJ demonstram que a percepção da qualidade das respostas varia conforme a ênfase atribuída a critérios como completude técnica, clareza comunicativa e eficiência estrutural, revelando que o impacto das combinações de PE não é absoluto, mas mediado pelo modelo avaliador. Assim, os achados do Experimento 3 indicam que a qualidade da geração e da explicação de código resulta da interação entre a combinação de técnicas de PE, o modelo gerador e o julgador automático adotado, reforçando que avaliações baseadas em LaaJ exigem interpretação contextualizada e escolhas metodológicas conscientes.

## 5.4 Ameaças à Validade

Uma potencial ameaça à validade do processo avaliativo adotado refere-se à natureza estática e sequencial do *prompt template* utilizado no paradigma LaaJ. Especificamente, o protocolo de avaliação foi estruturado em etapas fixas e sempre apresentadas na mesma ordem, o que pode introduzir vieses associados à posição dos elementos no prompt. Estudos recentes indicam que modelos de linguagem são sensíveis à organização estrutural das instruções, apresentando fenômenos como *position bias* e *order effects*, nos quais informações apresentadas em posições iniciais tendem a exercer maior influência no processo de julgamento (SHI et al., 2025; WANG et al., 2024).

Nesse contexto, a execução sequencial e invariável das etapas pode levar o modelo avali-

ador a priorizar determinadas fases do julgamento — especialmente as iniciais — em detrimento de outras, comprometendo a uniformidade da análise. Além disso, a primeira etapa pode atuar como um mecanismo de ancoragem cognitiva, influenciando as interpretações subsequentes realizadas pelo modelo. Como estratégia de mitigação, a literatura sugere abordagens como a randomização da ordem das etapas, execução múltipla com diferentes configurações de *prompt* ou decomposição do processo avaliativo em subtarefas independentes, visando reduzir a influência de vieses estruturais e aumentar a robustez dos resultados.

Outra ameaça relevante à validade está associada ao uso da métrica BERTScore para a avaliação da geração de código. Embora essa métrica seja eficaz para capturar similaridade semântica entre o código gerado e o código de referência, ela não garante a corretude funcional da solução, uma vez que um código pode apresentar alta similaridade semântica em relação à referência e, ainda assim, conter erros lógicos, falhas de execução ou comportamento incorreto quando aplicado ao problema proposto. Estudos recentes mostram que métricas baseadas em *embeddings*, como o CodeBERTScore, apresentam correlação limitada com a corretude funcional quando avaliadas por meio da execução do código, evidenciando que tais métricas podem não capturar adequadamente o comportamento real das soluções geradas (NAIK, 2024).

Tais limitações não invalidam os resultados obtidos, mas devem ser consideradas na interpretação dos achados, especialmente no que se refere à confiabilidade e à generalização do processo avaliativo automatizado, bem como às possíveis restrições na avaliação da corretude funcional das soluções geradas.

## 6 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo investigar, de forma sistemática, o impacto do uso combinado de técnicas de PE em tarefas de geração e explicação de código Python, com especial atenção ao papel de técnicas pouco exploradas na literatura quando integradas a estratégias mais consolidadas.

Em resposta à **QP4** — questão transversal que permeia todos os experimentos conduzidos — os resultados obtidos ao longo dos Experimentos 1, 2 e 3 indicam que técnicas de PE menos estudadas, quando combinadas de forma criteriosa a abordagens consolidadas, contribuem de maneira consistente para a melhoria da qualidade das respostas geradas por LLMs, tanto sob métricas quantitativas de similaridade semântica quanto sob avaliações qualitativas automatizadas. Observa-se que esses ganhos não decorrem da simples adição de complexidade, mas da exploração de interações sinérgicas entre técnicas, cujos efeitos se manifestam progressivamente nos treze cenários experimentais avaliados, nos seis modelos investigados e sob os critérios de avaliação considerados.

Os resultados do Experimento 1 indicam que determinadas combinações estratégicas de técnicas de PE apresentaram desempenho estatisticamente superior ao cenário de referência (*baseline*), especialmente sob a métrica BERTScore. O cenário que combina *Role Prompting* e RAG alcançou valores superiores a 0,98, configurando o melhor desempenho observado nesse experimento sob essa métrica. A análise estatística por meio do teste de Wilcoxon signed-rank confirmou que múltiplos cenários apresentaram diferenças significativas em relação ao *baseline*. Entretanto, também se verificou que a adição indiscriminada de técnicas pode comprometer o desempenho: combinações que empregaram SoT sem mecanismos de verificação degradaram consistentemente os resultados em todos os modelos avaliados. Esse achado reforça que a eficácia da engenharia de *prompts* depende menos da complexidade acumulada e mais da compatibilidade funcional entre as técnicas combinadas.

No Experimento 2, a ampliação de dois para seis modelos revelou padrões arquiteturais relevantes. DeepSeek-v3 e Mixtral-8x22b-instruct demonstraram robustez superior em geração de código, com o Mixtral alcançando o melhor score absoluto da pesquisa (0,973) aliado a elevada eficiência temporal. Em contraste, Llama3.1-405b apresentou desempenho consistentemente inferior, resultado contraintuitivo diante de sua escala. Esse achado evidencia que eficácia em domínios específicos não depende exclusivamente do número de parâmetros, mas de estratégias de treinamento e alinhamento voltadas para a tarefa. A caracterização destes perfis modelo-específicos constitui contribuição prática importante, permitindo que desenvolvedores e pesquisadores selecionem modelos adequados conforme suas necessidades.

O Experimento 3.1, ao empregar Claude como avaliador LaaJ, incorporou dimensões

qualitativas como clareza, profundidade e coerência explicativa. A análise confirmou a robustez do DeepSeek-v3 e destacou o cenário  $S_{10}$  (Role + ZS-CoT + SV + RAG) como configuração de equilíbrio entre qualidade e eficiência na geração e explicação de código. Também foi reiterado o padrão negativo associado ao uso de SoT sem verificação adequada.

No Experimento 3.2, ao utilizar o Gemini como avaliador sobre as mesmas saídas dos seis modelos do Experimento 3.1, investigou-se a estabilidade das avaliações quando se altera o modelo julgador. O Gemini apresentou maior aderência aos 22 critérios definidos, resultando em valores absolutos mais elevados em comparação ao Claude. A análise temporal revelou um comportamento distinto: elevada eficiência em cenários que não incorporavam SV, seguida de aumento substancial no tempo de processamento quando a verificação iterativa era introduzida. Apesar das diferenças, ambos os avaliadores convergiram quanto à importância central da SV, à robustez do DeepSeek-v3 e às limitações do Gemma3-27b. Observou-se, contudo, que o Claude apresentou maior poder discriminativo entre modelos, enquanto o Gemini produziu avaliações mais homogêneas.

Entre as técnicas menos estudadas, a *Self-Verification* emergiu como o elemento de maior impacto positivo. Cenários que incorporaram essa técnica apresentaram melhorias qualitativas consistentes tanto nas avaliações por BERTScore quanto nos julgamentos automatizados LaaJ. Todavia, tais ganhos vieram acompanhados de aumento substancial do custo computacional, com multiplicação de 10 a 30 vezes nos tempos de execução. Esse *trade-off* entre qualidade e eficiência constitui um dos achados centrais da pesquisa: não existe estratégia universalmente superior, mas um conjunto de alternativas cuja adequação depende dos requisitos da aplicação.

De forma geral, este trabalho demonstra que as combinações de técnicas de PE, quando fundamentadas em compreensão sistemática de técnicas e suas interações, constituem abordagem poderosa para otimizar o desempenho de LLMs em tarefas de geração e explicação de código. Os resultados indicam que combinações estrategicamente estruturadas tendem a apresentar desempenho superior ao cenário de referência, desde que respeitadas as compatibilidades entre técnicas, a arquitetura do modelo e os requisitos da aplicação.

## 6.1 Contribuições

As contribuições desta pesquisa estendem-se além dos achados empíricos específicos. Ao propor uma abordagem metodológica inovadora, utilizando *framework* e combinando avaliação quantitativa rigorosa com julgadores automatizados, este trabalho demonstra que é possível e necessário ir além de avaliações *ad hoc* ou exclusivamente qualitativas no estudo de PE. O mapeamento sistemático de 13 cenários experimentais aplicados a múltiplos LLMs e analisados sob diferentes critérios avaliativos configura um estudo de caráter abrangente sobre combinações de técnicas de PE voltadas à geração e explicação de código. De forma mais específica, as contribuições podem ser compreendidas em quatro dimensões principais.

Em primeiro lugar, destaca-se a sistematização e a justificativa racionalizada para a seleção e combinações de técnicas de PE. Diferentemente de abordagens exploratórias ou arbitrárias, esta pesquisa fundamentou suas escolhas na literatura especializada, analisando previamente possíveis interações, sinergias e conflitos entre técnicas antes de operacionalizá-las experimentalmente. A construção de 13 cenários experimentais permitiu examinar não apenas o efeito isolado das técnicas, mas sobretudo seus comportamentos combinados, oferecendo uma compreensão aprofundada das dinâmicas entre estratégias consolidadas e técnicas menos exploradas.

Em segundo lugar, a pesquisa realiza avaliações do impacto dessas combinações na geração e explicação de código, por meio de um desenho experimental estruturado envolvendo seis LLMs com diferentes arquiteturas. A seleção dos modelos foi fundamentada em critérios de diversidade arquitetural, capacidade de raciocínio e viabilidade experimental, assegurando que a análise permanecesse centrada no comportamento das combinações de técnicas e não em comparações isoladas entre modelos. O mapeamento dos 13 cenários experimentais sob critérios avaliativos complementares fornece evidências empíricas robustas sobre *trade-offs* entre qualidade, coerência explicativa e custo computacional. A partir desses resultados, derivam-se diretrizes práticas para o desenvolvimento de aplicações baseadas em LLMs, indicando que mecanismos de verificação iterativa tendem a favorecer consistência quando a precisão é requisito central; que estratégias de decomposição do raciocínio demandam validação adequada para evitar degradação da coerência; que o desempenho das combinações depende de características arquiteturais específicas dos modelos; e que decisões de projeto devem considerar explicitamente o equilíbrio entre qualidade da resposta e custo computacional.

Em terceiro lugar, destaca-se a concepção de um *framework* experimental integrado, desenvolvido para sustentar a condução desta pesquisa com rigor metodológico, controle estrutural e rastreabilidade das decisões experimentais. Sua arquitetura foi estruturada com base em boas práticas consolidadas de engenharia de software, priorizando modularidade, separação de responsabilidades e baixo acoplamento, o que assegurou extensibilidade e manutenção facilitada ao longo da evolução da investigação. Essa organização permitiu incorporar novos modelos, ajustar parâmetros e expandir estratégias avaliativas sem comprometer a consistência do desenho experimental.

Como resultado, o *framework* não apenas viabilizou a análise sistemática realizada nesta pesquisa, mas estabelece uma infraestrutura reutilizável para investigações futuras em *Prompt Engineering*. Ao favorecer reprodutibilidade, escalabilidade e transparência metodológica, essa contribuição extrapola o escopo específico do estudo e oferece um recurso técnico aplicável a outras pesquisas empíricas envolvendo LLMs.

Em quarto lugar, a pesquisa propõe um conjunto estruturado de 22 critérios para a avaliação automatizada da geração e das explicações de código no paradigma *LLM-as-a-Judge*. A principal inovação dessa proposta reside na integração, em um mesmo protocolo avaliativo, da

análise conjunta do código gerado e da explicação produzida pelo modelo, tratando-os como componentes interdependentes de uma única resposta. Até o momento, a utilização do paradigma LaaJ tem sido majoritariamente direcionada à avaliação isolada da geração ou da explicação de código, não contemplando de forma estruturada a coerência entre ambos.

Ao incorporar uma perspectiva que considera simultaneamente correção técnica, alinhamento semântico e qualidade explicativa, os critérios estabelecem um referencial avaliativo mais abrangente e alinhado às demandas reais de aplicações baseadas em LLMs. Organizados segundo uma escala de severidade e concebidos para simular aspectos do julgamento humano, esses critérios elevam o rigor metodológico da avaliação automatizada, ampliam a transparência do processo e oferecem uma base estruturada que pode ser reutilizada e adaptada em investigações futuras.

Adicionalmente, o código-fonte do *framework* e dos experimentos conduzidos nesta pesquisa encontra-se disponível publicamente em repositório GitHub<sup>29</sup>, permitindo a replicação, validação e extensão dos resultados por outros pesquisadores. Essa disponibilização reforça o compromisso com a transparência e a reprodutibilidade científica, ao oferecer acesso integral aos artefatos metodológicos que sustentam as análises realizadas neste trabalho.

## 6.2 Limitações

A interpretação dos resultados deve considerar algumas limitações inerentes ao escopo delimitado nesta pesquisa. A utilização de dez questões de programação em Python, embora suficiente para identificar padrões consistentes e estatisticamente relevantes, representa uma amostra limitada frente à diversidade de problemas e níveis de complexidade existentes no domínio da programação.

A concentração em uma única linguagem de programação limita a validade externa dos resultados, uma vez que diferenças estruturais e semânticas entre linguagens podem alterar o modo como técnicas de PE interagem com tarefas de geração e explicação de código.

Adicionalmente, as restrições operacionais associadas ao uso de contas *Free Tier* impactaram especialmente a execução de cenários mais complexos, como aqueles que empregam SoT, podendo influenciar os custos temporais observados.

Essas limitações não comprometem a validade dos resultados, mas delimitam o alcance de sua generalização, indicando oportunidades para investigações futuras em contextos mais amplos.

---

<sup>29</sup> GitHub da pesquisa

### 6.3 Trabalhos Futuros

Os achados e as limitações identificadas abrem múltiplas direções promissoras para investigações futuras. Em primeiro lugar, destaca-se a possibilidade de explorar outras técnicas de PE não contempladas nesta pesquisa, como *Self-Consistency*, com potencial para revelar novos padrões de combinações eficazes e ampliar o espectro de estratégias disponíveis em diferentes contextos de aplicação.

No âmbito avaliativo, investigações adicionais sobre o comportamento de modelos no paradigma LaaJ podem contribuir para o refinamento de critérios e protocolos de julgamento automatizado. A utilização de outros modelos como avaliadores — incluindo Grok e outros LLMs emergentes — permitiria expandir a abordagem metodológica iniciada com Claude e Gemini. Também se mostra pertinente um estudo específico sobre o comportamento dos avaliadores LaaJ diante de diferentes tipos de *prompts* na geração e explicação de código, aprofundando a compreensão de como esses juízes automáticos interpretam e priorizam critérios de qualidade, o que poderia fundamentar protocolos avaliativos mais robustos e padronizados.

Outra direção promissora envolve o aprofundamento do estudo sobre o comportamento das combinações de técnicas de PE, investigando sistematicamente quais técnicas são mais adequadas para determinados LLMs, o que poderia revelar princípios gerais de compatibilidade entre estratégias de *prompting* e arquiteturas específicas de modelos. Além disso, a avaliação de LLMs mais recentes, treinados com dados atualizados, permitiria examinar se os padrões observados nesta pesquisa se mantêm ou se evoluções arquiteturais e de treinamento alteram as dinâmicas de eficácia das combinações de técnicas de PE.

Adicionalmente, abre-se a possibilidade de aplicar os 13 cenários elaborados nesta pesquisa a outros domínios além da programação, como geração de texto científico, análise de sentimentos, sumarização de documentos ou resposta a perguntas factuais. Tal expansão permitiria investigar se os princípios identificados sobre eficácia de combinações de técnicas se generalizam para além do contexto específico de geração e explicação de código Python.

Por fim, uma evolução natural deste trabalho reside no aprimoramento do *framework* experimental desenvolvido. Futuras versões podem torná-lo mais consistente e parametrizável para múltiplas linguagens de programação e diferentes domínios, mediante a adoção de um *schema* unificado para definição de *datasets* e respectivos *ground truths*. Recomenda-se ainda sua modularização em bibliotecas reutilizáveis, bem como a adoção de tecnologias de containerização, permitindo execução em ambientes distintos sem dependência direta da configuração original do projeto. Tais aprimoramentos fortaleceriam o potencial do *framework* como infraestrutura de pesquisa aberta, extensível e sustentável para investigações em *Prompt Engineering*.

## REFERÊNCIAS BIBLIOGRÁFICAS

AGGARWAL, P. et al. Codesift: An llm-based reference-less framework for automatic code validation. In: IEEE. *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. [S.l.], 2024. p. 404–410. Citado na página 32.

ATIL, B. et al. *Non-Determinism of "Deterministic" LLM Settings*. 2025. Disponível em: <<https://arxiv.org/abs/2408.04667>>. Citado na página 17.

BANERJEE, S.; LAVIE, A. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In: *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. [S.l.: s.n.], 2005. p. 65–72. Citado na página 27.

BANSAL, P. Prompt engineering importance and applicability with generative ai. *Journal of Computer and Communications*, v. 12, 10 2024. Citado na página 17.

BHATTACHARYA, P. et al. Exploring large language models for code explanation. *arXiv preprint arXiv:2310.16673*, 2023. Citado 2 vezes nas páginas 17 e 26.

BIGGS, J. B.; COLLIS, K. F. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. [S.l.]: Academic press, 2014. Citado na página 25.

BROWN, P. F. et al. A statistical approach to language translation. In: *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*. [S.l.: s.n.], 1988. Citado na página 27.

BROWN, T. et al. Language models are few-shot learners. *Advances in neural information processing systems*, v. 33, p. 1877–1901, 2020. Disponível em: <[https://proceedings.neurips.cc/paper\\_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf)>. Citado na página 33.

CASELI, H. d. M.; NUNES, M. d. G. V. *Processamento de linguagem natural: conceitos, técnicas e aplicações em português*. 2024. Citado na página 21.

CHEN, L. et al. *A Survey on Evaluating Large Language Models in Code Generation Tasks*. 2025. Disponível em: <<https://arxiv.org/abs/2408.16498>>. Citado na página 23.

CHEN, M. et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. Citado 2 vezes nas páginas 24 e 45.

CHERNOZHUKOV, V. et al. Double/debiased machine learning for treatment and causal parameters. *arXiv preprint arXiv:1608.00060*, 2016. Disponível em: <<https://EconPapers.repec.org/RePEc:arx:papers:1608.00060>>. Citado na página 50.

CHOWDHERY, A. et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, v. 24, n. 240, p. 1–113, 2023. Citado na página 22.

COIGNION, T.; QUINTON, C.; ROUYOY, R. A performance study of llm-generated code on leetcode. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. [S.l.: s.n.], 2024. p. 79–89. Citado na página 19.

- DAMKE, G.; GREGORINI, D.; COPETTI, L. Avaliação da performance e correção da geração de código através de técnicas de engenharia de prompt: Um estudo comparativo. In: *Anais do XXI Congresso Latino-Americano de Software Livre e Tecnologias Abertas*. Porto Alegre, RS, Brasil: SBC, 2024. p. 400–403. Disponível em: <<https://sol.sbc.org.br/index.php/latinoware/article/view/31562>>. Citado 3 vezes nas páginas 45, 55 e 57.
- DENG, Y. et al. Rephrase and respond: Let large language models ask better questions for themselves. *arXiv preprint arXiv:2311.04205*, 2023. Citado 5 vezes nas páginas 35, 36, 37, 38 e 60.
- DEROY, A.; MAITY, S. *Code Generation and Algorithmic Problem Solving Using Llama 3.1 405B*. 2025. Disponível em: <<https://arxiv.org/abs/2409.19027>>. Citado na página 70.
- DEVLIN, J. et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Disponível em: <<https://arxiv.org/abs/1810.04805>>. Citado 2 vezes nas páginas 22 e 28.
- DING, H. et al. Reasoning and planning with large language models in code development. In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. [S.l.: s.n.], 2024. p. 6480–6490. Citado na página 17.
- DODGE, Y. *The concise encyclopedia of statistics*. [S.l.]: Springer Science & Business Media, 2008. Citado na página 103.
- FU, J. et al. Gptscore: Evaluate as you desire. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. [S.l.: s.n.], 2024. p. 6556–6576. Citado na página 31.
- GAO, J.; LIN, C.-Y. Introduction to the special issue on statistical language modeling. *ACM Transactions on Asian Language Information Processing*, Association for Computing Machinery, New York, NY, USA, v. 3, n. 2, p. 87–93, jun. 2004. ISSN 1530-0226. Disponível em: <<https://doi.org/10.1145/1034780.1034781>>. Citado na página 21.
- GIRARDI, C.; SOUZA, D.; RÊGO, A. S. da C. Unveiling power on combining prompt engineering techniques: An experimental evaluation on code generation. In: *SBBD 2025 - Full Papers ()*. Fortaleza, CE, Brazil: [s.n.], 2025. Citado 3 vezes nas páginas 71, 77 e 103.
- GOUVEIA, T.; ALBUQUERQUE, K. *SIM e C073: FERRAMENTAS DE APOIO AO ENSINO DE CIRCUITOS LÓGICOS*. 2021. Citado na página 80.
- GOUVEIA, T. et al. C073: ferramenta para apoio ao ensino de programação usando a metodologia de aprendizagem baseada em problemas. *Revista Principia*, v. 60, n. 1, p. 70–87, mar. 2023. Disponível em: <<https://periodicos.ifpb.edu.br/index.php/principia/article/view/5942>>. Citado 2 vezes nas páginas 80 e 86.
- HADA, R. et al. Are large language model-based evaluators the solution to scaling up multilingual evaluation? In: GRAHAM, Y.; PURVER, M. (Ed.). *Findings of the Association for Computational Linguistics: EACL 2024*. St. Julian's, Malta: Association for Computational Linguistics, 2024. p. 1051–1070. Disponível em: <<https://aclanthology.org/2024.findings-eacl.71/>>. Citado na página 75.

HASSANY, M. et al. *Human-AI Co-Creation of Worked Examples for Programming Classes*. 2024. Disponível em: <<https://arxiv.org/abs/2402.16235>>. Citado na página 30.

HENDRYCKS, D. et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021. Citado na página 50.

HOSSEINI, R. et al. Improving engagement in program construction examples for learning python programming. *International Journal of Artificial Intelligence in Education*, Springer, v. 30, n. 2, p. 299–336, 2020. Citado na página 52.

HU, T.; ZHOU, X.-H. Unveiling llm evaluation focused on metrics: Challenges and solutions. *arXiv preprint arXiv:2404.09135*, 2024. Citado 2 vezes nas páginas 26 e 27.

JI, Z. et al. Benchmarking and explaining large language model-based code generation: A causality-centric approach. *arXiv preprint arXiv:2310.06680*, 2023. Disponível em: <<https://arxiv.org/pdf/2310.06680>>. Citado 5 vezes nas páginas 9, 50, 51, 55 e 57.

KHOJAH, R. et al. The impact of prompt programming on function-level code generation. *IEEE Transactions on Software Engineering*, v. 51, n. 8, p. 2381–2395, 2025. Disponível em: <<https://doi.org/10.1109/TSE.2025.3587794>>. Citado 5 vezes nas páginas 17, 18, 53, 55 e 57.

KINCAID, J. P. et al. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Institute for Simulation and Training, University of Central Florida, 1975. Citado na página 49.

KOJIMA, T. et al. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, v. 35, p. 22199–22213, 2022. Citado 4 vezes nas páginas 17, 37, 60 e 62.

KORZYŃSKI, P. et al. Artificial intelligence prompt engineering as a new digital competence: Analysis of generative ai technologies such as chatgpt. *Entrepreneurial Business and Economics Review*, 2023. Disponível em: <<https://api.semanticscholar.org/CorpusID:264049829>>. Citado na página 15.

LEKSHMI-NARAYANAN, A.-B. et al. Explaining code examples in introductory programming courses: Llm vs humans. *Grantee Submission*, ERIC, 2024. Disponível em: <<https://eric.ed.gov/?id=ED655912>>. Citado 3 vezes nas páginas 52, 55 e 57.

LEWIS, M. et al. *BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension*. 2020. 7871–7880 p. Disponível em: <<https://aclanthology.org/anthology-files/anthology-files/pdf/acl/2020.acl-main.703.pdf>>. Citado na página 22.

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, v. 33, p. 9459–9474, 2020. Citado na página 41.

LI, D. et al. From generation to judgment: Opportunities and challenges of llm-as-a-judge. In: *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. [S.l.: s.n.], 2025. p. 2757–2791. Citado 2 vezes nas páginas 30 e 31.

LI, H.; LEUNG, J.; SHEN, Z. *Towards Goal-oriented Prompt Engineering for Large Language Models: A Survey*. 2024. Disponível em: <<https://arxiv.org/abs/2401.14043>>. Citado na página 16.

- LI, R. et al. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. Disponível em: <<https://weizmann.elsevierpure.com/en/publications/starcoder-may-the-source-be-with-you/>>. Citado na página 24.
- LIN, C.-Y. Rouge: A package for automatic evaluation of summaries. In: *Text summarization branches out*. [S.l.: s.n.], 2004. p. 74–81. Citado na página 27.
- LISTER, R. On the cognitive development of the novice programmer: and the development of a computing education researcher. In: *Proceedings of the 9th computer science education research conference*. [S.l.: s.n.], 2020. p. 1–15. Citado na página 25.
- LIU, P. et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, ACM New York, NY, v. 55, n. 9, p. 1–35, 2023. Disponível em: <<https://dl.acm.org/doi/pdf/10.1145/3560815>>. Citado 2 vezes nas páginas 16 e 33.
- LIU, X.; CROFT, W. B. Statistical language modeling for information retrieval. *Annu. Rev. Inf. Sci. Technol.*, v. 39, n. 1, p. 1–31, 2005. Citado na página 21.
- LIU, Y. et al. *G-eval: NLG evaluation using gpt-4 with better human alignment*. [S.l.]: Proceedings of the 2023 conference on empirical methods in natural language processing, 2023. 2511–2522 p. Citado na página 32.
- LIU, Y. et al. On the reliability and explainability of language models for program generation. *ACM Transactions on Software Engineering and Methodology*, ACM New York, NY, v. 33, n. 5, p. 1–26, 2024. Citado na página 17.
- LORCH, L. et al. Dibs: Differentiable bayesian structure learning. *Advances in Neural Information Processing Systems*, v. 34, p. 24111–24123, 2021. Citado na página 50.
- LUO, Z. et al. Faithfulpersona: Balancing faithfulness and personalization in code explanations through self-critique. In: *Findings of the Association for Computational Linguistics: NAACL 2025*. [S.l.: s.n.], 2025. p. 930–944. Citado 2 vezes nas páginas 29 e 30.
- MACNEIL, S. et al. *Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book*. 2022. 931–937 p. Citado 4 vezes nas páginas 16, 25, 26 e 49.
- MAHIR, A.; SHOHEL, M. M. C.; SALL, W. The role of ai in programming education: An exploration of the effectiveness of conversational versus structured prompting. In: \_\_\_\_\_. [S.l.]: Practitioner Research in College-Based Education, 2024. p. 319–352. ISBN 9798369314999. Citado na página 16.
- MEDEIROS, A. et al. Contrato360: uma aplicação de perguntas e respostas usando modelos de linguagem, documentos e bancos de dados. In: *Anais do XXXIX Simpósio Brasileiro de Bancos de Dados*. Porto Alegre, RS, Brasil: SBC, 2024. p. 155–166. ISSN 2763-8979. Disponível em: <<https://sol.sbc.org.br/index.php/sbbd/article/view/30690>>. Citado na página 15.
- MIKOLOV, T. et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. Citado na página 21.
- NAIK, A. On the limitations of embedding based methods for measuring functional correctness for code generation. *arXiv preprint arXiv:2405.01580*, 2024. Citado na página 129.

NEAL, B. Introduction to causal inference. *Course Lecture Notes (draft)*, v. 132, 2020. Citado na página 50.

NEVES, B. et al. Explorando o potencial e a viabilidade de llms open-source na análise de sentimentos. In: *Anais Estendidos do XV Congresso Brasileiro de Software: Teoria e Prática*. Porto Alegre, RS, Brasil: SBC, 2024. p. 89–98. ISSN 0000-0000. Disponível em: <[https://sol.sbc.org.br/index.php/cbsoft\\_estendido/article/view/30249](https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/30249)>. Citado 2 vezes nas páginas 16 e 17.

NIJKAMP, E. et al. *Codegen: An open large language model for code with multi-turn program synthesis*. [S.l.]: The Eleventh International Conference on Learning Representations, 2023. Citado na página 24.

NING, X. et al. Skeleton-of-thought: Large language models can do parallel decoding. *Proceedings ENLSP-III*, 2024. Citado 3 vezes nas páginas 17, 38 e 39.

OLI, P. et al. *The behavior of large language models when prompted to generate code explanations*. [S.l.]: Proceedings of The workshop on Generative AI for Education (GAIED): Advances, Opportunities, and Challenges, The Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS 2023), 2023. Citado 3 vezes nas páginas 48, 49 e 57.

OLI, P. et al. *Can LLMs Identify Gaps and Misconceptions in Students' Code Explanations?* 2024. Disponível em: <<https://arxiv.org/abs/2501.10365>>. Citado na página 24.

OUYANG, L. et al. *Training language models to follow instructions with human feedback*. 2022. 27730–27744 p. Citado na página 22.

OWEN, A.; ANDREW, M.; PADLSKI, L. Techniques and technologies-analyzing the underlying architectures of chatgpt-4 and llama in summarization and analysis. 2024. Citado na página 16.

PALLA, D.; SLABY, A. Evaluation of generative ai models in python code generation: A comparative study. *IEEE Access*, PP, p. 1–1, 01 2025. Citado na página 70.

PAPINENI, K. et al. Bleu: a method for automatic evaluation of machine translation. In: ISABELLE, P.; CHARNIAK, E.; LIN, D. (Ed.). *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, 2002. p. 311–318. Disponível em: <<https://aclanthology.org/P02-1040/>>. Citado na página 27.

PEEPERKORN, M. et al. Is temperature the creativity parameter of large language models? *arXiv preprint arXiv:2405.00492*, 2024. Citado na página 23.

POPOVIĆ, M. chrF: character n-gram F-score for automatic MT evaluation. In: BOJAR, O. et al. (Ed.). *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Lisbon, Portugal: Association for Computational Linguistics, 2015. p. 392–395. Disponível em: <<https://aclanthology.org/W15-3049/>>. Citado na página 27.

RADFORD, A. et al. Language models are unsupervised multitask learners. *OpenAI blog*, v. 1, n. 8, p. 9, 2019. Citado na página 22.

RAO, C. R.; SHI, X.; WU, Y. Approximation of the expected value of the harmonic mean and some applications. *Proceedings of the National Academy of Sciences*, v. 111, n. 44, p. 15681–15686, 2014. Disponível em: <<https://www.pnas.org/doi/abs/10.1073/pnas.1412216111>>. Citado na página 100.

REYNOLDS, L.; MCDONELL, K. Prompt programming for large language models: Beyond the few-shot paradigm. In: *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. [S.l.: s.n.], 2021. p. 1–7. Citado 4 vezes nas páginas 17, 33, 34 e 84.

SADIK, A. R.; GOVIND, S. *Benchmarking LLM for Code Smells Detection: OpenAI GPT-4.0 vs DeepSeek-V3*. [S.l.]: Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, 2025. 969–975 p. Citado na página 70.

SARKER, L. et al. Syntactic robustness for llm-based code generation. *arXiv preprint arXiv:2404.01535*, 2024. Citado na página 19.

SARSA, S. et al. Automatic generation of programming exercises and code explanations using large language models. In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. [s.n.], 2022. p. 27–43. Disponível em: <<https://dl.acm.org/doi/pdf/10.1145/3501385.3543957>>. Citado 2 vezes nas páginas 25 e 26.

SCHULHOFF, S. et al. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*, 2025. Disponível em: <<https://arxiv.org/abs/2406.06608>>. Citado 4 vezes nas páginas 15, 34, 35 e 36.

SCHULTE, C. et al. An introduction to program comprehension for computer science educators. In: *Proceedings of the 2010 ITiCSE Working Group Reports*. New York, NY, USA: Association for Computing Machinery, 2010. (ITiCSE-WGR '10), p. 65–86. ISBN 9781450306775. Disponível em: <<https://doi.org/10.1145/1971681.1971687>>. Citado na página 24.

SHI, L. et al. Judging the judges: A systematic study of position bias in LLM-as-a-judge. In: INUI, K. et al. (Ed.). *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*. Mumbai, India: The Asian Federation of Natural Language Processing and The Association for Computational Linguistics, 2025. p. 292–314. ISBN 979-8-89176-298-5. Disponível em: <<https://aclanthology.org/2025.ijcnlp-long.18/>>. Citado na página 128.

SHIN, J. et al. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *ArXiv*, abs/2310.10508, 2023. Disponível em: <<https://api.semanticscholar.org/CorpusID:264145854>>. Citado na página 17.

STEFANO, G. D.; SCHÖNHERR, L.; PELLEGRINO, G. Rag and roll: An end-to-end evaluation of indirect prompt manipulations in llm-based application frameworks. *arXiv preprint arXiv:2408.05025*, 2024. Citado na página 41.

TEAMGEMMA et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025. Citado na página 70.

TEAMQWEN et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, v. 2, n. 3, 2024. Citado na página 70.

- TONG, W.; ZHANG, T. *CodeJudge: Evaluating Code Generation with Large Language Models*. [S.l.]: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, 2024. 20032–20051 p. Citado 3 vezes nas páginas 27, 32 e 64.
- TOUVRON, H. et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. Citado na página 26.
- VASWANI, A. et al. *Attention Is All You Need*. 2023. Disponível em: <<https://arxiv.org/abs/1706.03762>>. Citado na página 22.
- VATSAL, S.; DUBEY, H. A survey of prompt engineering methods in large language models for different nlp tasks. *abs/2407.12994*, 2024. Disponível em: <<https://api.semanticscholar.org/CorpusID:271270817>>. Citado 3 vezes nas páginas 15, 16 e 17.
- VO, N. P. A.; PAULOVICKS, B.; SHEININ, V. Llm-as-a-judge for reference-less automatic code validation and refinement for natural language to bash in it automation. *arXiv preprint arXiv:2506.11237*, 2025. Citado na página 31.
- WANG, C.; LIU, X.; AWADALLAH, A. H. Cost-effective hyperparameter optimization for large language model generation inference. In: PMLR. *International Conference on Automated Machine Learning*. [S.l.], 2023. p. 21–1. Citado na página 22.
- WANG, P. et al. Large language models are not fair evaluators. In: KU, L.-W.; MARTINS, A.; SRIKUMAR, V. (Ed.). *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Bangkok, Thailand: Association for Computational Linguistics, 2024. p. 9440–9450. Disponível em: <<https://aclanthology.org/2024.acl-long.511/>>. Citado na página 128.
- WANG, P. et al. *What are Models Thinking about? Understanding Large Language Model Hallucinations "Psychology" through Model Inner State Analysis*. 2025. Disponível em: <<https://arxiv.org/abs/2502.13490>>. Citado na página 17.
- WANG, R. et al. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proceedings of the ACM on Software Engineering*, ACM New York, NY, USA, v. 2, n. ISSTA, p. 1955–1977, 2025. Citado na página 27.
- WANG, T.; ZHOU, N.; CHEN, Z. Enhancing computer programming education with llms: A study on effective prompt engineering for python code generation. *arXiv preprint arXiv:2407.05437*, 2024. Disponível em: <<https://web3.arxiv.org/abs/2407.05437>>. Citado 3 vezes nas páginas 44, 55 e 57.
- WANG, Y.; GUO, S.; TAN, C. W. From code generation to software testing: Ai copilot with context-based retrieval-augmented generation. *IEEE Software*, v. 42, n. 4, p. 34–42, 2025. Citado na página 41.
- WEI, J. et al. *Emergent Abilities of Large Language Models*. 2022. Disponível em: <<https://arxiv.org/abs/2206.07682>>. Citado na página 22.
- 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. Citado 2 vezes nas páginas 36 e 62.

- WENG, Y. et al. *Large language models are better reasoners with self-verification*. [S.l.]: Findings of the Association for Computational Linguistics: EMNLP 2023, 2023. 2550–2575 p. Citado 2 vezes nas páginas 39 e 40.
- WHALLEY, J. L. et al. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In: *Conferences in Research and Practice in Information Technology Series*. [S.l.: s.n.], 2006. Citado na página 25.
- WOOLSON, R. F. Wilcoxon signed-rank test. *Encyclopedia of biostatistics*, Wiley Online Library, v. 8, 2005. Citado na página 103.
- XU, H. et al. TacoPrompt: A collaborative multi-task prompt learning method for self-supervised taxonomy completion. In: BOUAMOR, H.; PINO, J.; BALI, K. (Ed.). *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, 2023. p. 15804–15817. Disponível em: <<https://aclanthology.org/2023.emnlp-main.979/>>. Citado na página 16.
- YANG, A. et al. *Qwen2 Technical Report*. 2024. Disponível em: <<https://arxiv.org/abs/2407.10671>>. Citado na página 70.
- YEO, S. et al. Framework for evaluating code generation ability of large language models. *ETRI Journal*, v. 46, n. 1, p. 106–117, 2024. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.2023-0357>>. Citado na página 17.
- YU, Y. et al. Dag-gnn: Dag structure learning with graph neural networks. In: PMLR. *International conference on machine learning*. [S.l.], 2019. p. 7154–7163. Citado na página 50.
- ZHANG, T. et al. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2020. Citado 2 vezes nas páginas 27 e 28.
- ZHAO, H. et al. Advancing single-and multi-task text classification through large language model fine-tuning. *arXiv preprint arXiv:2412.08587*, 2024. Citado na página 27.
- ZHAO, W. X. et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, v. 1, n. 2, 2025. Disponível em: <<https://arxiv.org/abs/2303.18223v16>>. Citado 4 vezes nas páginas 21, 22, 23 e 24.
- ZHAO, Y. et al. *CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding?* [S.l.]: Proceedings of the 31st International Conference on Computational Linguistics, 2025. 73–95 p. Citado na página 27.
- ZHENG, K. et al. What makes large language models reason in (multi-turn) code generation? Singapore, maio 2025. Disponível em: <<https://hal.science/hal-05070997>>. Citado 2 vezes nas páginas 46 e 57.
- ZHOU, S. et al. *Codebertscore: Evaluating code generation with pretrained models of code*. [S.l.]: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, 2023. 13921–13937 p. Citado na página 32.
- ZHOU, Y. et al. *Large language models are human-level prompt engineers*. [S.l.]: The eleventh international conference on learning representations, 2022. Citado na página 37.

ZHUO, T. Y. Ice-score: Instructing large language models to evaluate code. In: *Findings of the Association for Computational Linguistics: EACL 2024*. [S.l.: s.n.], 2024. p. 2232–2242. Citado na página 32.

ZHUO, T. Y. et al. *BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions*. United States of America: International Conference on Learning Representations (ICLR), 2025. 99488–99542 p. Disponível em: <<https://openreview.net/group?id=ICLR.cc/2025/Conference>>. Citado na página 70.

ZIMMERMAN, J. W. et al. Tokens, the oft-overlooked appetizer: Large language models, the distributional hypothesis, and meaning. *arXiv preprint arXiv:2412.10924*, 2024. Citado na página 23.

## **Apêndices**

## APÊNDICE A – EXEMPLO DE CÓDIGO E EXPLICAÇÃO NA LINGUAGEM JAVA

### Exemplo de Código do Point.java

```
public class Point {  
    private int x ;  
    private int y ;  
  
    public void setX ( int newX ) {  
        this. x = newX ;  
    }  
    public int getX () {  
        return x ;  
    }  
  
    public void setY ( int newY ) {  
        this. y = newY ;  
    }  
    public int getY () {  
        return y ;  
    }  
  
    public void translate ( int dx , int dy ) {  
        x += dx ;  
        y += dy ;  
    }  
  
    public static void main ( String [] args ) {  
  
        Point point = new Point () ;  
        point.setX (7) ;  
        point.setY (2) ;  
  
        point.translate (11 , 6) ;  
        System.out.println ( " The new coordinates : ( "  
            + point.getX () + " , " + point.getY () + " ) " ) ;  
    }  
}
```

## Explicação do código pelo LLM

### Versão do artigo

This Java code defines a class named “Point” that represents a point in a two-dimensional space. The class has two private instance variables “x” and “y” that represent the x-coordinate and y-coordinate of the point, respectively.

The class provides getter and setter methods for both “x” and “y”. The “setX(int newX)” and “setY(int newY)” methods are used to set the values of “x” and “y”, respectively. The “getX()” and “getY()” methods are used to retrieve the current values of “x” and “y”, respectively.

The “translate(int dx, int dy)” method is used to change the position of the point by a certain amount. The parameters “dx” and “dy” represent the amount to change the x-coordinate and y-coordinate, respectively. The method adds “dx” to “x” and “dy” to “y”, effectively moving the point to a new position.

The “main(String[] args)” method is the entry point of the program. It creates a new “Point” object, sets its “x” and “y” values to 7 and 2, respectively, then translates the point by 11 in the x-direction and 6 in the y-direction. Finally, it prints out the new coordinates of the point.

This code is a good example of encapsulation in object-oriented programming. The “x” and “y” variables are hidden from other classes and can only be accessed through the getter and setter methods. This ensures that the internal state of the “Point” object is always controlled and can’t be changed arbitrarily by other parts of the program.

### Versão traduzida

Este código Java define uma classe chamada “Point” que representa um ponto em um espaço bidimensional. A classe tem duas variáveis de instância privadas, “x” e “y”, que representam a coordenada x e a coordenada y do ponto, respectivamente.

A classe fornece métodos getter e setter para “x” e “y”. Os métodos “setX(int newX)” e “setY(int newY)” são usados para definir os valores de “x” e “y”, respectivamente. Os métodos “getX()” e “getY()” são usados para recuperar os valores atuais de “x” e “y”, respectivamente.

O método “translate(int dx, int dy)” é usado para alterar a posição do ponto em uma determinada quantidade. Os parâmetros “dx” e “dy” representam a quantidade a ser alterada na coordenada x e na coordenada y, respectivamente. O método adiciona “dx” a “x” e “dy” a “y”, movendo efetivamente o ponto para uma nova posição.

O método “main(String[] args)” é o ponto de entrada do programa. Ele cria um novo objeto “Point”, define seus valores “x” e “y” como 7 e 2, respectivamente, e então transla o ponto em 11 na direção x e 6 na direção y. Por fim, ele imprime as novas coordenadas do ponto.

Este código é um bom exemplo de encapsulamento na programação orientada a objetos. As variáveis “x” e “y” ficam ocultas de outras classes e só podem ser acessadas por meio dos métodos getter e setter. Isso garante que o estado interno do objeto “Point” seja sempre controlado e não possa ser alterado arbitrariamente por outras partes do programa.

## APÊNDICE B – IMPLEMENTAÇÃO DA CAMADA DE INTEGRAÇÃO COM APIS DE LLMS

### B.1 Requisição via API OpenAI

```

Logger.info("Creating a model response for the given chat conversation...")
self.client = OpenAI(api_key = self.key)
self.model_name = self.model
response = self.client.chat.completions.create(
    model=self.model_name,
    messages=messages
)
try:
    return response.choices[0].message.content
except Exception as e:
    Logger.error(f"Error in the model response: {e}")
    return "Sorry, I am not able to answer that question."

```

### B.2 Requisição via API ApiLLM (HTTP requests)

```

Logger.info(f"Sending request to model {self.model}...")
headers = {
    "Authorization": f"Bearer {self.key}",
    "Content-Type": "application/json",
}
data = {
    "model": self.model,
    "messages": messages
}
response = requests.post(self.url, headers=headers, json=data)

try:
    df_nested = pd.json_normalize(response.json())
    df_nested_choices = pd.DataFrame(df_nested.choices[0])
    df_nested_message = pd.DataFrame([df_nested_choices.message[0]])
    return df_nested_message.content[0]
except Exception as e:
    Logger.error(f"Error in the model {self.model} response: {e}")
    return "Sorry, I am not able to answer that question."

```

### B.3 Requisição via API Claude (Anthropic)

```

Logger.info("Creating a model response for the given chat conversation...")
self.client = Anthropic(api_key = self.key)
self.model_name = self.model
response = self.client.messages.create(
    model=self.model_name,
    max_tokens=20000,
    temperature=0,

```

```
        messages=messages
    )

    try:
        return response.content[0].text
    except Exception as e:
        Logger.error(f"Error in the model response: {e}")
        return "Sorry, I am not able to answer that question."
```

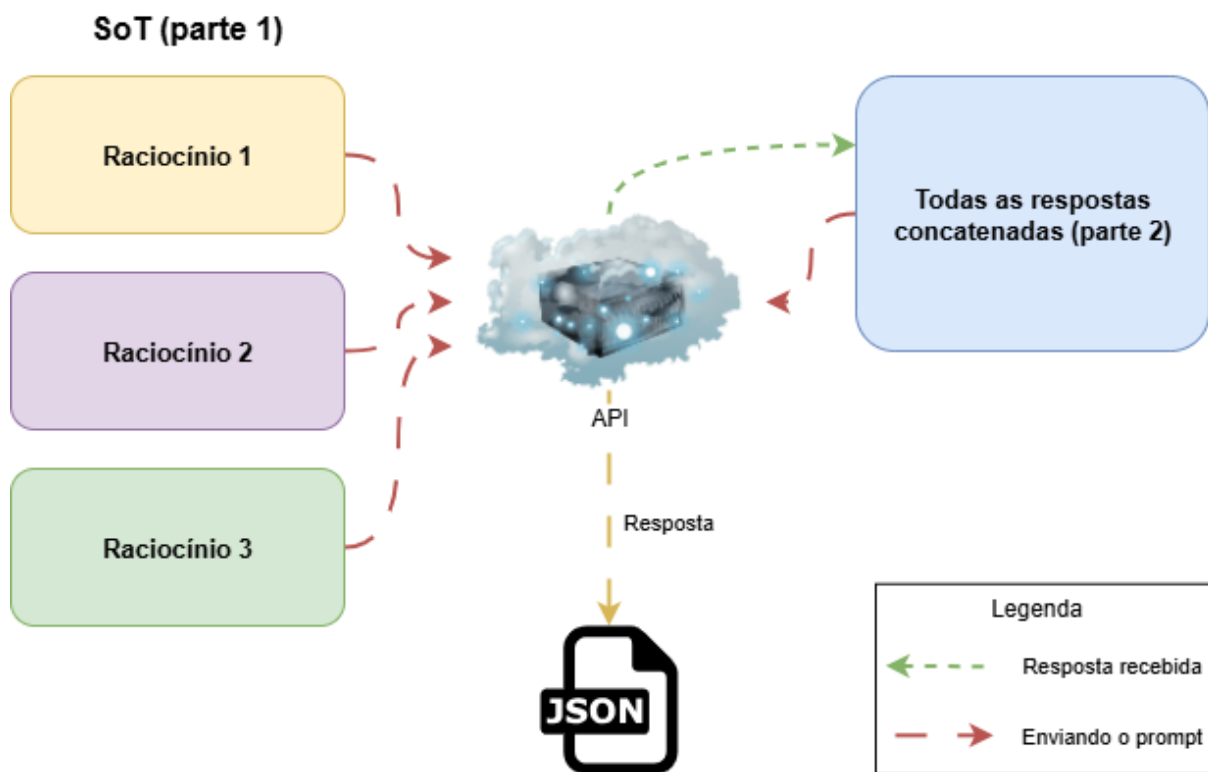
#### B.4 Requisição via API Gemini

```
Logger.info("Creating a model response for the given chat conversation...")
self.client = Client(api_key = self.key)
self.model_name = self.model
response = self.client.models.generate_content(
    model=self.model_name,
    contents=messages
)

try:
    return response.text
except Exception as e:
    Logger.error(f"Error in the model response: {e}")
    return "Sorry, I am not able to answer that question."
```

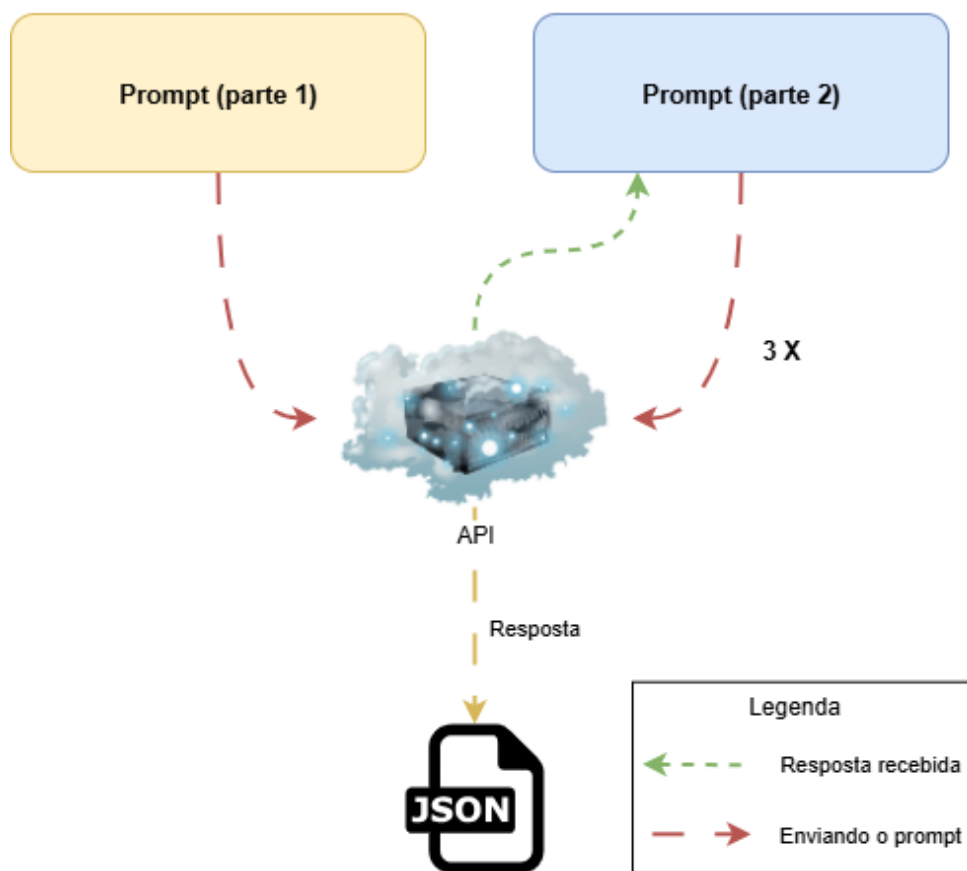
## APÊNDICE C – WORKFLOWS DOS CENÁRIOS

Figura 24 – *Workflow* do cenário S<sub>5</sub>



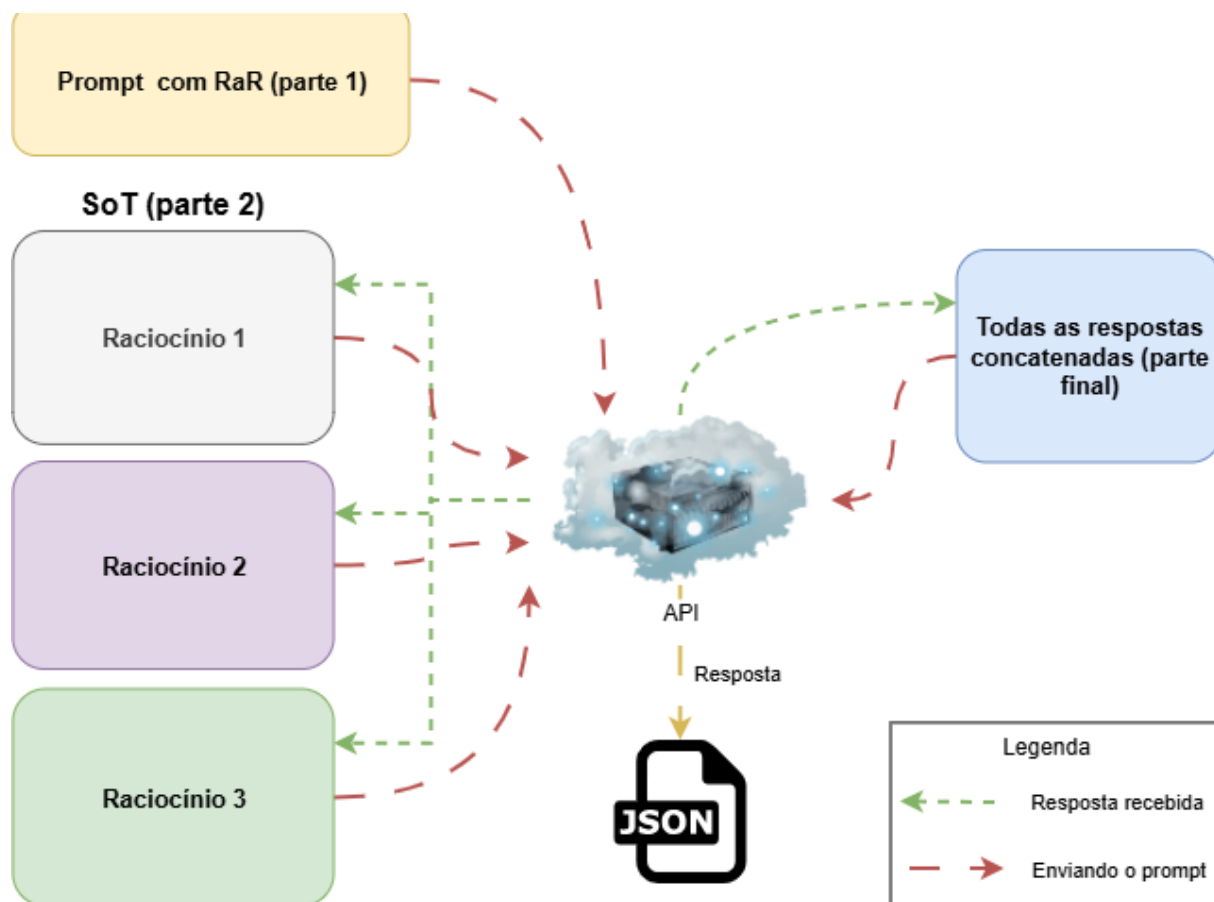
Elaborada pelo Autor

Figura 25 – Workflow do cenário S<sub>6</sub>



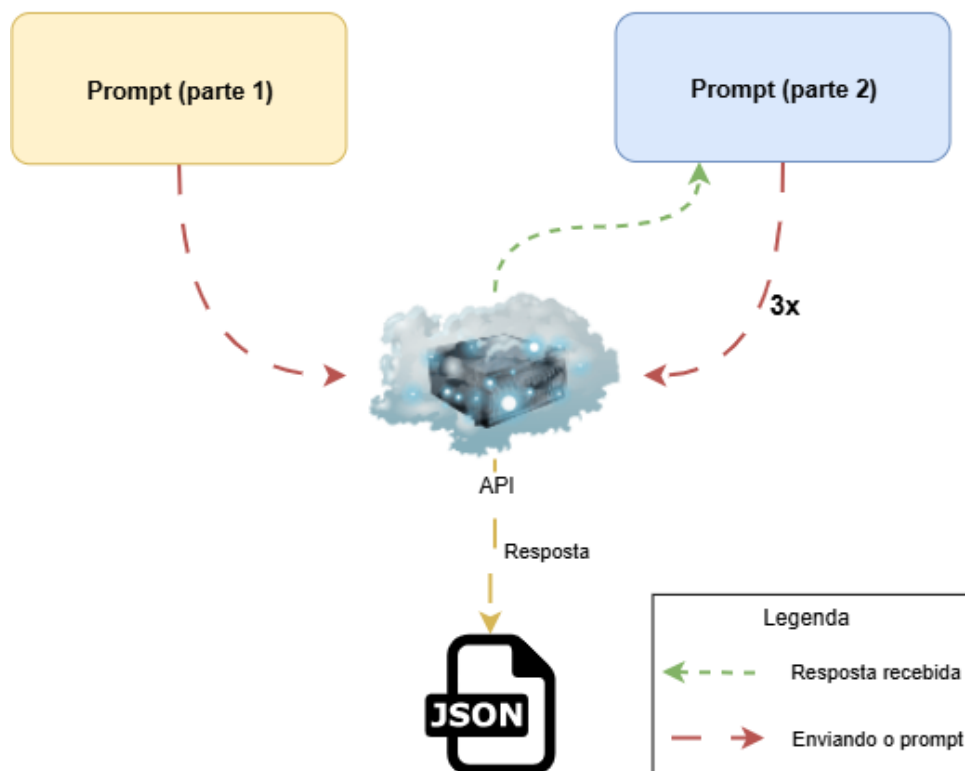
Elaborada pelo Autor

Figura 26 – Workflow do cenário S7



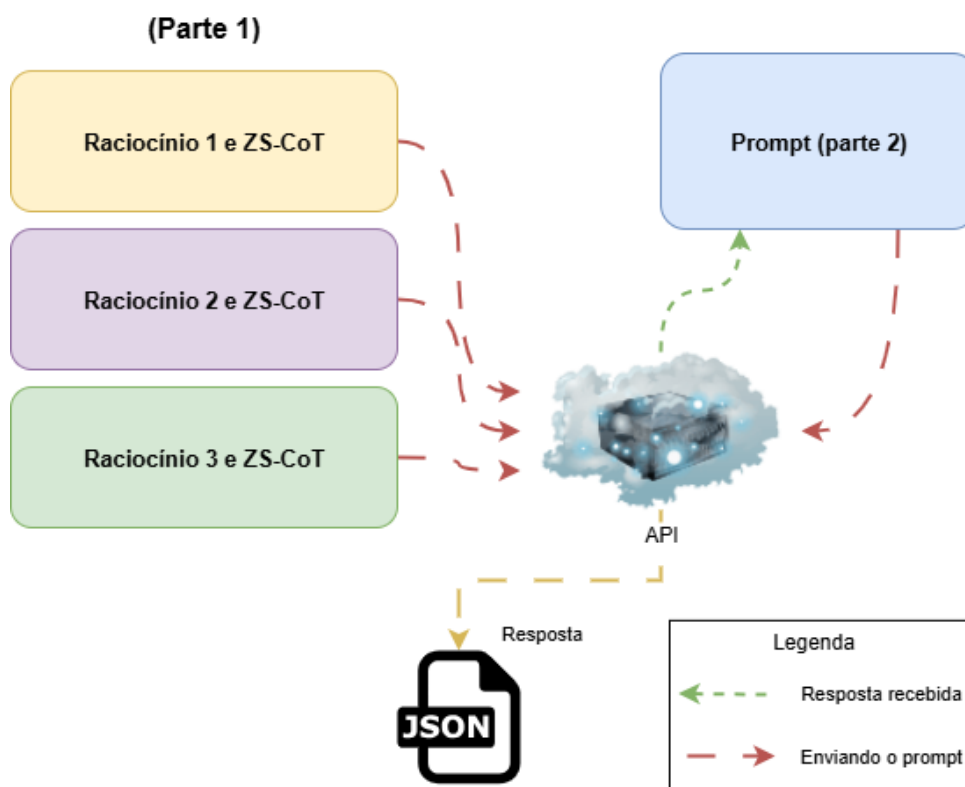
Elaborada pelo Autor

Figura 27 – Workflow do cenário S8



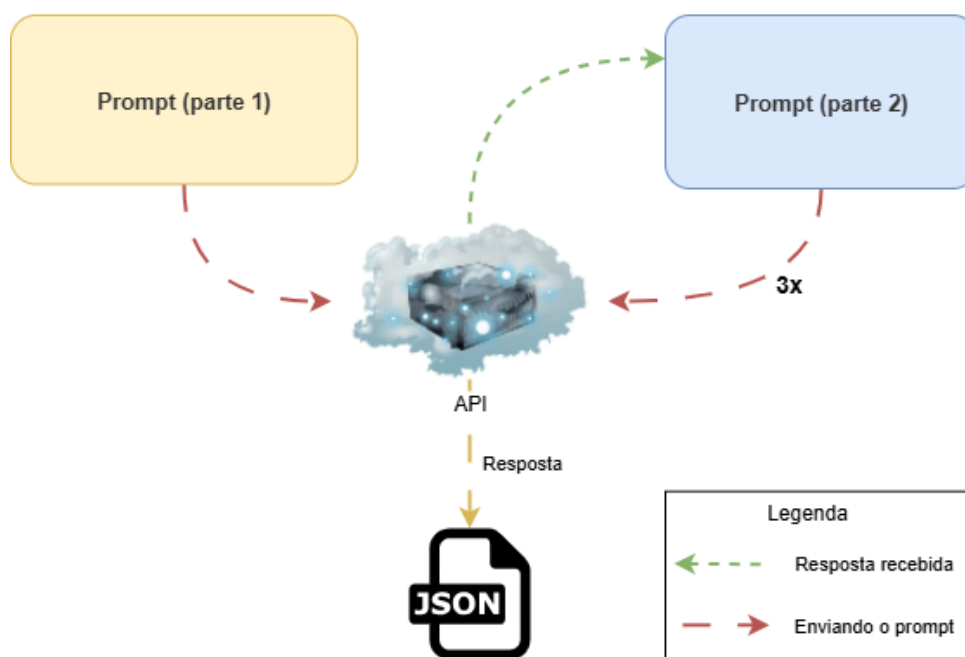
Elaborada pelo Autor

Figura 28 – Workflow do cenário S<sub>9</sub>



Elaborada pelo Autor

Figura 29 – Workflow do cenário S<sub>10</sub>



Elaborada pelo Autor

Figura 30 – Workflow do cenário S<sub>11</sub>

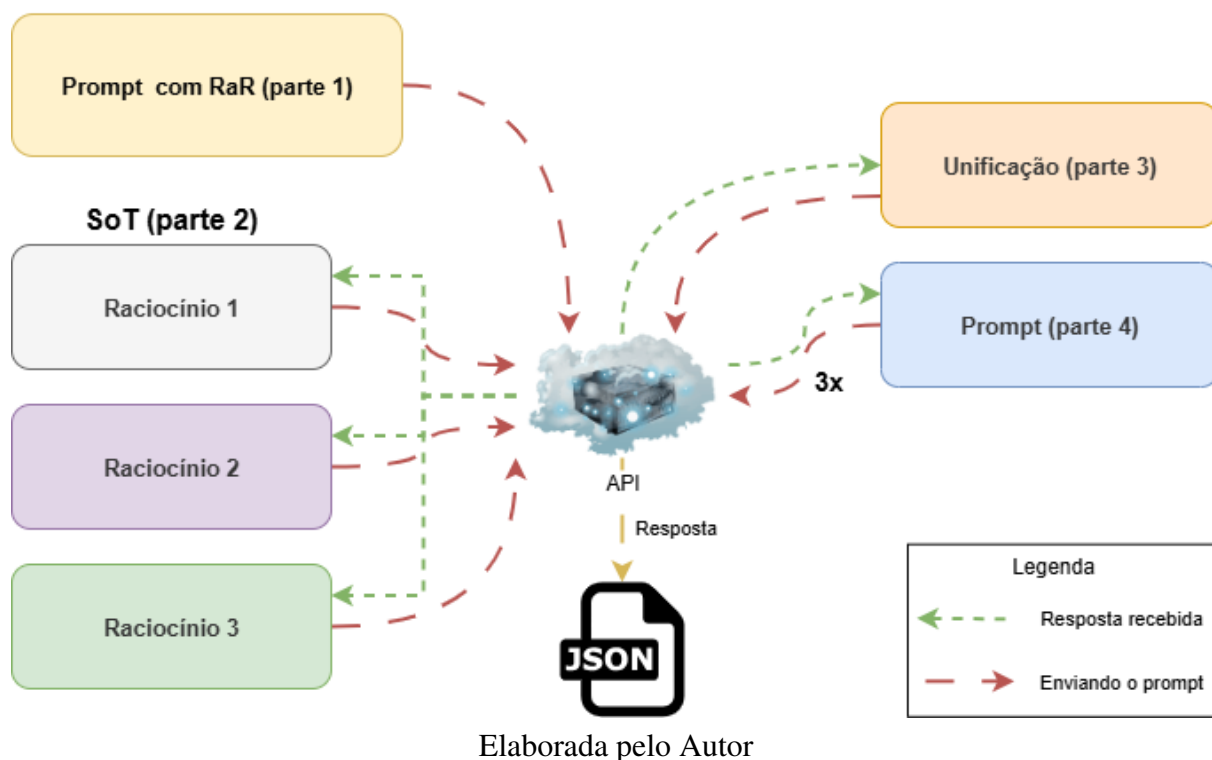
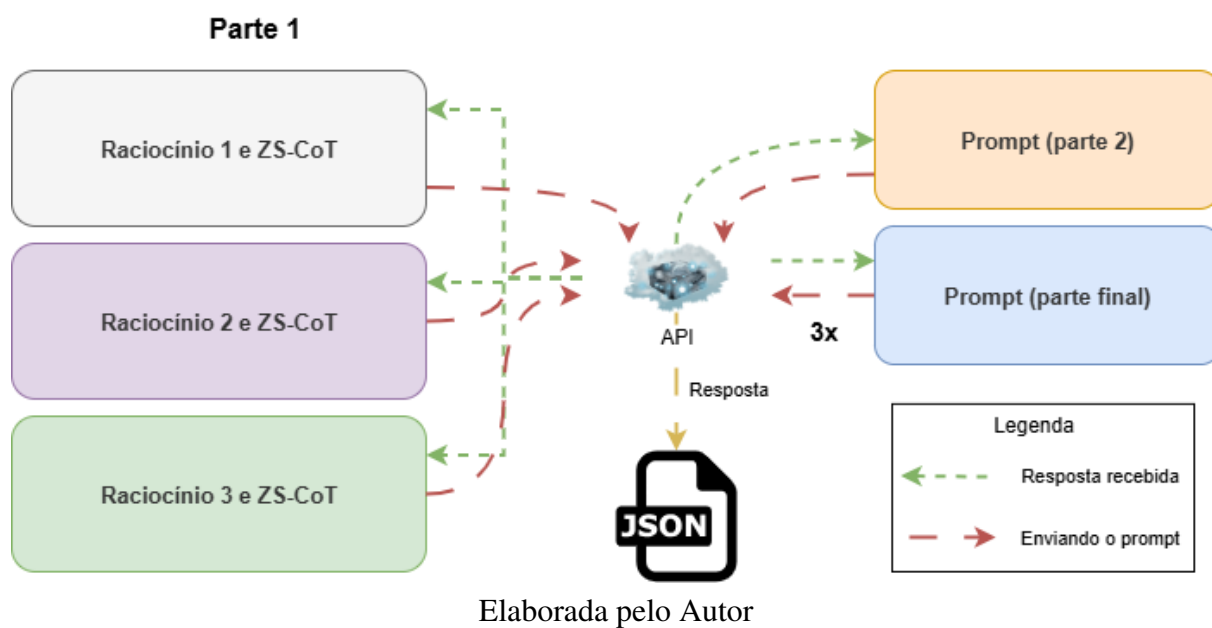
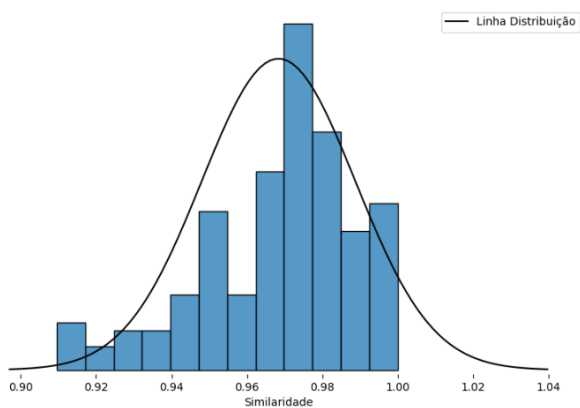


Figura 31 – Workflow do cenário S<sub>12</sub>



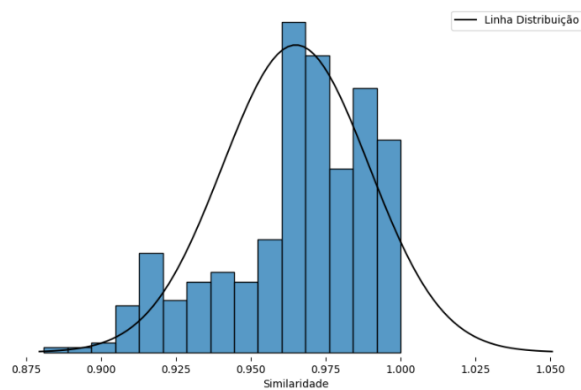
## APÊNDICE D – GRÁFICOS DE NORMALIDADE DE DISTRIBUIÇÃO DOS VALORES DE BERTSCORE

Figura 32 – OpenAI



Elaborada pelo Autor

Figura 33 – LLaMA



Elaborada pelo Autor

## APÊNDICE E – EXEMPLO DE RESULTADO DO EXPERIMENTO 2

Código 1: Saída do Experimento 2 - Cenário S<sub>6</sub>

### Ground-truth

```
def main():
    N = int(input())
    matriz = []
    count_1 = count_2 = count_3 = 0
    for i in range(N):
        matriz.append([0] * N)
    for i in range(N):
        matriz[i] = input().split()
        matriz[i] = list(map(int,
            matriz[i]))
    for i in range(N):
        for j in range(N):
            if matriz[i][j] <= 90:
                matriz[i][j] = '+'
                count_1 += 1
            elif matriz[i][j] > 90 and
                matriz[i][j] <= 100:
                matriz[i][j] = 'o'
                count_2 += 1
            else:
                matriz[i][j] = '-'
                count_3 += 1
    for i in range(N):
        for j in range(N):
            print(matriz[i][j], end='')
        print()
    print()
    print(f'+: {count_1}')
    print(f'o: {count_2}')
    print(f'-: {count_3}')

if __name__ == "__main__":
    main()
```

### Código Gerado

```
N = int(input())
M = [list(map(int, input().split())) for _
    in range(N)]

for l in range(N):
    for c in range(N):
        if M[l][c] > 100:
            M[l][c] = '-'
        elif M[l][c] > 90:
            M[l][c] = 'o'
        else:
            M[l][c] = '+'

a, b, c = [0, 0, 0]
for l in range(N):
    print(' '.join(M[l]), sep='')
    a += M[l].count('+')
    b += M
```

**APÊNDICE F – EXEMPLOS DE RESULTADOS DA  
AVALIAÇÃO DO CLAUDE LAAJ**

Código 2: Saída do Experimento 3.1 - Cenário S<sub>10</sub>**Ground-truth**

```
def main():
    N = int(input())
    matriz = []
    count_1 = count_2 = count_3 = 0
    for i in range(N):
        matriz.append([0] * N)
    for i in range(N):
        matriz[i] = input().split()
        matriz[i] = list(map(int,
            matriz[i]))
    for i in range(N):
        for j in range(N):
            if matriz[i][j] <= 90:
                matriz[i][j] = '+'
                count_1 += 1
            elif matriz[i][j] > 90 and
                matriz[i][j] <= 100:
                matriz[i][j] = 'o'
                count_2 += 1
            else:
                matriz[i][j] = '-'
                count_3 += 1
    for i in range(N):
        for j in range(N):
            print(matriz[i][j], end='')
        print()
    print()
    print(f'+: {count_1}')
    print(f'o: {count_2}')
    print(f'-: {count_3}')

if __name__ == "__main__":
    main()
```

**Código Gerado**

```
n = int(input()) # Le o tamanho da matriz
                N x N
matrix = [list(map(int, input().split()))
            for _ in range(n)] # Le a matriz de
                estimativas

sempre = 0 # Contador para celulas sempre
                acessiveis
somente = 0 # Contador para celulas
                acessiveis somente quando K-Oru nao
                esta
nunca = 0 # Contador para celulas nunca
                acessiveis
simbolos = [] # Matriz para armazenar os
                simbolos de saida

for linha in matrix: # Processa cada linha
                da matriz
    nova_linha = []
    for valor in linha:
        if valor > 100: # Nunca acessivel
                (ja ultrapassa sem K-Oru)
            nova_linha.append('-')
            nunca += 1
        elif valor > 90: # Acessivel
                somente quando K-Oru nao esta
                (+10 = 101)
            nova_linha.append('o')
            somente += 1
        else: # Sempre acessivel
                (K-Oru nao faz passar de 100)
            nova_linha.append('+')
            sempre += 1
    simbolos.append(nova_linha)

# Imprime a matriz transformada
for linha in simbolos:
    print(''.join(linha)) # Junta os
                simbolos sem espacos
print()

# Imprime as contagens conforme
                especificado
print(f'+: {sempre}')
print(f'o: {somente}')
print(f'-: {nunca}')
```

**Prompt 12: Resultado Final da avaliação do Claude LaaJ**

```
{
  "response": " n = int(input()) # Le o tamanho da matriz N x N matrix = [list(map(int,
  ↪ input().split())) for _ in range(n)] # Le a matriz de estimativas sempre = 0
  ↪ # Contador para celulas sempre acessiveis somente = 0 # Contador para celulas
  ↪ acessiveis somente quando K-Oru nao esta nunca = 0 # Contador para celulas
  ↪ nunca acessiveis simbolos = [] # Matriz para armazenar os simbolos de saida
  ↪ for linha in matrix: # Processa cada linha da matriz nova_linha = [] for
  ↪ valor in linha: if valor > 100: # Nunca acessivel (ja ultrapassa sem K-Oru)
  ↪ nova_linha.append('-') nunca += 1 elif valor > 90: # Acessivel somente quando
  ↪ K-Oru nao esta (+10 = 101) nova_linha.append('o') somente += 1
  ↪ else: # Sempre acessivel (K-Oru nao faz passar de 100)
  ↪ nova_linha.append('+') sempre += 1 simbolos.append(nova_linha) #
  ↪ Imprime a matriz transformada for linha in simbolos: print(''.join(linha)) #
  ↪ Junta os simbolos sem espacos print() # Imprime as contagens conforme
  ↪ especificado print(f'+: {sempre}') print(f'o: {somente}') print(f'-:
  ↪ {nunca}') ",
  "metric_laaj": " [ {  \"inconsistency\": \"None\", \"severity\": \"Negligible\" },
  ↪ {  \"final_answer\": 0 } ] ",
  "final_answer": 0
}
```

**Prompt 13: Resultados Parciais da avaliação do Claude LaaJ**

```
[
  {
    "response": " # Le o tamanho da matriz quadrada N = int(input()) # Numero de
    ↪ linhas/colunas da matriz # Le a matriz de estimativa de goblins (sem
    ↪ K-Oru, que vale +10) matriz = [list(map(int, input().split())) for _ in
    ↪ range(N)] # Cria matriz NxN # Inicializa contadores para cada tipo de
    ↪ celula sempre_possivel = 0 # Celulas onde sempre pode entrar (<90)
    ↪ somente_quando = 0 # Celulas que so pode entrar sem K-Oru (90-100)
    ↪ nunca_possivel = 0 # Celulas onde nunca pode entrar (>100) # K-Oru segue
    ↪ um padrao ciclico: linha (t//N)%N, coluna t%N onde t e o tempo # Para
    ↪ nosso problema, sua posicao atual nao importa para a classificacao inicial
    ↪ # Gera a matriz simplificada e conta celulas acessiveis simplificada = []
    ↪ # Matriz com representacao simplificada for linha in range(N): nova_linha
    ↪ = [] for coluna in range(N): goblins = matriz[linha][coluna] if goblins >
    ↪ 100: # Nunca e possivel entrar (mesmo sem K-Oru) nova_linha.append('-')
    ↪ nunca_possivel += 1 elif goblins <= 90: # Sempre possivel entrar (even
    ↪ with K-Oru +10) nova_linha.append('+') sempre_possivel += 1 else: # So
    ↪ possivel quando K-Oru nao esta (90 < x 100) nova_linha.append('o')
    ↪ somente_quando += 1 simplificada.append(nova_linha) # Imprime a
    ↪ matriz simplificada for linha in simplificada: print(''.join(linha)) #
    ↪ Imprime linha sem espacos # Imprime contagem de cada tipo print(f'\
    ↪ +:{sempre_possivel}') # Sempre possivel print(f'o:{somente_quando}') #
    ↪ Somente quando K-Oru nao esta print(f'-:{nunca_possivel}') # Nunca
    ↪ possivel ",
    "metric_laaj": " [ {  \"inconsistency\": \"The code snippet has a logical error
    ↪ in the output format. The reference solution prints the counts with a
    ↪ space after the colon (e.g., '+: 1'), but the code snippet prints without
    ↪ a space (e.g., '+:1'). This leads to incorrect output format.\",
    ↪ \"severity\": \"Major\" }, { \"inconsistency\": \"The line by line
    ↪ explanation contains an incorrect comment on line 19 that says 'even with
```

```

    ↪ K-Oru +10' when it should clarify that cells with 90 goblins can
    ↪ accommodate K-Oru's additional 10 goblins and still stay under the 100
    ↪ limit.\", \"severity\": \"Minor\" }, { \"inconsistency\": \"The
    ↪ explanation does not adequately describe the logic behind the threshold
    ↪ values (90, 100) and why these specific boundaries were chosen in relation
    ↪ to K-Oru's value of 10 goblins.\", \"severity\": \"Moderate\" }, {
    ↪ \"final_answer\": -3 } ] ",
"final_answer": -3
},
{
"response": " # Read the size of the matrix (N x N) N = int(input()) # First
    ↪ input is the matrix dimension (1 <= N <= 50) # Read the matrix of goblin
    ↪ counts (without K-Oru's 10 goblins) matrix = [list(map(int,
    ↪ input().split())) for _ in range(N)] # Read N lines of space-separated
    ↪ integers # Initialize counters for each accessibility type always_possible
    ↪ = 0 # When goblins <= 100 even with K-Oru (current + 10 <= 100)
    ↪ sometimes_possible = 0 # When goblins <= 100 only without K-Oru (current
    ↪ <= 100 < current + 10) never_possible = 0 # When goblins > 100 even
    ↪ without K-Oru (current > 100) # Analyze each cell in the matrix for row in
    ↪ range(N): # Iterate through each row for col in range(N): # Iterate
    ↪ through each column goblins = matrix[row][col] # Current goblin count
    ↪ (without K-Oru) if goblins > 100: # Case 1: Never accessible (too many
    ↪ goblins) never_possible += 1 elif goblins + 10 <= 100: # Case 2: Always
    ↪ accessible (K-Oru doesn't push over 100) always_possible += 1 else:
    ↪ # Case 3: Only accessible when K-Oru isn't present (90 <
    ↪ goblins <= 100) sometimes_possible += 1 # Output the results (counts
    ↪ of each accessibility type) print(f\"sempre: {always_possible}\") # Cells
    ↪ always accessible print(f\"somete-quando: {sometimes_possible}\") # Cells
    ↪ accessible only when K-Oru isn't present print(f\"nunca:
    ↪ {never_possible}\") # Cells never accessible ",
"metric_laaj": " [ { \"inconsistency\": \"Logical errors leading to incorrect
    ↪ outputs\", \"severity\": \"Major\" }, { \"inconsistency\": \"Determines
    ↪ whether the explanation correctly describes what the code actually does\",
    ↪ \"severity\": \"Fatal\" }, { \"final_answer\": -7 } ] **Explanation:** 1.
    ↪ **Logical Error (Major -3):** The code snippet produces completely
    ↪ different output from what the problem requires. The reference solution
    ↪ outputs a matrix with symbols (`+`, `o`, `-`) representing accessibility
    ↪ status for each cell, followed by counts. The code snippet only outputs
    ↪ three count lines with different labels (`sempre`, `somete-quando`,
    ↪ `nunca`) and does not output the matrix visualization at all. This is a
    ↪ fundamental mismatch in the required output format. 2. **Explanation
    ↪ Misalignment (Fatal -4):** The line-by-line explanation describes the code
    ↪ as outputting counts with labels like \"sempre: {always_possible}\",
    ↪ \"somete-quando: {sometimes_possible}\", and \"nunca: {never_possible}\".
    ↪ However, the problem statement and reference solution clearly show that
    ↪ the output should be a matrix of symbols followed by counts with labels
    ↪ `+`, `o`, and `-.`. The explanation fails to identify that the code is
    ↪ missing the critical matrix output functionality entirely, making it
    ↪ fundamentally incorrect in describing what the code should do versus what
    ↪ it actually does. **Final Answer:** -7 (Major + Fatal = -3 + -4 = -7)",
"final_answer": -7
},
{
"response": " n = int(input()) # Le o tamanho da matriz N x N matrix =
    ↪ [list(map(int, input().split())) for _ in range(n)] # Le a matriz de

```

```
↪ estimativas sempre = 0 # Contador para celulas sempre acessiveis somente =
↪ 0 # Contador para celulas acessiveis somente quando K-Oru nao esta nunca =
↪ 0 # Contador para celulas nunca acessiveis simbolos = [] # Matriz para
↪ armazenar os simbolos de saida for linha in matrix: # Processa cada linha
↪ da matriz nova_linha = [] for valor in linha: if valor > 100: # Nunca
↪ acessivel (ja ultrapassa sem K-Oru) nova_linha.append('-') nunca += 1 elif
↪ valor > 90: # Acessivel somente quando K-Oru nao esta (+10 = 101)
↪ nova_linha.append('o')           somente += 1           else:           # Sempre
↪ acessivel (K-Oru nao faz passar de 100)           nova_linha.append('+')
↪           sempre += 1           simbolos.append(nova_linha) # Imprime a matriz
↪ transformada for linha in simbolos: print(''.join(linha)) # Junta os
↪ simbolos sem espacos print() # Imprime as contagens conforme especificado
↪ print(f'+: {sempre}') print(f'o: {somente}') print(f'-: {nunca}') ",
"metric_laaj": " [ {      \"inconsistency\": \"None\", \"severity\":
↪ \"Negligible\" }, {      \"final_answer\": 0 } ] ",
"final_answer": 0
}
]
```

**APÊNDICE G – EXEMPLOS DE RESULTADOS DA  
AVALIAÇÃO DO GEMINI LAAJ**

Código 3: Saída do Experimento 3.2 - Cenário S<sub>8</sub>**Ground-truth**

```
def main():
    N = int(input())
    matriz = []
    count_1 = count_2 = count_3 = 0
    for i in range(N):
        matriz.append([0] * N)
    for i in range(N):
        matriz[i] = input().split()
        matriz[i] = list(map(int,
            matriz[i]))
    for i in range(N):
        for j in range(N):
            if matriz[i][j] <= 90:
                matriz[i][j] = '+'
                count_1 += 1
            elif matriz[i][j] > 90 and
                matriz[i][j] <= 100:
                matriz[i][j] = 'o'
                count_2 += 1
            else:
                matriz[i][j] = '-'
                count_3 += 1
    for i in range(N):
        for j in range(N):
            print(matriz[i][j], end='')
        print()
    print()
    print(f'+: {count_1}')
    print(f'o: {count_2}')
    print(f'-: {count_3}')

if __name__ == "__main__":
    main()
```

**Código Gerado**

```
N = int(input()) # Le o tamanho da matriz
                 # quadrada (N x N)
matriz = [list(map(int, input().split()))
           for _ in range(N)] # Le as
                             # estimativas de goblins
sempre_seguro = as_vezes_seguro =
nunca_seguro = 0 # Contadores para
                 # cada categoria de celulas
for i in range(N): # Percorre cada linha
da matriz
    for j in range(N): # Percorre cada
coluna
        goblins = matriz[i][j] # Numero
                                # base de goblins na celula
        # Analisa as possibilidades
        # considerando K-Oru (que
        # adiciona +10 goblins quando
        # presente)
        if goblins > 100: # Nunca seguro
(com ou sem K-Oru)
            simbolo = '-' # Pois mesmo sem
K-Oru ja ultrapassa 100
            nunca_seguro += 1
        elif goblins + 10 > 100: # Seguro
apenas sem K-Oru (91-100 com
K-Oru)
            simbolo = 'o' # Pois com K-Oru
ficaria >100 (inseguro)
            as_vezes_seguro += 1
        else: # Sempre seguro (90 sem
K-Oru, 100 com)
            simbolo = '+' # Pois mesmo com
+10 (K-Oru) fica 100
            sempre_seguro += 1
        print(simbolo, end='') # Imprime o
simbolo sem quebra de linha
    print() # Quebra linha apos cada linha
da matriz

# Imprime os resultados totais
print(f"\n+: {sempre_seguro}") # Celulas
sempre seguras
print(f"o: {as_vezes_seguro}") # Celulas
seguras apenas sem K-Oru
print(f"-: {nunca_seguro}") # Celulas
nunca seguras
```

## Prompt 14: Resultado Final da avaliação do Gemini LaaJ

```

{
"response": " N = int(input())                                # Le o tamanho da matriz
↳ quadrada (N x N) matriz = [list(map(int, input().split())) for _ in range(N)]
↳ # Le as estimativas de goblins sempre_seguro = as_vezes_seguro = nunca_seguro
↳ = 0 # Contadores para cada categoria de celulas for i in range(N): #
↳ Percorre cada linha da matriz for j in range(N): # Percorre
↳ cada coluna goblins = matriz[i][j] # Numero base de goblins na
↳ celula # Analisa as possibilidades considerando K-Oru (que
↳ adiciona +10 goblins quando presente) if goblins > 100: # Nunca
↳ seguro (com ou sem K-Oru) simbolo = '-' # Pois mesmo
↳ sem K-Oru ja ultrapassa 100 nunca_seguro += 1 elif goblins + 10 >
↳ 100: # Seguro apenas sem K-Oru (91-100 com K-Oru) simbolo = 'o'
↳ # Pois com K-Oru ficaria >100 (inseguro)
↳ as_vezes_seguro += 1 else: # Sempre seguro
↳ (90 sem K-Oru, 100 com) simbolo = '+' # Pois
↳ mesmo com +10 (K-Oru) fica 100 sempre_seguro += 1
↳ print(simbolo, end='') # Imprime o simbolo sem quebra de linha
↳ print() # Quebra linha apos cada linha da matriz
↳ # Imprime os resultados totais print(f"\n +: {sempre_seguro}") # Celulas
↳ sempre seguras print(f"\no: {as_vezes_seguro}") # Celulas seguras
↳ apenas sem K-Oru print(f"\n -: {nunca_seguro}") # Celulas nunca
↳ seguras ",
"metric_laaj": " [ { \inconsistency\": \The problem statement implies that
↳ K-Oru adds 10 goblins, making a total of `goblins + 10`. However, the code
↳ interprets this as a cell being safe if `goblins <= 90` (always safe) or
↳ `goblins > 90 and goblins <= 100` (safe only if K-Oru is not present). The
↳ problem asks to determine if it's possible to enter a cell, meaning the total
↳ number of goblins should not exceed 100. The reference solution also uses a
↳ similar logic of categorizing cells based on goblin counts. The code's logic
↳ for `goblins + 10 > 100` where `goblins` is between 91 and 100 is correct in
↳ identifying these as 'o' cells (safe only without K-Oru). The logic for
↳ `goblins <= 90` being '+' (always safe) is also correct, as even with K-Oru,
↳ the total would be at most 100. The logic for `goblins > 100` being '-'
↳ (never safe) is also correct. The inconsistency arises from the
↳ interpretation of the problem's outcome. The problem asks for cells where
↳ it's possible to enter. This means the total number of goblins (base + K-Oru,
↳ if present) must be <= 100. The reference solution's logic of `goblins <= 90`
↳ being '+' and `goblins > 90 and goblins <= 100` being 'o' seems to align more
↳ with the problem's implied conditions for safety. The code snippet's
↳ conditions are: `goblins > 100` (never safe), `goblins + 10 > 100` (safe only
↳ without K-Oru), and `else` (always safe). This logic correctly translates to
↳ the problem statement's requirements.\", \severity\": \Negligible\" }, {
↳ \inconsistency\": \The code snippet does not explicitly define the behavior
↳ for edge cases where N might be outside the specified range of 1 to 50. While
↳ the problem statement guarantees N is within this range, robust code would
↳ typically include checks for such inputs. However, given the constraints,
↳ this is a minor oversight.\", \severity\": \Small\" }, { \inconsistency\":
↳ \The problem statement implies K-Oru adds 10 goblins to the base count. The
↳ code correctly implements the logic to check if `goblins + 10` exceeds 100.
↳ The problem asks for three categories: always possible to enter, never
↳ possible to enter, or only possible when K-Oru is not present. The code
↳ translates these into symbols: '+' for always safe, 'o' for safe only without
↳ K-Oru, and '-' for never safe. The logic within the conditional statements
↳ correctly determines these categories. For example, if `goblins` is 91,

```

↪ ``goblins + 10`` is 101, so it's not always safe and not never safe, hence 'o'.  
 ↪ If ``goblins`` is 90, ``goblins + 10`` is 100, so it's always safe ('+'). If  
 ↪ ``goblins`` is 101, it's never safe ('-'). This logic appears sound based on  
 ↪ the problem description.\", \"severity\": \"Negligible\" }, {  
 ↪ \"inconsistency\": \"The code snippet implements the logic to categorize  
 ↪ cells based on the number of goblins and the presence of K-Oru. It correctly  
 ↪ identifies cells that are always safe ('+'), sometimes safe ('o'), and never  
 ↪ safe ('-'). The counting of these categories is also accurate. The output  
 ↪ format matches the example in the problem statement. Therefore, the code  
 ↪ snippet correctly addresses the problem's requirements.\", \"severity\":  
 ↪ \"Negligible\" }, { \"inconsistency\": \"The line-by-line explanation  
 ↪ correctly describes the functionality of each line of the code snippet. It  
 ↪ accurately explains how the input is read, how the matrix is processed, and  
 ↪ how the conditions for categorizing cells are evaluated. The explanation  
 ↪ covers the logic for assigning symbols ('+', 'o', '-') and incrementing the  
 ↪ respective counters. The final output of the matrix and the counts is also  
 ↪ explained. No major discrepancies between the code and its explanation were  
 ↪ found.\", \"severity\": \"Negligible\" }, { \"inconsistency\": \"The code  
 ↪ snippet is a correct implementation of the problem statement. It correctly  
 ↪ reads the input, processes the matrix according to the specified conditions  
 ↪ for safety (considering K-Oru's presence and the 100 goblin limit), and  
 ↪ categorizes each cell. The counters for each category are also correctly  
 ↪ updated. The output format is as required. No major logical errors or  
 ↪ inefficiencies were identified.\", \"severity\": \"Negligible\" }, {  
 ↪ \"inconsistency\": \"The problem statement asks for a determination of  
 ↪ whether it's possible to enter a cell. This implies checking if the \*total\*  
 ↪ number of goblins (base + K-Oru, if present) is less than or equal to 100.  
 ↪ The code snippet correctly implements this by checking ``goblins > 100`` (never  
 ↪ safe), ``goblins + 10 > 100`` (safe only without K-Oru), and the ``else`` case  
 ↪ (always safe). The interpretation of the conditions and the resulting symbols  
 ↪ ('+', 'o', '-') accurately reflect the problem's requirements. The reference  
 ↪ solution uses a similar logic of categorizing cells based on goblin counts  
 ↪ relative to the 100-goblin limit, also considering K-Oru's potential  
 ↪ addition.\", \"severity\": \"Negligible\" }, { \"inconsistency\": \"The code  
 ↪ snippet correctly processes all the cells in the N x N matrix and calculates  
 ↪ the counts for each category. The logic for determining the category of each  
 ↪ cell is based on the number of goblins and the condition of K-Oru's presence,  
 ↪ which adds 10 goblins. The code correctly identifies cells that are always  
 ↪ safe ('+'), only safe when K-Oru is absent ('o'), and never safe ('-'). The  
 ↪ total counts for each category are accumulated accurately. The output format  
 ↪ also matches the problem requirements.\", \"severity\": \"Negligible\" }, {  
 ↪ \"inconsistency\": \"The code snippet doesn't include any exception handling  
 ↪ for invalid input, such as non-integer values for N or matrix elements. While  
 ↪ the problem statement provides constraints, in a real-world scenario, this  
 ↪ would be a necessary addition for robustness.\", \"severity\": \"Small\" }, {  
 ↪ \"inconsistency\": \"The code snippet correctly implements all the required  
 ↪ functionalities. It reads the input, processes the matrix based on the goblin  
 ↪ counts and K-Oru's impact, categorizes each cell, counts the cells in each  
 ↪ category, and prints the results in the specified format. There are no  
 ↪ missing functionalities or placeholders.\", \"severity\": \"Negligible\" }, {  
 ↪ \"inconsistency\": \"The problem statement defines the conditions for  
 ↪ entering a cell: 'nao pode haver mais de 100 goblins naquela celula.' and  
 ↪ that K-Oru 'equivale a 10 goblins'. The code snippet correctly interprets  
 ↪ this. A cell is always safe ('+') if ``goblins <= 90``, because even with K-Oru  
 ↪ (``goblins + 10``), the total is at most 100. A cell is safe only when K-Oru is

```

↪ absent ('o') if `goblins > 90` and `goblins + 10 > 100` (meaning `goblins` is
↪ between 91 and 100 inclusive). A cell is never safe ('-') if `goblins > 100`,
↪ as even without K-Oru, the limit is exceeded. This logic is sound and
↪ directly addresses the problem's safety conditions.\", \"severity\":
↪ \"Negligible\" }, { \"inconsistency\": \"The line-by-line explanation covers
↪ all the essential parts of the code snippet, including variable
↪ initializations, input reading, matrix traversal, conditional logic for cell
↪ categorization, and output printing. It provides a clear understanding of how
↪ the code works.\", \"severity\": \"Negligible\" }, { \"inconsistency\": \"The
↪ explanation correctly describes what the code actually does. It accurately
↪ reflects the logic for determining cell safety based on goblin counts and
↪ K-Oru's presence. The symbols assigned ('+', 'o', '-') and the counting
↪ mechanism are all explained in alignment with the code's behavior.\",
↪ \"severity\": \"Negligible\" }, { \"inconsistency\": \"The explanation
↪ accurately covers all relevant parts of the snippet. It describes the
↪ initialization of counters, the nested loops for matrix traversal, the
↪ `map(int, input().split())` for reading input, the conditional statements
↪ that define the cell categories, and the `print()` statements for output. All
↪ significant logic and variables are addressed.\", \"severity\":
↪ \"Negligible\" }, { \"inconsistency\": \"The explanation is generally
↪ understandable. It uses clear language to describe the code's functionality.
↪ However, it could benefit from slightly more detail on *why* certain
↪ conditions are chosen (e.g., why `goblins <= 90` implies always safe). For
↪ instance, explicitly stating that `90 + 10 = 100` clarifies the 'always safe'
↪ condition.\", \"severity\": \"Minor\" }, { \"inconsistency\": \"The
↪ explanation is well-organized, following the flow of the code from input to
↪ output. It uses bullet points and clear sentence structures, making it easy
↪ to follow.\", \"severity\": \"Negligible\" }, { \"inconsistency\": \"The
↪ explanation does not explicitly relate the snippet to its broader purpose or
↪ why certain design decisions were made. For example, it doesn't discuss the
↪ trade-offs of this categorization method or its role in a larger system (if
↪ applicable). The focus is primarily on explaining *what* the code does,
↪ rather than *why*.\", \"severity\": \"Moderate\" }, { \"inconsistency\":
↪ \"The explanation does not acknowledge potential limitations, such as the
↪ lack of error handling for invalid inputs, which was noted as a minor code
↪ issue. It also doesn't discuss how the code might behave with inputs outside
↪ the specified constraints (though the problem statement limits these).\",
↪ \"severity\": \"Minor\" }, { \"inconsistency\": \"The explanation has minor
↪ grammatical issues and could be more fluent in places. For example, some
↪ sentences are a bit long or could be rephrased for better clarity.\",
↪ \"severity\": \"Minor\" }, { \"inconsistency\": \"The explanation aligns well
↪ with the code's inline comments and naming conventions. The tone is
↪ consistent with technical documentation. The explanation is straightforward
↪ and avoids overly technical jargon where unnecessary.\", \"severity\":
↪ \"Negligible\" }, { \"inconsistency\": \"The explanation is reasonably
↪ accessible for a junior developer. It explains the core logic clearly.
↪ However, it could be enhanced by elaborating on the reasoning behind the
↪ conditional logic to improve understanding for learners.\", \"severity\":
↪ \"Moderate\" }, { \"inconsistency\": \"The code correctly implements the
↪ logic to determine cell safety based on the number of goblins and the
↪ potential addition of K-Oru (+10 goblins), with a maximum of 100 goblins
↪ allowed for entry. The categorization into '+', 'o', and '-' symbols, along
↪ with the counting, is accurate. The output format also matches the problem's
↪ requirements. No logical errors were found.\", \"severity\": \"Negligible\"
↪ }, { \"final_answer\": -2 } ] \",

```

```
"final_answer": -2
}
```

### Prompt 15: Resultados Parciais da avaliação do Gemini LaaJ

```
[
{
  "response": " N = int(input()) # Le o tamanho N da matriz
  ↪ quadrada matrix = [] # Armazenara a matriz de
  ↪ estimativas for _ in range(N): # Para cada linha da
  ↪ matriz row = list(map(int, input().split())) # Le uma linha de inteiros
  ↪ separados por espaco matrix.append(row) # Adiciona a
  ↪ matriz results = [] # Armazenara os resultados
  ↪ ('+', 'o' ou '-') count_plus = count_o = count_minus = 0 # Contadores para
  ↪ cada tipo de celula for i in range(N): # Percorre cada
  ↪ linha da matriz result_row = [] # Armazena
  ↪ resultados da linha atual for j in range(N): # Percorre
  ↪ cada coluna goblins = matrix[i][j] # Goblins na celula sem
  ↪ K-Oru if goblins > 100: # Nunca seguro (K-Oru nao
  ↪ afeta) result = '-' count_minus += 1 elif goblins +
  ↪ 10 > 100: # Seguro apenas se K-Oru nao estiver result =
  ↪ 'o' count_o += 1 else: #
  ↪ Sempre seguro (com ou sem K-Oru) result = '+' count_plus
  ↪ += 1 result_row.append(result) # Adiciona resultado a linha
  ↪ results.append(result_row) # Adiciona linha aos resultados for
  ↪ row in results: # Imprime a matriz de resultados
  ↪ print(' '.join(row)) # Espacos entre simbolos conforme
  ↪ especificado print() # Linha em branco de
  ↪ separacao print(f'+: {count_plus}') # Total de celulas sempre
  ↪ seguras print(f'o: {count_o}') # Total de celulas
  ↪ condicionalmente seguras print(f'-: {count_minus}') # Total de
  ↪ celulas nunca seguras ",
  "metric_laaj": " [ { \"inconsistency\": \"The code snippet does not correctly
  ↪ implement the logic for determining cell accessibility based on the
  ↪ problem statement. The problem states that K-Oru is equivalent to 10
  ↪ goblins and should be considered when determining accessibility. The code
  ↪ snippet only checks if `goblins + 10 > 100` for the 'o' case, which
  ↪ implies that the base number of goblins is less than or equal to 90.
  ↪ However, the problem statement implies that K-Oru's presence makes a cell
  ↪ inaccessible if the total (base goblins + 10) exceeds 100. Therefore,
  ↪ cells with 91-100 goblins (without K-Oru) should be marked as 'o' (only
  ↪ accessible when K-Oru is not present). Cells with 101+ goblins should be
  ↪ marked as '-' (never accessible). Cells with <= 90 goblins should be
  ↪ marked as '+' (always accessible). The current logic `if goblins > 100:
  ↪ result = '-'` and `elif goblins + 10 > 100: result = 'o'` incorrectly
  ↪ categorizes cells. For example, a cell with 95 goblins would be marked
  ↪ 'o', but `95 + 10 = 105` which is greater than 100, so it should be '-'. A
  ↪ cell with 90 goblins would be marked '+', but `90 + 10 = 100`, which is
  ↪ not greater than 100, so it should be '+'. The logic is reversed for the
  ↪ 'o' and '-' conditions.\", \"severity\": \"Major\" }, { \"inconsistency\":
  ↪ \"The code snippet fails to implement the specific logic for determining
  ↪ the three types of cells ('+', 'o', '-') as described in the problem
  ↪ statement and implied by the reference solution. The problem states: 'Para
  ↪ cada celula, sera definido se sempre e possivel entrar la, se nunca e
```

↪ possível, ou se só é possível entrar quando K-Oru não estiver presente na  
 ↪ célula.' and K-Oru is '10 goblins'. The conditions in the code snippet are  
 ↪ not aligned with this. Specifically, the condition ``if goblins > 100:``  
 ↪ correctly identifies cells that are never safe ('-'). However, the ``elif`  
 ↪ `goblins + 10 > 100:`` condition, which is supposed to represent cells only  
 ↪ safe when K-Oru is not present ('o'), is flawed. It should be that if the  
 ↪ base goblin count is such that adding K-Oru's 10 goblins makes the total  
 ↪ exceed 100, then it's only safe if K-Oru isn't there. This implies that if  
 ↪ ``goblins <= 100`` but ``goblins + 10 > 100``, then it's 'o'. The current  
 ↪ code's ``goblins + 10 > 100`` condition is too broad and doesn't account for  
 ↪ the ``goblins <= 100`` constraint correctly. A cell with 101 goblins should  
 ↪ be '-', but ``101 + 10 > 100`` is true, and the code would incorrectly  
 ↪ assign it 'o' if the ``goblins > 100`` check wasn't there. Crucially, the  
 ↪ problem implies a range for 'o' cells: where the base goblin count is such  
 ↪ that adding K-Oru exceeds 100, but the base count itself does not. The  
 ↪ code's ``else: result = '+'`` condition is also problematic. It assigns '+'  
 ↪ if ``goblins + 10 <= 100``. This means any cell with 90 or fewer goblins is  
 ↪ always safe, which is correct. But it doesn't cover the boundary correctly  
 ↪ in relation to the 'o' condition.\", `\"severity\": \"Major\"` }, {  
 ↪ `\"inconsistency\": \"The line-by-line explanation is missing entirely.`  
 ↪ `Therefore, it cannot be evaluated against the code snippet.\"`,  
 ↪ `\"severity\": \"Fatal\"` }, { `\"inconsistency\": \"The code snippet does`  
 ↪ `not output the results in the format specified by the problem statement.`  
 ↪ `The problem statement does not explicitly state the output format for the`  
 ↪ `matrix, but the reference solution prints each row with elements separated`  
 ↪ `by no space, and then a blank line, followed by the counts. The code`  
 ↪ `snippet prints each row with elements separated by spaces (`print('  
 ↪ `.join(row))`).\", \"severity\": \"Small\" }, { \"inconsistency\": \"The  
 ↪ code snippet does not count the total number of cells that are always  
 ↪ possible to enter, never possible to enter, or only possible when K-Oru is  
 ↪ not present. It only counts the number of '+' and 'o' and '-' cells, which  
 ↪ is implied to be what is needed but the problem states: 'voce tambem deve  
 ↪ contar em quantas celulas e possivel entrar.' This implies a single total  
 ↪ count, not separate counts for each category, though the reference  
 ↪ solution does separate them. Given the reference solution's output, it  
 ↪ seems the intention was to count each category. However, the wording is  
 ↪ slightly ambiguous.\", \"severity\": \"Small\" }, { \"inconsistency\":  
 ↪ \"The code snippet does not implement the problem's requirement to define  
 ↪ cells as 'always possible', 'never possible', or 'only possible when K-Oru  
 ↪ is not present'. Instead, it directly assigns symbols ('+', 'o', '-')  
 ↪ based on the goblin count, which is an interpretation of these categories.  
 ↪ The core logic for determining these categories is flawed as described in  
 ↪ other inconsistencies.\", \"severity\": \"Major\" }, { \"inconsistency\":  
 ↪ \"The code snippet does not handle the constraints on N ( $1 \leq N \leq 50$ ) and  
 ↪ goblin counts ( $1 < goblins < 300$ ). While Python's integer types can handle  
 ↪ these, there's no explicit validation or handling for values outside these  
 ↪ ranges, which could be considered a corner case, though the prompt assumes  
 ↪ valid inputs.\", \"severity\": \"Small\" }, { \"inconsistency\": \"The  
 ↪ code does not address K-Oru's movement pattern or his cyclic presence in  
 ↪ cells, other than his value of 10 goblins. The problem states 'K-Oru, que  
 ↪ voce enfrentou na primeira cidade, movia-se por todas as celulas da matriz  
 ↪ seguindo um padrao ciclico. Sendo assim, voce o desconsiderou nas suas  
 ↪ estimativas (mas tenha em mente que ele equivale a 10 goblins), mas como e  
 ↪ um padrao muito simples, sempre da pra saber em qual celula ele esta.' The  
 ↪ code only uses his goblin value and doesn't consider any cyclic movement`

```

↪ or presence. This might be an over-interpretation of the problem, as the
↪ core task is the classification of cells based on K-Oru's *potential*
↪ presence.\", \"severity\": \"Small\" }, { \"inconsistency\": \"The
↪ explanation is completely missing, so its understandability, readability,
↪ use of programming terms, organization, relation to broader purpose,
↪ acknowledgment of limitations, spelling, grammar, fluency, and alignment
↪ with code comments/tone cannot be evaluated.\", \"severity\": \"Fatal\" },
↪ { \"inconsistency\": \"The explanation is completely missing, so its
↪ correctness in describing what the code does cannot be evaluated.\",
↪ \"severity\": \"Fatal\" }, { \"inconsistency\": \"The explanation is
↪ completely missing, so its coverage of all relevant parts of the snippet
↪ cannot be evaluated.\", \"severity\": \"Fatal\" }, { \"inconsistency\":
↪ \"The explanation is completely missing, so its relation to the code's
↪ purpose, design decisions, limitations, or error handling cannot be
↪ evaluated.\", \"severity\": \"Fatal\" }, { \"inconsistency\": \"The
↪ explanation is completely missing, so its use of programming terms and
↪ concepts cannot be evaluated.\", \"severity\": \"Fatal\" }, {
↪ \"inconsistency\": \"The explanation is completely missing, so its
↪ organization cannot be evaluated.\", \"severity\": \"Fatal\" }, {
↪ \"inconsistency\": \"The explanation is completely missing, so its ability
↪ to help others learn cannot be evaluated.\", \"severity\": \"Fatal\" }, {
↪ \"inconsistency\": \"The explanation is completely missing, so its
↪ spelling, grammar, and sentence fluency cannot be evaluated.\",
↪ \"severity\": \"Fatal\" }, { \"inconsistency\": \"The explanation is
↪ completely missing, so its alignment with the code's inline comments,
↪ naming conventions, and overall tone cannot be evaluated.\", \"severity\":
↪ \"Fatal\" }, { \"final_answer\": -30 } ] ",
"final_answer": -30
},
{
"response": " N = int(input())                # Le o tamanho da matriz
↪ quadrada (N x N) matriz = [list(map(int, input().split())) for _ in
↪ range(N)] # Le as estimativas de goblins sempre_seguro = as_vezes_seguro =
↪ nunca_seguro = 0 # Contadores para cada categoria de celulas for i in
↪ range(N): # Percorre cada linha da matriz for j in range(N):
↪ # Percorre cada coluna goblins = matriz[i][j]
↪ # Numero base de goblins na celula # Analisa as
↪ possibilidades considerando K-Oru (que adiciona +10 goblins quando
↪ presente) if goblins > 100: # Nunca seguro (com ou sem
↪ K-Oru) simbolo = '-' # Pois mesmo sem K-Oru ja
↪ ultrapassa 100 nunca_seguro += 1 elif goblins + 10 > 100:
↪ # Seguro apenas sem K-Oru (91-100 com K-Oru) simbolo = 'o'
↪ # Pois com K-Oru ficaria >100 (inseguro)
↪ as_vezes_seguro += 1 else: # Sempre
↪ seguro (90 sem K-Oru, 100 com) simbolo = '+' #
↪ Pois mesmo com +10 (K-Oru) fica 100 sempre_seguro += 1
↪ print(simbolo, end='') # Imprime o simbolo sem
↪ quebra de linha print() # Quebra linha apos
↪ cada linha da matriz # Imprime os resultados totais print(f"\n +:
↪ {sempre_seguro}\n) # Celulas sempre seguras print(f"\n o:
↪ {as_vezes_seguro}\n) # Celulas seguras apenas sem K-Oru print(f"\n -:
↪ {nunca_seguro}\n) # Celulas nunca seguras ",
"metric_laaj": " [ { \"inconsistency\": \"The problem statement implies that
↪ K-Oru adds 10 goblins, making a total of `goblins + 10`. However, the code
↪ interprets this as a cell being safe if `goblins <= 90` (always safe) or

```

```

↪ `goblins > 90 and goblins <= 100` (safe only if K-Oru is not present). The
↪ problem asks to determine if it's possible to enter a cell, meaning the
↪ total number of goblins should not exceed 100. The reference solution also
↪ uses a similar logic of categorizing cells based on goblin counts. The
↪ code's logic for `goblins + 10 > 100` where `goblins` is between 91 and
↪ 100 is correct in identifying these as 'o' cells (safe only without
↪ K-Oru). The logic for `goblins <= 90` being '+' (always safe) is also
↪ correct, as even with K-Oru, the total would be at most 100. The logic for
↪ `goblins > 100` being '-' (never safe) is also correct. The inconsistency
↪ arises from the interpretation of the problem's outcome. The problem asks
↪ for cells where it's possible to enter. This means the total number of
↪ goblins (base + K-Oru, if present) must be <= 100. The reference
↪ solution's logic of `goblins <= 90` being '+' and `goblins > 90 and
↪ goblins <= 100` being 'o' seems to align more with the problem's implied
↪ conditions for safety. The code snippet's conditions are: `goblins > 100`
↪ (never safe), `goblins + 10 > 100` (safe only without K-Oru), and `else`
↪ (always safe). This logic correctly translates to the problem statement's
↪ requirements.\", \"severity\": \"Negligible\" }, { \"inconsistency\":
↪ \"The code snippet does not explicitly define the behavior for edge cases
↪ where N might be outside the specified range of 1 to 50. While the problem
↪ statement guarantees N is within this range, robust code would typically
↪ include checks for such inputs. However, given the constraints, this is a
↪ minor oversight.\", \"severity\": \"Small\" }, { \"inconsistency\": \"The
↪ problem statement implies K-Oru adds 10 goblins to the base count. The
↪ code correctly implements the logic to check if `goblins + 10` exceeds
↪ 100. The problem asks for three categories: always possible to enter,
↪ never possible to enter, or only possible when K-Oru is not present. The
↪ code translates these into symbols: '+' for always safe, 'o' for safe only
↪ without K-Oru, and '-' for never safe. The logic within the conditional
↪ statements correctly determines these categories. For example, if
↪ `goblins` is 91, `goblins + 10` is 101, so it's not always safe and not
↪ never safe, hence 'o'. If `goblins` is 90, `goblins + 10` is 100, so it's
↪ always safe ('+'). If `goblins` is 101, it's never safe ('-'). This logic
↪ appears sound based on the problem description.\", \"severity\":
↪ \"Negligible\" }, { \"inconsistency\": \"The code snippet implements the
↪ logic to categorize cells based on the number of goblins and the presence
↪ of K-Oru. It correctly identifies cells that are always safe ('+'),
↪ sometimes safe ('o'), and never safe ('-'). The counting of these
↪ categories is also accurate. The output format matches the example in the
↪ problem statement. Therefore, the code snippet correctly addresses the
↪ problem's requirements.\", \"severity\": \"Negligible\" }, {
↪ \"inconsistency\": \"The line-by-line explanation correctly describes the
↪ functionality of each line of the code snippet. It accurately explains how
↪ the input is read, how the matrix is processed, and how the conditions for
↪ categorizing cells are evaluated. The explanation covers the logic for
↪ assigning symbols ('+', 'o', '-') and incrementing the respective
↪ counters. The final output of the matrix and the counts is also explained.
↪ No major discrepancies between the code and its explanation were found.\",
↪ \"severity\": \"Negligible\" }, { \"inconsistency\": \"The code snippet is
↪ a correct implementation of the problem statement. It correctly reads the
↪ input, processes the matrix according to the specified conditions for
↪ safety (considering K-Oru's presence and the 100 goblin limit), and
↪ categorizes each cell. The counters for each category are also correctly
↪ updated. The output format is as required. No major logical errors or
↪ inefficiencies were identified.\", \"severity\": \"Negligible\" }, {

```

```

↪ \"inconsistency\": \"The problem statement asks for a determination of
↪ whether it's possible to enter a cell. This implies checking if the
↪ *total* number of goblins (base + K-Oru, if present) is less than or equal
↪ to 100. The code snippet correctly implements this by checking `goblins >
↪ 100` (never safe), `goblins + 10 > 100` (safe only without K-Oru), and the
↪ `else` case (always safe). The interpretation of the conditions and the
↪ resulting symbols ('+', 'o', '-') accurately reflect the problem's
↪ requirements. The reference solution uses a similar logic of categorizing
↪ cells based on goblin counts relative to the 100-goblin limit, also
↪ considering K-Oru's potential addition.\", \"severity\": \"Negligible\" },
↪ { \"inconsistency\": \"The code snippet correctly processes all the cells
↪ in the N x N matrix and calculates the counts for each category. The logic
↪ for determining the category of each cell is based on the number of
↪ goblins and the condition of K-Oru's presence, which adds 10 goblins. The
↪ code correctly identifies cells that are always safe ('+'), only safe when
↪ K-Oru is absent ('o'), and never safe ('-'). The total counts for each
↪ category are accumulated accurately. The output format also matches the
↪ problem requirements.\", \"severity\": \"Negligible\" }, {
↪ \"inconsistency\": \"The code snippet doesn't include any exception
↪ handling for invalid input, such as non-integer values for N or matrix
↪ elements. While the problem statement provides constraints, in a
↪ real-world scenario, this would be a necessary addition for robustness.\",
↪ \"severity\": \"Small\" }, { \"inconsistency\": \"The code snippet
↪ correctly implements all the required functionalities. It reads the input,
↪ processes the matrix based on the goblin counts and K-Oru's impact,
↪ categorizes each cell, counts the cells in each category, and prints the
↪ results in the specified format. There are no missing functionalities or
↪ placeholders.\", \"severity\": \"Negligible\" }, { \"inconsistency\":
↪ \"The problem statement defines the conditions for entering a cell: 'nao
↪ pode haver mais de 100 goblins naquela celula.' and that K-Oru 'equivale a
↪ 10 goblins'. The code snippet correctly interprets this. A cell is always
↪ safe ('+') if `goblins <= 90`, because even with K-Oru (`goblins + 10`),
↪ the total is at most 100. A cell is safe only when K-Oru is absent ('o')
↪ if `goblins > 90` and `goblins + 10 > 100` (meaning `goblins` is between
↪ 91 and 100 inclusive). A cell is never safe ('-') if `goblins > 100`, as
↪ even without K-Oru, the limit is exceeded. This logic is sound and
↪ directly addresses the problem's safety conditions.\", \"severity\":
↪ \"Negligible\" }, { \"inconsistency\": \"The line-by-line explanation
↪ covers all the essential parts of the code snippet, including variable
↪ initializations, input reading, matrix traversal, conditional logic for
↪ cell categorization, and output printing. It provides a clear
↪ understanding of how the code works.\", \"severity\": \"Negligible\" }, {
↪ \"inconsistency\": \"The explanation correctly describes what the code
↪ actually does. It accurately reflects the logic for determining cell
↪ safety based on goblin counts and K-Oru's presence. The symbols assigned
↪ ('+', 'o', '-') and the counting mechanism are all explained in alignment
↪ with the code's behavior.\", \"severity\": \"Negligible\" }, {
↪ \"inconsistency\": \"The explanation accurately covers all relevant parts
↪ of the snippet. It describes the initialization of counters, the nested
↪ loops for matrix traversal, the `map(int, input().split())` for reading
↪ input, the conditional statements that define the cell categories, and the
↪ `print()` statements for output. All significant logic and variables are
↪ addressed.\", \"severity\": \"Negligible\" }, { \"inconsistency\": \"The
↪ explanation is generally understandable. It uses clear language to
↪ describe the code's functionality. However, it could benefit from slightly

```

```

↪ more detail on *why* certain conditions are chosen (e.g., why `goblins <=
↪ 90` implies always safe). For instance, explicitly stating that `90 + 10 =
↪ 100` clarifies the 'always safe' condition.\", \"severity\": \"Minor\" },
↪ { \"inconsistency\": \"The explanation is well-organized, following the
↪ flow of the code from input to output. It uses bullet points and clear
↪ sentence structures, making it easy to follow.\", \"severity\":
↪ \"Negligible\" }, { \"inconsistency\": \"The explanation does not
↪ explicitly relate the snippet to its broader purpose or why certain design
↪ decisions were made. For example, it doesn't discuss the trade-offs of
↪ this categorization method or its role in a larger system (if applicable).
↪ The focus is primarily on explaining *what* the code does, rather than
↪ *why*.\", \"severity\": \"Moderate\" }, { \"inconsistency\": \"The
↪ explanation does not acknowledge potential limitations, such as the lack
↪ of error handling for invalid inputs, which was noted as a minor code
↪ issue. It also doesn't discuss how the code might behave with inputs
↪ outside the specified constraints (though the problem statement limits
↪ these).\", \"severity\": \"Minor\" }, { \"inconsistency\": \"The
↪ explanation has minor grammatical issues and could be more fluent in
↪ places. For example, some sentences are a bit long or could be rephrased
↪ for better clarity.\", \"severity\": \"Minor\" }, { \"inconsistency\":
↪ \"The explanation aligns well with the code's inline comments and naming
↪ conventions. The tone is consistent with technical documentation. The
↪ explanation is straightforward and avoids overly technical jargon where
↪ unnecessary.\", \"severity\": \"Negligible\" }, { \"inconsistency\": \"The
↪ explanation is reasonably accessible for a junior developer. It explains
↪ the core logic clearly. However, it could be enhanced by elaborating on
↪ the reasoning behind the conditional logic to improve understanding for
↪ learners.\", \"severity\": \"Moderate\" }, { \"inconsistency\": \"The code
↪ correctly implements the logic to determine cell safety based on the
↪ number of goblins and the potential addition of K-Oru (+10 goblins), with
↪ a maximum of 100 goblins allowed for entry. The categorization into '+',
↪ 'o', and '-' symbols, along with the counting, is accurate. The output
↪ format also matches the problem's requirements. No logical errors were
↪ found.\", \"severity\": \"Negligible\" }, { \"final_answer\": -2 } ] ",
"final_answer": -2
},
{
"response": " # Solucao para analise de seguranca na cidade dos goblins N =
↪ int(input())           # Le o tamanho N da matriz quadrada
↪ matriz_goblins = []   # Armazenara as estimativas de
↪ goblins # Le a matriz de estimativas (sem contar K-Oru) for _ in range(N):
↪ linha = list(map(int, input().split())) # Le uma linha de inteiros
↪ matriz_goblins.append(linha) # Adiciona a matriz # Variaveis para
↪ contagem dos tipos de celulas sempre_seguro = 0 #
↪ Celulas com <= 90 goblins condicional = 0 # Celulas
↪ com 91-100 goblins (K-Oru faz diferenca) nunca_seguro = 0
↪ # Celulas com >100 goblins # Processa cada celula da
↪ matriz for i in range(N): for j in range(N): goblins =
↪ matriz_goblins[i][j] # Estimativa atual (sem K-Oru) #
↪ K-Oru vale +10 goblins quando presente na celula if goblins > 100:
↪ # Nunca seguro (mesmo sem K-Oru) print('-',
↪ end='') nunca_seguro += 1 elif goblins > 90:
↪ # Depende de K-Oru (91-100 sem ele) print('o', end='')
↪ # 101-110 com ele nunca_seguro condicional += 1 else:
↪ # Sempre seguro (<=90 sem ele, <=100 com ele)

```

```

↪      print('+', end='')          sempre_seguro += 1  print()
↪                                  # Nova linha apos cada linha da matriz #
↪ Imprime resumo (com linhas vazias antes e depois) print(f'\ +:
↪ {sempre_seguro}') # Celulas sempre seguras print(f'o: {condicional}')
↪                  # Celulas condicionalmente seguras print(f'-: {nunca_seguro}\
↪ ') # Celulas nunca seguras ",
"metric_laaJ": " [ {  \inconsistency": \The code snippet does not fully
↪ implement the problem statement's requirements. Specifically, it does not
↪ account for K-Oru's presence in the conditional cells (those with 91-100
↪ goblins) by adding 10 to the count and re-evaluating their safety. The
↪ problem states that if K-Oru is present, a cell with 91-100 goblins
↪ becomes 101-110, which would then make it 'never safe' (represented by
↪ '-'), not 'conditionally safe' (represented by 'o').\, \severity":
↪ \Major\ } , { \inconsistency": \The problem statement asks to count
↪ the total number of cells where it is possible to enter, based on the
↪ simplified matrix. The code snippet calculates and prints counts for '+',
↪ 'o', and '-' cells, but it doesn't explicitly calculate the total number
↪ of 'entry-possible' cells as requested by the problem statement. In the
↪ context of the problem, '+' cells are always safe, and 'o' cells are safe
↪ only when K-Oru is not present. However, the problem's output requirement
↪ is to 'contar em quantas celulas e possivel entrar'. The current output
↪ counts types of cells but not the total possible entry cells.\,
↪ \severity": \Major\ } , { \inconsistency": \The problem statement
↪ states: 'Para cada celula, sera definido se sempre e possivel entrar la,
↪ se nunca e possivel, ou se so e possivel entrar quando K-Oru nao estiver
↪ presente na celula.' The code snippet defines these based on goblin
↪ counts: '+' (sempre seguro) for <= 90, 'o' (condicional) for 91-100, and
↪ '-' (nunca seguro) for > 100. However, the interpretation of 'o' as
↪ 'conditionally safe' is flawed given K-Oru's effect. If K-Oru adds 10
↪ goblins, then a cell with 91-100 goblins will have 101-110 goblins when
↪ K-Oru is present, making it 'never safe' ('-'), not 'conditionally safe'
↪ ('o'). The current logic treats 'o' as conditionally safe meaning K-Oru
↪ *makes* it safe, which is the opposite of the problem's implication.\,
↪ \severity": \Major\ } , { \inconsistency": \The explanation states
↪ that 'K-Oru vale +10 goblins quando presente na celula'. However, the code
↪ does not use this information to re-evaluate the 'condicional' cells.
↪ Instead, it assigns 'o' to cells with 91-100 goblins and stops processing.
↪ The problem implies that these 'o' cells are safe only if K-Oru is absent.
↪ When K-Oru is present, these cells become unsafe (91+10=101 to
↪ 100+10=110). The code incorrectly labels these as 'conditionally safe'
↪ ('o') without reflecting this K-Oru presence rule.\, \severity":
↪ \Major\ } , { \inconsistency": \The line-by-line explanation for the
↪ conditional logic is incorrect. For example, the explanation states: 'elif
↪ goblins > 90: # Depende de K-Oru (91-100 sem ele) print('o', end='') #
↪ 101-110 com ele nunca seguro condicional += 1'. This comment correctly
↪ identifies that with K-Oru, these cells become unsafe (101-110). However,
↪ the code itself does not implement this re-evaluation; it simply assigns
↪ 'o' to `condicional` and prints 'o'. The code does not check if K-Oru is
↪ present or absent to determine the final safety status for these cells,
↪ which is a critical oversight according to the problem statement.\,
↪ \severity": \Fatal\ } , { \inconsistency": \The code does not output
↪ the total count of cells where entry is possible. The problem statement
↪ explicitly asks to 'contar em quantas celulas e possivel entrar'. The code
↪ only prints the counts of '+', 'o', and '-' cells. While one could infer
↪ entry-possible cells by summing '+' and 'o', the problem requires a direct

```

```

↪ count of 'entry-possible' cells.\", \"severity\": \"Major\" }, {
↪ \"inconsistency\": \"The explanation claims the code determines 'se sempre
↪ e possivel entrar la, se nunca e possivel, ou se so e possivel entrar
↪ quando K-Oru nao estiver presente na celula'. While the code assigns
↪ symbols ('+', 'o', '-'), its logic for 'o' is flawed. It labels cells with
↪ 91-100 goblins as 'conditionally safe' ('o'), implying they are safe when
↪ K-Oru is absent. However, it doesn't correctly represent that these cells
↪ become *unsafe* when K-Oru is present (as 91+10=101, 100+10=110). The
↪ code's representation of conditional safety is incomplete and
↪ misleading.\", \"severity\": \"Major\" }, { \"inconsistency\": \"The
↪ explanation for the `else` block states: '# Sempre seguro (<=90 sem ele,
↪ <=100 com ele)'. This statement is incorrect because the condition
↪ `goblins <= 90` is for the `else` block, meaning it is executed when
↪ `goblins` is not greater than 100 and not greater than 90. Thus, it
↪ handles `goblins <= 90`. With K-Oru, this becomes `goblins + 10 <= 100`,
↪ which is true for `goblins <= 90`. The explanation correctly reflects the
↪ condition `goblins <= 90` but then incorrectly states `goblins <= 100`
↪ with him, which is not explicitly checked by the code's structure as the
↪ `else` block is for `goblins <= 90`.\", \"severity\": \"Major\" }, {
↪ \"inconsistency\": \"The explanation for the `elif goblins > 90:` block
↪ states: '# 101-110 com ele nunca seguro'. This comment correctly
↪ identifies the outcome when K-Oru is present. However, the code does not
↪ implement this logic. The code assigns 'o' to `condicional` for `goblins`
↪ in the range 91-100, and this count (`condicional`) is printed as 'Celulas
↪ condicionalmente seguras'. This is a misrepresentation of the problem's
↪ rules, as the problem implies these cells are *only* safe when K-Oru is
↪ absent, and unsafe when present. The code does not differentiate or
↪ reflect this by changing the symbol or the count type.\", \"severity\":
↪ \"Fatal\" }, { \"inconsistency\": \"The explanation incorrectly describes
↪ the outcome for cells with 91-100 goblins. It states '# 101-110 com ele
↪ nunca seguro', which is a correct deduction from the problem statement.
↪ However, the code then assigns these cells to the 'condicional' category
↪ and prints 'o', labeled as 'Celulas condicionalmente seguras'. This is
↪ contradictory: if they become unsafe with K-Oru, they are not
↪ 'conditionally safe' in the way the code presents them.\", \"severity\":
↪ \"Fatal\" }, { \"inconsistency\": \"The problem asks to determine if a
↪ cell is 'sempre e possivel entrar la', 'nunca e possivel', or 'so e
↪ possivel entrar quando K-Oru nao estiver presente'. The code's output for
↪ 'o' cells (91-100 goblins) is 'Celulas condicionalmente seguras'. This
↪ implies they are safe under certain conditions. However, the problem
↪ states that K-Oru is always present in a cyclic pattern and is worth 10
↪ goblins. Thus, cells with 91-100 goblins would become 101-110 goblins when
↪ K-Oru is present, making them 'never safe'. The code fails to capture this
↪ nuance, treating 'o' as a separate safe category rather than a category
↪ that becomes unsafe.\", \"severity\": \"Major\" }, { \"inconsistency\":
↪ \"The code does not output the total number of cells where entry is
↪ possible. The problem statement requires this count. The code provides
↪ counts for '+' (always safe), 'o' (conditionally safe, but
↪ miscategorized), and '-' (never safe). A correct solution would identify
↪ all '+' cells as entry-possible and also all 'o' cells *when K-Oru is
↪ absent*, and then sum these to get the total entry-possible cells. Since
↪ the code doesn't correctly identify 'o' cells and doesn't sum up the
↪ entry-possible cells, it fails this requirement.\", \"severity\":
↪ \"Major\" }, { \"final_answer\": -21 } ] \",
\"final_answer\": -21

```

] }