UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

ALICE OLIVEIRA DE QUEIROZ BRITO

Avaliação de docstrings utilizando LLMs: uma análise baseada em atributos de qualidade

Recife

2025

ALICE OLIVEIRA DE QUEIROZ BRITO

Avaliação de docstrings utilizando LLMs: uma análise baseada em atributos de qualidade

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Graduado em Engenharia da Computação.

Área de Concentração: Engenharia de software

Orientador (a): Filipe Carlos de Albuquerque Calegario

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Brito, Alice Oliveira de Queiroz.
Avaliação de docstrings utilizando LLMs: uma análise baseada em atributos de qualidade / Alice Oliveira de Queiroz Brito. - Recife, 2025.
71 p. : il., tab.

Orientador(a): Filipe Carlos de Albuquerque Calegario
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. Documentação de código. 2. Docstring. 3. Modelos de linguagem grandes. 4. GPT. 5. Avaliação. 6. LLM-as-a-Judge. I. Calegario, Filipe Carlos de Albuquerque. (Orientação). II. Título.

000 CDD (22.ed.)

ALICE OLIVEIRA DE QUEIROZ BRITO

Avaliação de docstrings utilizando LLMs: uma análise baseada em atributos de qualidade

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 05/08/2025

BANCA EXAMINADORA

_____

Profa. Dr. Filipe Carlos de Albuquerque Calegario (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Leopoldo Motta Teixeira (Examinador Interno)

Universidade Federal de Pernambuco

Dedico este trabalho aos meus pais, minha irmã e meus amigos.

AGRADECIMENTOS

Agradeço, antes de tudo, à minha mãe, minha irmã e meu pai, que me ajudaram a ser quem sou hoje.

Agradeço aos meus amigos da faculdade, especialmente Aninha, Bilas, Vic, Day, Elias, Morone e Lucca, que me acompanharam e me aguentaram em cada projeto desde 2019.

Aos meus amigos do IF, em especial Bril, Mari e Nanda, que me acompanham a mais de uma década, estiveram comigo nos piores momentos e que, sem dúvida, vou levar comigo pros melhores também.

E agradeço aos professores que ao longo da graduação me motivaram e me deram orientação, na monitoria, nos projetos de pesquisa e agora no TCC.

RESUMO

A documentação de código é fundamental para a compreensão e manutenção de sistemas de software. As docstrings, por estarem integradas ao código, são amplamente utilizadas e favorecem a atualização contínua da documentação. O avanço dos Large Language Models tem impulsionado esforços para automatizar tanto a geração quanto a avaliação desse tipo de conteúdo. Métodos tradicionais de avaliação apresentam limitações, seja pela baixa expressividade de métricas automáticas, seja pela subjetividade e alto custo da avaliação manual. Então, este trabalho investiga o uso de LLMs como avaliadores automatizados da qualidade de docstrings, com base nos critérios de completude, acurácia, legibilidade e relevância. Foram testadas diferentes estratégias de prompting, incluindo few-shot prompting, e comparadas com avaliações humanas. Os resultados mostram que modelos como o GPT-4.1 apresentam forte correlação com os julgamentos manuais em todos os critérios, mesmo com instruções simples. Também foi demonstrada a viabilidade de utilizar os modelos para corrigir automaticamente docstrings com base em justificativas, elevando a qualidade final da documentação. Conclui-se que os LLMs são ferramentas promissoras para compor pipelines automatizados de geração, avaliação e refinamento de docstrings, embora a supervisão humana ainda seja necessária em casos mais críticos.

Palavras-chaves: Documentação de código. Docstring. Modelos de linguagem grandes. GPT. Avaliação. LLM-as-a-Judge.

ABSTRACT

Code documentation is essential for the understanding and maintenance of software systems. Docstrings, being embedded in the code, are widely used and facilitate the continuous updating of documentation. The advancement of Large Language Models (LLMs) has driven efforts to automate both the generation and evaluation of such content. Traditional evaluation methods face limitations, either due to the limited expressiveness of automatic metrics or the subjectivity and high cost of manual assessment. This study investigates the use of LLMs as automated evaluators of docstring quality, focusing on the criteria of completeness, accuracy, readability, and relevance. Various prompting strategies, including few-shot prompting, were tested and compared against human evaluations. The results show that models such as GPT-4.1 exhibit strong correlation with human judgments across all criteria, even when provided with simple instructions. The study also demonstrates the feasibility of using these models to automatically revise docstrings based on evaluation justifications, thereby improving overall documentation quality. It is concluded that LLMs are promising tools for integrating into automated pipelines for docstring generation, evaluation, and refinement, although human oversight remains necessary in more critical cases.

Keywords: Code documentation. Docstrings. Large language models. GPT. Evaluation. LLM-as-a-Judge.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A documentação desempenha um papel fundamental no desenvolvimento e manutenção de *software*, sendo uma das práticas mais recomendadas para auxiliar no processo de desenvolvimento (SOUZA; ANQUETIL; OLIVEIRA, 2005). Estima-se que aproximadamente 58% do tempo dos desenvolvedores seja dedicado à compreensão de código, uma atividade diretamente influenciada pela presença (ou ausência) de documentação adequada (BILLAH; RAHMAN; ROY, 2025).

A relevância da documentação se reflete na diversidade de seus formatos, cada um voltado para aspectos específicos do desenvolvimento e uso de sistemas. De modo geral, esses tipos podem ser agrupados conforme sua finalidade: documentações voltadas ao usuário final, como manuais e tutoriais, que oferecem uma visão geral do sistema e orientações de uso; materiais exemplificativos, como guias passo a passo e galerias de exemplos, que auxiliam na execução de tarefas específicas e no aprendizado de funcionalidades; e documentações técnicas, destinadas a desenvolvedores e à equipe técnica, que descrevem detalhes internos necessários para a manutenção e evolução do código (GEIGER et al., 2018; SILVA; UNTERKALMSTEINER; WNUK, 2023).

Dentre os vários formatos existentes, este trabalho foca especificamente nas *docstrings*: comentários estruturados incluídos no corpo do código que descrevem o comportamento de funções, classes e métodos. Esses comentários são considerados os artefatos mais importantes para entender o sistema, atrás somente do próprio código-fonte (SOUZA; ANQUETIL; OLIVEIRA, 2005). Estudos reforçam essa percepção demonstrando que códigos acompanhados de comentários são significativamente mais fáceis de entender do que aqueles sem nenhuma anotação (WOODFIELD; DUNSMORE; SHEN, 1981). Além disso, se destacam pela conveniência, por estarem incorporadas diretamente no código elas tendem a ser mais facilmente mantidas atualizadas à medida que o sistema evolui (BILLAH; RAHMAN; ROY, 2025).

Abaixo temos um exemplo de *docstring* real obtido através do *dataset* CodeSearchNet.

Código Fonte 1 – Exemplo de docstring

```
/*
* Compute checksum for a file.
*
* @param {string} filename The absolute path to a filename.
* @return {string} The checksum for `filename`."
*/
```

```
7  function computeChecksum(filename) {
       var contents = fs.readFileSync(filename);
9
       var hash = crypto
11         .createHash('md5')
           .update(contents)
13         .digest('base64')
           .replace(/=+$/, '');
15
       return hash;
17 }
```

Essa *docstring* é objetiva e descreve corretamente a funcionalidade principal da função: calcular o *checksum* de um arquivo. Ela especifica o tipo e significado do parâmetro, bem como o tipo e significado do valor de retorno. Esse formato é típico de estilos como o *JSDoc* e favorece a leitura por desenvolvedores e ferramentas automatizadas.

No entanto, mesmo em um exemplo aparentemente correto, é possível identificar pontos passíveis de melhoria, como por exemplo, não há exemplo de uso ou justificativa da escolha do algoritmo *md5*. Ou seja, mesmo quando a documentação está presente, lacunas sutis podem comprometer ou limitar a compreensão do código.

A avaliação de *docstrings* tem impacto direto na sua escrita, manutenção e no uso de ferramentas dependentes de documentação. Por exemplo, em abordagens de geração de teste baseadas em descrições textuais, uma *docstring* incompleta pode levar a casos de teste incorretos ou insuficientes. Da mesma forma, ferramentas de geração automática de documentação que extraem *docstrings* para produzir materiais navegáveis dependem da clareza e completude dos comentários, e em ambientes de desenvolvimento integrados (IDEs), *docstrings* de qualidade enriquecem recursos como *auto-complete*, facilitando o trabalho de desenvolvedores.

Contudo, a simples presença desses comentários não garante sua utilidade. Embora ofereçam vantagens significativas para a compreensão e manutenção, sua eficácia está diretamente ligada à qualidade do conteúdo que apresentam. Comentários mal escritos ou desatualizados podem comprometer o entendimento do sistema, levando a interpretações equivocadas e dificultando a manutenção do código.

No entanto, produzir documentação de alta qualidade requer tempo, atenção e constante alinhamento com a evolução do código, aspectos que, na prática, são frequentemente negligenciados devido à pressão por prazos e alta produtividade (STEIDL; HUMMEL; JUERGENS, 2013). Como consequência, a documentação costuma ser tratada como um artefato secundário den-

tro do processo de desenvolvimento (SILVA; UNTERKALMSTEINER; WNUK, 2023), resultando em materiais frequentemente mal escritos ou desatualizados, o que contribui para a insatisfação dos desenvolvedores com a documentação disponível (SCHRECK; DALLMEIER; ZIMMERMANN, 2007).

Nos últimos anos, com o surgimento e a popularização dos *large language models* (LLMs), houve um avanço significativo na performance de tarefas relacionadas ao código, como a geração automática de código e de documentação de *software* (SUN et al., 2024; BILLAH; RAHMAN; ROY, 2025). A geração de documentação, em especial, surge como uma resposta promissora ao problema recorrente de documentação negligenciada nos projetos de software. Já no contexto da geração de código, a presença de documentação permanece fundamental: estudos indicam que incorporar descrições em linguagem natural no processo de geração melhora significativamente a qualidade do código produzido (ZHOU et al., 2022). E além disso, a documentação desempenha um papel crucial na compreensão e manutenção do código gerado, contribuindo para sua reutilização e evolução ao longo do tempo.

Apesar disso, a garantia de qualidade das documentações geradas por LLMs continua sendo um desafio (YANG et al., 2025). Durante o desenvolvimento de métodos de geração automática de documentação, a avaliação das saídas geradas geralmente tem se limitado ao uso de métricas automáticas de NLP, muitas vezes baseadas em similaridade com referências humanas, ou então à avaliação manual por especialistas (SUN et al., 2024). Ambas as abordagens apresentam limitações: as métricas automáticas nem sempre refletem aspectos qualitativos importantes (DIGGS et al., 2024) e a avaliação humana, além de ser custosa, não é escalável ou reproduzível (CHIANG; LEE, 2023).

Além das limitações associadas aos métodos de avaliação existentes, há também uma falta de consenso sobre o que configura uma documentação de alta qualidade. A qualidade da documentação é um conceito multifacetado, que pode ser analisado sob diferentes critérios e perspectivas, variando conforme o tipo e o contexto de uso da documentação (TREUDE; MIDDLETON; ATAPATTU, 2020). Abordagens simplificadas, como classificações binárias, não capturam adequadamente essas nuances. Ainda assim, a literatura carece de diretrizes claras e sistemáticas sobre quais critérios devem ser adotados e como aplicá-los de maneira consistente em diferentes cenários (RANI et al., 2023).

Diante das limitações dos métodos tradicionais de avaliação, uma abordagem alternativa tem ganhado destaque: o *LLM-as-a-Judge*. Nesse paradigma, modelos de linguagem são utilizados como avaliadores, com potencial para capturar nuances presentes em textos em lin-

guagem natural, como aqueles encontrados em documentações técnicas (CHIANG; LEE, 2023). A técnica tem demonstrado resultados promissores em tarefas como revisão de código (oES; VENSON, 2024) e avaliação de textos gerados por LLMs (CHIANG; LEE, 2023), oferecendo uma solução mais escalável, econômica e potencialmente mais consistente do que a avaliação humana. No entanto, sua aplicação específica à avaliação de documentação de código ainda está em estágio inicial, exigindo investigações que validem sua eficácia e adequação aos diversos critérios de qualidade associados a esse tipo de artefato.

Motivado por esse contexto, este trabalho investiga a viabilidade do uso de LLMs como avaliadores de documentação de código. As principais etapas e contribuições do estudo são:

(i) Identificação e sistematização dos critérios fundamentais para a avaliação da qualidade da documentação de código, estabelecendo uma base conceitual para as análises subsequentes;

(ii) Formulação e implementação de diferentes estratégias de *prompting* para a utilização de um LLM como avaliador, com o objetivo de examinar a correspondência entre os julgamentos gerados pelo modelo e as avaliações realizadas por humanos;

(iii) Aplicação do método de avaliação desenvolvido no item (ii) para analisar e corrigir docstrings geradas por LLMs, investigando se a técnica contribui para a melhoria da qualidade da documentação.

O objetivo do trabalho é investigar a viabilidade do uso de LLMs para avaliação de documentação de código em forma de *docstrings*, motivados pela necessidade de automatizar o processo de atualizar e gerar documentação. Ao reunir esses elementos, o estudo oferece uma base inicial para o uso de modelos de linguagem na avaliação automatizada de documentação de código.

Este trabalho está estruturado em quatro seções principais. Na Seção 2, são apresentados os principais trabalhos relacionados. A Seção 3 descreve os procedimentos adotados, incluindo a definição dos critérios de avaliação e as etapas de execução do estudo. Em seguida, a Seção 4 discute os resultados obtidos e a Seção 5 apresenta as limitações do estudo. Por fim, a Seção 6 apresenta os principais achados e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO

Atualmente, diversos trabalhos exploram a geração e avaliação de documentação de código. Esta seção apresenta os principais estudos relacionados, destacando suas contribuições e limitações.

2.1 AVALIAÇÃO DE DOCUMENTAÇÃO GERADA POR LLMS

Antes de definir critérios de avaliação para documentações, é necessário compreender como essas avaliações têm sido conduzidas na literatura recente, especialmente em um cenário onde o uso de LLMs para geração de documentação se torna cada vez mais comum.

No trabalho *Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation* (DIGGS et al., 2024) os autores investigaram o uso de LLMs na geração de documentação para código legado, utilizando conjuntos de dados com linguagens como MUMPS e *Assembly Language Code*. Para avaliar a qualidade dos comentários gerados, foi elaborada uma rubrica baseada em quatro critérios: completude, legibilidade, utilidade e ausência de alucinações. O estudo também analisou a correlação entre essas avaliações humanas e métricas automáticas.

Os resultados indicaram que, de modo geral, os comentários produzidos pelos modelos eram completos, legíveis, úteis e não apresentavam alucinações. No entanto, nenhuma das métricas automáticas utilizadas, incluindo BLEU, ROUGE, métricas de custo e complexidade ciclomática, apresentou correlação significativa com os julgamentos humanos.

Segundo os autores, a única forma de garantir a correção da documentação gerada é contar com especialistas humanos, que precisam investir tempo considerável para revisar e validar as respostas. Como caminho para reduzir essa dependência, o trabalho aponta a necessidade de desenvolver novas métricas de avaliação.

Em *Using Large Language Models to Document Code: A First Quantitative and Qualitative Assessment* (GUELMAN et al., 2024) avaliaram o desempenho do GPT-3.5-turbo na geração automática de documentação de código (Javadoc) para métodos e classes. A análise envolveu tanto uma avaliação quantitativa, usando a métrica BLEU, quanto uma avaliação qualitativa com revisão humana.

Os resultados mostraram que o LLM foi capaz de gerar documentação de alta qualidade.

No entanto, uma limitação importante identificada foi a inadequação do BLEU para avaliar a qualidade de documentação, a métrica mostrou baixa correlação com as avaliações humanas. O estudo reforça a necessidade de métricas de avaliação mais adequadas para tarefas de geração de documentação de código.

Já no trabalho *A Comparative Analysis of Large Language Models for Code Documentation Generation* (DVIVEDI et al., 2024) os autores realizaram uma análise comparativa envolvendo cinco LLMs (GPT-3.5, GPT-4, Bard, Llama 2 e Starchat) para geração de documentação de código em diferentes níveis (inline, função e arquivo).

Foram avaliadas 14 docstrings, tanto originais quanto geradas por LLMs, com base em seis critérios: acurácia, completude, relevância, compreensibilidade, legibilidade e tempo de geração. As cinco primeiras métricas foram analisadas manualmente por dois avaliadores humanos, com o uso de checklists para reduzir a subjetividade. Já o tempo de geração foi mensurado automaticamente. Não foram empregadas métricas automáticas de qualidade textual, toda a análise de qualidade da documentação baseou-se exclusivamente na avaliação humana, com o apoio dos checklists.

2.2 AVALIAÇÃO DE DOCUMENTAÇÃO UTILIZANDO LLMS

No estudo *DocAgent: A Multi-Agent System for Automated Code Documentation Generation* (YANG et al., 2025), é introduzido o DocAgent, um sistema multiagente projetado para gerar documentação automaticamente. O sistema processa o código-fonte ordenado topologicamente e utiliza agentes especializados para compor diferentes partes da documentação.

A etapa de avaliação automatizada proposta no trabalho considera três dimensões principais: completude, utilidade e veracidade. Essa avaliação é realizada pelo agente denominado *Verifier*, responsável por aprovar ou rejeitar a documentação gerada com base em critérios pré-definidos, além de fornecer sugestões específicas para melhorias.

Os autores argumentam que métricas tradicionais de avaliação de geração de linguagem natural não são aplicáveis nesse contexto devido à ausência de *gold references*, e que a avaliação humana, embora mais precisa, é subjetiva, onerosa e difícil de escalar, especialmente em ambientes com integração contínua. Diante disso, propõem uma abordagem híbrida de avaliação, que combina verificações estruturais determinísticas, julgamentos qualitativos via LLMs e validação com base de dados.

A completude é verificada por meio de uma checagem automatizada baseada na *Abstract*

Syntax Tree. Já a utilidade é avaliada em termos de clareza, concisão, profundidade descritiva, utilidade dos atributos documentados e da capacidade de orientar o desenvolvedor sobre o uso adequado do componente. Cada seção da documentação (sumário, descrição principal, parâmetros, etc.) possui *prompts* específicos, adaptados para avaliar critérios distintos. Esses *prompts* incluem rubricas explícitas com escalas de pontuação detalhadas, exemplos concretos e instruções passo a passo para guiar o LLM no processo de avaliação.

Apesar da proposta inovadora, o trabalho não discute como a própria avaliação automatizada foi validada. Em vez disso, concentra-se apenas na qualidade final da documentação gerada pelo sistema multiagente, sem examinar se o LLM está de fato avaliando de forma confiável.

Em *Leveraging Language Models for Code Comment Classification* (PATEL, 2023), foi realizada a classificação automática de comentários de código em úteis ou inúteis utilizando LLMs. O trabalho aborda o desafio de distinguir comentários informativos de anotações redundantes ou triviais.

Os autores realizaram experimentos com dois conjuntos de dados: um com 10 mil comentários rotulados manualmente por humanos e outro com 100 mil comentários gerados e classificados automaticamente pelo GPT.

Foram utilizados modelos como DistilGPT e GPT-2, aplicando fine-tuning com os dados rotulados. E para modelos maiores, como Codex e GPT-3, apenas extraíram embeddings dos comentários e treinaram classificadores simples para prever sua utilidade. Além disso, conduziram experimentos para avaliar o impacto de fatores como o tamanho do pré-treinamento, o comprimento dos comentários e o tipo de arquitetura utilizada.

Os testes mostraram que os LLMs alcançaram mais de 90% de acurácia na tarefa de classificação, com o GPT-3 chegando a 96%. O estudo demonstra o potencial desses modelos na avaliação automatizada de comentários. No entanto, a abordagem se limita a uma classificação binária da utilidade, sem considerar outras dimensões relevantes da qualidade da documentação.

O trabalho *Code Documentation and Analysis to Secure Software Development* (ATTIE et al., 2024) apresenta uma ferramenta chamada CoDAT, cujo principal objetivo é manter a consistência da documentação do código. Isso significa que, sempre que uma linha de código é modificada, o comentário correspondente também deve ser ajustado. Para realizar essa verificação, os autores utilizam um modelo de linguagem (Claude Haiku) com a finalidade de checar a consistência semântica entre o fragmento de código e os comentários que o descrevem.

No entanto, os autores destacam uma limitação importante: o modelo demonstrou tendência a gerar respostas incorretas durante a verificação de consistência, resultando em falsos positivos ou negativos. Como solução futura, o trabalho sugere o uso de múltiplos LLMs em paralelo, validação cruzada com métodos formais e o desenvolvimento de *prompts* mais robustos, visando reduzir erros de interpretação.

No trabalho *Source Code Summarization in the Era of Large Language Models* (SUN et al., 2024), os autores investigaram a tarefa de sumarização de código com o uso de LLMs, avaliando tanto a qualidade dos resumos gerados quanto os métodos utilizados para essa avaliação. Um dos principais focos foi analisar a viabilidade de empregar LLMs, especificamente o GPT-3.5 e o GPT-4, como substitutos ou complementos à avaliação humana.

Para isso, os modelos foram instruídos de forma semelhante aos avaliadores humanos: a cada exemplo, recebiam o código-fonte, um resumo de referência e os resumos gerados por diferentes sistemas, sendo solicitados a atribuir notas de 1 a 5 com base na qualidade percebida. Os resultados mostraram que o GPT-4 alcançou a maior correlação (Spearman) com as avaliações humanas, superando todas as métricas automáticas tradicionais, tanto as baseadas em similaridade textual (BLEU, METEOR, ROUGE-L) quanto as baseadas em similaridade semântica (BERTScore, SBERT).

3 EXPERIMENTOS

Nesta seção, são apresentadas as etapas realizadas para a condução dos experimentos. Primeiramente, foram definidos os critérios de avaliação adotados ao longo do estudo. Em seguida, foram selecionados e organizados os dados utilizados nos testes. Então, realizou-se a avaliação de *docstrings* escritas por humanos e, por fim, foram geradas *docstrings* para os mesmos trechos de código, que passaram por uma etapa de avaliação seguida de correção com base nos resultados obtidos.

3.1 CRITÉRIOS DE AVALIAÇÃO

A primeira etapa do processo consistiu em definir como as *docstrings* seriam avaliadas e o que caracteriza uma boa documentação. Para isso, foram selecionados trabalhos que abordassem essa questão, com foco em revisões e mapeamentos sistemáticos da literatura, além de *surveys* com desenvolvedores. A partir dessas fontes, identificamos os principais critérios utilizados na avaliação de documentação.

Foram analisados trabalhos que abordavam diferentes formatos de documentação, mas em sua maioria tratavam de comentários de código em diversas linguagens. Esses trabalhos destacavam diversos atributos de qualidade, muitos recorrentes entre si, embora com variações sutis em seus significados.

Com base nessa análise, foi possível reunir os atributos de qualidade utilizados para avaliar documentação, listados abaixo:

- **Acessibilidade ou disponibilidade:** Mede a facilidade com que o conteúdo da documentação pode ser acessado pelos desenvolvedores de software (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020; HARGIS et al., 2004).
- **Acurácia:** Avalia não apenas a veracidade das informações, mas também a precisão com que o conteúdo do comentário é expresso. Comentários vagos ou excessivamente abstratos podem ser considerados imprecisos, dificultando a transmissão clara da informação. A acurácia impacta diretamente na facilidade com que os profissionais compreendem e aplicam o conteúdo da documentação (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020; HARGIS et al., 2004)

- **Atualização:** Avalia o quanto os comentários são atualizados em relação à evolução do software. Comentários desatualizados podem conter informações incorretas ou omitir dados relevantes, o que pode levar os profissionais a erros ou interpretações equivocadas (RANI et al., 2023; ZHI et al., 2015; SANTOS; CORREIA, 2020).

- **Clareza:** Avalia a ausência de ambiguidade e obscuridade, isto é, se o conteúdo é compreendido desde a primeira leitura (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020; HARGIS et al., 2004).

- **Coerência:** Como comentário e código são relacionados, por exemplo, o comentário do método deve ser relacionado ao nome do método. (RANI et al., 2023)

- **Completude:** Avalia se o conteúdo do comentário é completo para apoiar tarefas de desenvolvimento e manutenção. O comentário precisa ter todas as informações necessárias sobre o código para que os profissionais consigam realizar suas tarefas. Caso alguma informação essencial falte, a documentação não cumpre seu propósito e deixa de ser útil. (RANI et al., 2023; ZHI et al., 2015; SANTOS; CORREIA, 2020; HARGIS et al., 2004)

- **Compreensibilidade:** Refere-se à capacidade dos comentários de contribuir para o entendimento do sistema (RANI et al., 2023; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020).

- **Concisão:** Mede até que ponto os comentários não são verbosos e não contém informação desnecessária. (RANI et al., 2023; PLöSCH; DAUTOVIC; SAFT, 2014)

- **Consistência / uniformidade:** Avalia se o conteúdo do comentário é coerente com o restante da documentação e com o comportamento do código, verificando a ausência de contradições entre partes distintas. Também inclui a uniformidade na forma de apresentação e no formato utilizado. Manter a consistência contribui para uma documentação mais clara, confiável e fácil de navegar. (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020)

- **Corretude:** Avalia se a informação no comentário é correta ou não, e se não tem nenhum conflito com informações factuais. Se a documentação apresenta informações incorretas, é provável de confundir e atrapalhar os profissionais. (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020)

- **Formato:** Considera o estilo de escrita, uso de diagramas, exemplos e elementos visuais que ajudam na compreensão do conteúdo (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; HARGIS et al., 2004).
- **Grau de automação:** Existência de processos automatizados para escrever ou manter documentação (SANTOS; CORREIA, 2020).
- **Informação sobre autores:** Identificação dos autores, rastreabilidade sobre quem criou e editou o documento, importante para garantir a qualidade da documentação (RANI et al., 2023; ZHI et al., 2015).
- **Internacionalização:** Verifica se a documentação é corretamente traduzida ou adaptada para diferentes idiomas (RANI et al., 2023).
- **Legibilidade:** Mede a facilidade de leitura dos comentários, considerando a fluidez do texto, estrutura e linguagem acessível (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020).
- **Manutenibilidade:** Avalia o quão fácil é manter os comentários atualizados ao longo do tempo (RANI et al., 2023; SANTOS; CORREIA, 2020).
- **Ortografia e gramática:** Avalia a correção gramatical e ortográfica. Muitos erros pode comprometer a experiência de leitura da documentação (RANI et al., 2023; ZHI et al., 2015).
- **Organização da informação:** Refere-se à forma como os conteúdos são estruturados logicamente nos comentários, facilitando a compreensão (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014; HARGIS et al., 2004).
- **Padronização ou estruturação:** Definição e implementação de padrões e diretrizes para o desenvolvimento de documentação (PLöSCH; DAUTOVIC; SAFT, 2014; SANTOS; CORREIA, 2020).
- **Rastreabilidade:** Verifica se as modificações nos comentários podem ser rastreadas, indicando quando, como, por que e por quem foram feitas (RANI et al., 2023; ZHI et al., 2015; PLöSCH; DAUTOVIC; SAFT, 2014).
- **Relevância do conteúdo / orientação à tarefa:** Avalia o quanto o conteúdo do comentário é relevante para o propósito da documentação, com foco em dar suporte

aos objetivos dos usuários e ajudá-los na realização de suas tarefas. (RANI et al., 2023; SANTOS; CORREIA, 2020)

- **Suficiência:** A documentação fornece informação suficiente para o leitor atingir seus objetivos (SANTOS; CORREIA, 2020)
- **Tecnologia de documentação:** Avalia se as ferramentas utilizadas para escrever, gerar e armazenar a documentação estão atualizadas (RANI et al., 2023).
- **Usabilidade ou utilidade:** Mede a capacidade da documentação de ajudar os leitores a atingir seus objetivos, considerando sua utilidade prática, satisfação de uso e impacto gerado (RANI et al., 2023; SANTOS; CORREIA, 2020).

Esses atributos podem ser organizados em oito grupos conforme mostra a Figura 1.

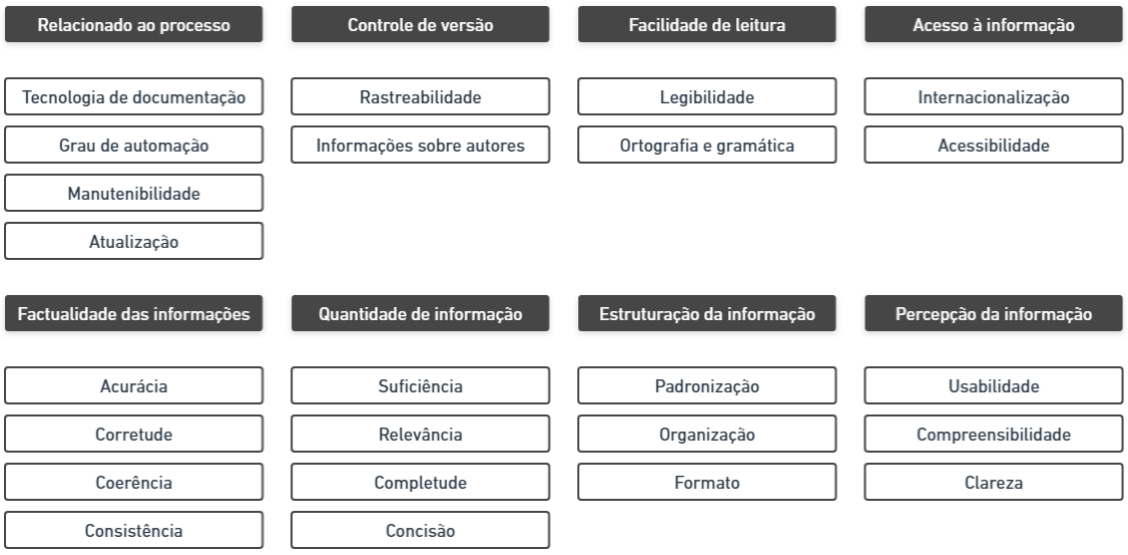

Figura 1 – Atributos de qualidade por grupo

Além dos critérios apresentados, alguns trabalhos também consideram aspectos relacionados ao custo da documentação, como os custos de desenvolvimento, manutenção e uso (ZHI et al., 2015; SANTOS; CORREIA, 2020). No entanto, esses fatores não podem ser plenamente avaliados por meio de inspeção manual, pois dependem do uso real da documentação em contextos práticos.

De modo geral, os estudos indicam que não há uma definição única do que constitui uma boa documentação, mas que existem atributos amplamente reconhecidos como fundamentais para sua qualidade.

Para definir os critérios que seriam usados na avaliação da documentação, o primeiro passo foi excluir aqueles que não poderiam ser aplicados com a metodologia adotada ou que não são adequados ao tipo de documentação analisado, no caso, *docstrings* associadas a funções.

Inicialmente, foram descartados os atributos dos grupos de critérios relacionados ao processo, controle de versão e acesso à informação. Esses critérios exigem informações contextuais mais amplas sobre o sistema ou infraestrutura de documentação, o que está fora do escopo deste trabalho.

Em seguida, também foram excluídos os atributos que dependem da comparação entre diferentes documentos do mesmo sistema, como consistência e padronização. Como a análise será feita com base em unidades isoladas, isto é, cada função acompanhada de sua *docstring*, não é possível avaliar a uniformidade entre documentos.

Então, reagrupamos os critérios restantes em quatro grupos principais: factualidade das informações, quantidade de informação, apresentação das informações e percepção das informações, isso pode ser visto na Figura 2, que também destaca os critérios considerados mais importantes considerando os trabalhos analisados. E a presença de ao menos um critério de cada agrupamento entre os mais relevantes demonstra que a divisão reflete de forma adequada as múltiplas dimensões da documentação.

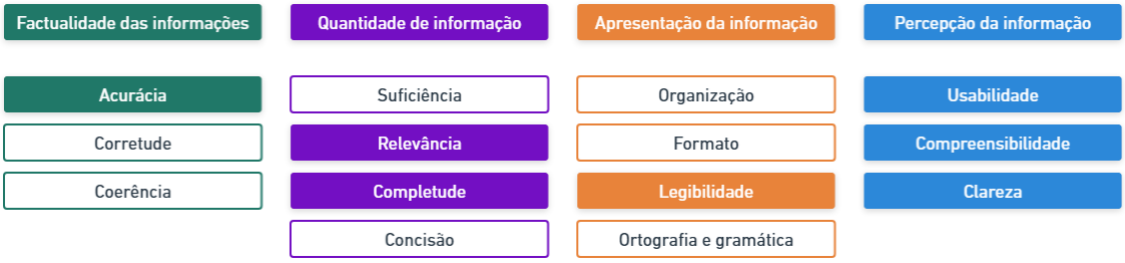

Figura 2 – Atributos de qualidade finais

Em seguida, analisamos a relação entre os critérios individualmente. Embora eles tenham sido agrupados com base em suas características principais, essa divisão por si só não é suficiente para compreender como os critérios se conectam. A Figura 3 ilustra essas relações.

A partir do diagrama, observa-se que a usabilidade (ou utilidade) não constitui um critério simples ou isolado, sendo diretamente influenciada por diversos outros critérios. Por esse motivo, ela pode ser considerada o atributo de qualidade mais importante, uma vez que, para que a documentação cumpra sua função, é necessário que seja efetivamente utilizada no apoio às tarefas dos desenvolvedores. Considerando-se a importância e as relações entre os critérios, foram selecionados quatro que representam adequadamente as diferentes dimensões da docu-

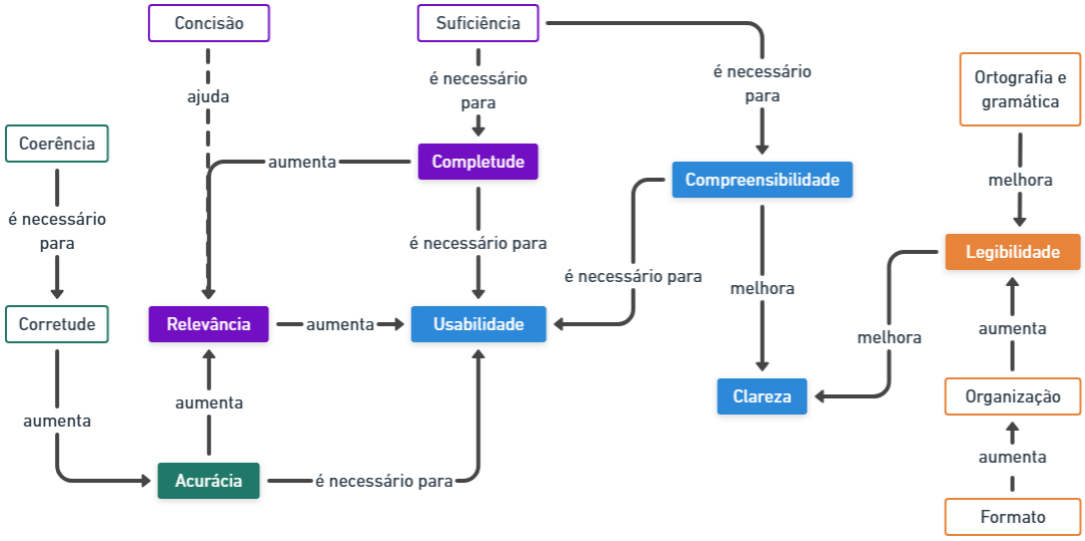

Figura 3 – Diagrama da relação entre critérios

mentação: completude, acurácia, relevância e clareza. As definições atribuídas a esses critérios podem ter sido levemente ajustadas em relação às formulações iniciais:

- **Completude:** Avalia se o comentário cobre todas as partes essenciais do código para que um desenvolvedor compreenda, utilize e mantenha o código. São considerados elementos como finalidade, lógica principal, parâmetros, valores de retorno e, quando apropriado, exemplos de uso ou casos especiais. Essa definição cobre completude e suficiência.
- **Clareza:** Avalia se o comentário é escrito de forma clara, direta e compreensível. O texto deve ser bem estruturado, livre de ambiguidade, vaguidade ou complexidade desnecessária. Essa definição cobre clareza, compreensibilidade, legibilidade, ortografia e gramática, organização e formato.
- **Acurácia:** Avalia se o comentário está correto e é preciso, ou seja, não omite informações que possam resultar em uso incorreto do código. Essa definição cobre acurácia, corretude e coerência.
- **Relevância:** Avalia se o comentário agrega valor prático à compreensão e ao uso do código, explicando apenas o que é necessário e evitando repetir o que já é evidente na implementação. Essa definição cobre relevância e concisão.

3.2 DADOS UTILIZADOS

Inicialmente, realizou-se uma busca por *datasets* que contivessem trechos de código, suas respectivas *docstrings* e alguma forma de avaliação de qualidade. No entanto, não foram encontrados *datasets* abrangentes que apresentassem avaliações separadas por critérios de qualidade. Diante dessa limitação, optou-se pela avaliação manual de um conjunto de pares de código e comentário, conduzida por um avaliador com base nos critérios estabelecidos neste estudo.

Os pares de código e comentário foram obtidos a partir do *dataset* CodeSearchNet[1], que reúne trechos em diversas linguagens de programação, extraídos de repositórios disponíveis no GitHub.

Esse dataset foi escolhido devido à sua ampla adoção na literatura. Além disso, oferece um grande volume de dados e suporte a múltiplas linguagens de programação, o que viabiliza a expansão futura dos experimentos de avaliação para outros contextos. Outro fator relevante é que o CodeSearchNet já se encontra pré-processado e organizado, contendo exemplos extraídos de projetos reais do GitHub, o que garante variedade de estilos e maior realismo nas amostras analisadas.

Embora o repositório original tenha sido arquivado e o link de acesso ao *dataset* esteja indisponível, a base de dados permaneceu acessível por meio da plataforma Hugging Face[2].

A seleção das *docstrings* foi realizada de forma aleatória com base nos seguintes critérios: (i) o código correspondente deveria ser compreensível pelo avaliador; (ii) as *docstrings* não deveriam conter links externos, uma vez que não seriam acessíveis durante a avaliação por modelos de linguagem; (iii) os comentários deveriam estar escritos em inglês; e (iv) foram selecionados apenas pares pertencentes ao subconjunto de código em JavaScript, devido à familiaridade do avaliador com essa linguagem, a fim de evitar avaliações incorretas decorrentes de possíveis interpretações equivocadas do código.

3.3 AVALIAÇÃO

Esta seção descreve os experimentos conduzidos para avaliar a eficácia de diferentes estratégias de *prompting* na avaliação de *docstrings* por LLMs. São detalhados os métodos

[1] https://github.com/github/CodeSearchNet
[2] https://huggingface.co/datasets/code-search-net/code_search_net

utilizados, bem como os resultados obtidos com a aplicação de cada uma das estratégias.

3.3.1 Testes exploratórios

Inicialmente, foram classificadas manualmente 30 *docstrings* com o objetivo de analisar, de forma preliminar, o comportamento dos modelos de linguagem na tarefa de avaliação. Esta etapa tem caráter exploratório e não busca, neste momento, resultados estatisticamente significativos.

Com base em estratégias identificadas na literatura, foi elaborado um *prompt* que define a tarefa de avaliação e apresenta os quatro critérios adotados: completude, relevância, acurácia e clareza. Para cada critério, foram fornecidas descrições detalhadas para pontuações de 1 a 5, com o intuito de reduzir a subjetividade das respostas. O modelo recebia como entrada um trecho de código e sua respectiva *docstring*, devendo atribuir uma nota para cada critério com base nas definições estabelecidas.

A avaliação foi conduzida no ambiente de avaliação do *Playground* da OpenAI. Para cada *docstring*, comparou-se diretamente a nota atribuída pelo LLM à nota atribuída pela avaliação humana em cada um dos critérios.

Os resultados iniciais foram considerados moderados e, com base na análise dos erros, as definições de pontuação do *prompt* foram ajustadas iterativamente. No entanto, essas alterações não promoveram melhorias relevantes na quantidade de acertos em todos os critérios. Um dos *prompts* utilizados estão disponibilizados no Apêndice A.

Os resultados obtidos, incluindo acertos e erros com diferença de até um ponto, são apresentados na Tabela 1.

Tabela 1 – Desempenho dos modelos por critério de avaliação

Critério	**GPT-4.1**				**GPT-4.1-mini**			
	Exato	**%**	**Aprox**	**%**	**Exato**	**%**	**Aprox**	**%**
Completude	12	40,00%	27	90,00%	10	33,33%	27	90,00%
Acurácia	14	46,67%	27	90,00%	18	60,00%	29	96,67%
Claridade	10	33,33%	26	86,67%	14	46,67%	26	86,67%
Relevância	17	56,67%	27	90,00%	13	43,33%	28	93,33%

Na tentativa de melhorar a taxa de acertos exatos, foi testada uma abordagem em que cada critério foi avaliado separadamente. A proposta visava investigar se a separação dos critérios

reduziria possíveis interferências decorrentes da avaliação simultânea. Os *prompts* utilizados encontram-se no Apêndice B.

Como algumas definições de pontuação foram ajustadas, todas as *docstrings* foram reavaliadas com base nas novas instruções. A Tabela 2 apresenta os resultados obtidos pelo modelo GPT-4.1-mini, que demonstrou o melhor desempenho entre os modelos testados. No entanto, apenas o critério de completude apresentou diferença significativa.

Tabela 2 – Acertos Exatos do GPT-4.1-mini por Critério

Critério	**Acertos Exatos**	**Porcentagem**
Completude	21	70,00%
Acurácia	16	53,33%
Clareza	14	46,67%
Relevância	13	43,44%

Observou-se que critérios mais subjetivos, em especial, clareza, continuavam apresentando desempenho inferior, independentemente do modelo utilizado. Diante disso, foram testadas outras estratégias, como *few-shot prompting* e o uso de *checklists* para orientar a avaliação desse critério. Embora os testes tenham indicado uma melhora na taxa de acertos, seria necessário um número maior de iterações para se alcançar um *prompt* realmente eficaz, além de ser necessária a replicação da abordagem para os demais critérios. Por esse motivo, optou-se por interromper essa linha de experimentação e explorar alternativas capazes de produzir resultados satisfatórios com menor complexidade.

3.3.2 Ajustes finais e redefinição de critérios

Nos testes finais, duas alterações principais foram implementadas com base nas análises anteriores. A primeira consistiu na substituição do critério de clareza pelo de legibilidade. Essa mudança foi motivada pelo fato de que a clareza tende a envolver aspectos mais subjetivos relacionados à compreensão do conteúdo, enquanto a legibilidade se concentra na qualidade da escrita, configurando-se como um critério mais consistente para fins de avaliação. A seguir, apresenta-se a definição adotada para o novo critério:

- **Legibilidade:** Mede a facilidade de leitura dos comentários, considerando a fluidez do texto, estrutura e linguagem acessível. Essa definição cobre legibilidade, ortografia e gramática, organização e formato.

A segunda mudança consistiu na adoção de um novo método para análise dos resultados: a partir desta etapa, as respostas dos LLMs passaram a ser avaliadas também por meio de um coeficiente de correlação. Essa decisão foi motivada pela dificuldade em se obter correspondência exata entre as notas. Considerando que variações entre avaliadores humanos são relativamente comuns, isso levanta a possibilidade de que não seja realista esperar uma correspondência perfeita por parte dos LLMs em relação às avaliações humanas (CHIANG; LEE, 2023).

Assim, optou-se por empregar um coeficiente de correlação com o objetivo de verificar o grau de associação entre as notas atribuídas pelos LLMs e aquelas provenientes da avaliação humana. O coeficiente adotado foi o de Kendall (τ), sua utilização no estudo *Can Large Language Models Be an Alternative to Human Evaluation?* (CHIANG; LEE, 2023), que avaliou critérios de qualidade em textos em inglês, contribuiu para a escolha neste trabalho.

O coeficiente de Kendall τ mede a correlação ordinal entre duas classificações, considerando a proporção de pares concordantes (avaliados na mesma ordem por ambos os julgadores) e discordantes (avaliados em ordem inversa). Em termos práticos, essa métrica busca responder à seguinte questão: "Dado que o avaliador humano atribuiu uma nota maior à *docstring* A do que à *docstring* B, o LLM também fez essa distinção de forma consistente?"

Para interpretação dos valores de τ, foi adotada a escala proposta por Rosenthal, conforme apresentado na Tabela 3.

Tabela 3 – Interpretação dos valores de correlação

Valor de correlação	**Interpretação**
$[0,0,1]$ ou $[0,-0,1]$	Desprezível
$[0,1,0,3]$ ou $[-0,1,-0,3]$	Fraca
$[0,3,0,5]$ ou $[-0,3,-0,5]$	Média
$[0,5,0,7]$ ou $[-0,5,-0,7]$	Forte
$[0,7,1]$ ou $[-0,7,-1]$	Muito forte

Fonte: AMIDEI; PIWEK; WILLIS (2019)

Para as avaliações finais, foram analisadas manualmente 150 *docstrings*. Os *prompts* utilizados nessa etapa, descritos no Apêndice C, foram empregados em conjunto com a funcionalidade de *structured output*, cuja estrutura de resposta foi definida externamente por meio de um *JSON Schema*, além da pontuação, a resposta precisava conter a justificativa para cada nota.

Embora modelos como o GPT-4.1 tenham respondido adequadamente às instruções de formatação incluídas no próprio *prompt*, outros, como o GPT-4o, apresentaram dificuldades, produzindo objetos com formatos inconsistentes. Além disso, a partir desta fase, todos os testes passaram a ser executados por meio da *batch* API da OpenAI, o que permitiu maior escalabilidade e controle no processamento das requisições.

E afim de aumentar a reprodutibilidade dos experimentos, os modelos foram avaliados com configurações tão semelhantes quanto possível, adotando temperatura igual a zero e a mesma *seed*, sempre que essas definições eram suportadas.

3.3.3 Avaliação simples

Nessa avaliação, o modelo foi instruído a atribuir notas de um a cinco para cada um dos quatro critérios, completude, relevância, acurácia e legibilidade, além de fornecer uma justificativa para cada pontuação. O *prompt* apenas informou a definição de cada critério. A Tabela 4 apresenta os resultados obtidos, incluindo a porcentagem de acertos e os valores do coeficiente de correlação de Kendall (τ).

Tabela 4 – Desempenho dos modelos por critério de avaliação: prompt simples

Modelo	**Completude**		**Acurácia**		**Legibilidade**		**Relevância**	
	%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
GPT-4.1	44.00	0.709	37.33	0.668	**50.67**	**0.640**	43.33	0.634
GPT-4.1-mini	36.00	**0.718**	50.00	0.569	45.44	0.530	40.00	**0.705**
o3	24.67	0.700	24.67	0.512	45.33	0.538	**44.67**	0.642
GPT-4o	32.67	0.618	25.33	**0.677**	42.00	0.563	27.33	0.608
GPT-4o-mini	**46.00**	0.647	**51.33**	0.548	45.33	0.495	34.00	0.549
GPT-3.5-turbo	40.00	0.418	28.67	0.180	38.00	0.190	32.67	0.324

Como era esperado, o modelo GPT-3.5-turbo apresentou o pior desempenho entre os avaliados. E de modo geral, os modelos não foram capazes de reproduzir exatamente as mesmas notas atribuídas na avaliação humana, no entanto, os coeficientes indicaram correlação forte e muito forte, que mostra que as avaliações seguiram uma tendência semelhante, evidenciando alinhamento com os julgamentos humanos.

A Figura 4 apresenta os coeficientes de correlação entre os modelos. Observa-se que, em geral, os modelos apresentaram correlações muito fortes entre si, com exceção do GPT-3.5-turbo. Além disso, os critérios de completude e relevância demonstraram maior consistência entre os modelos, enquanto acurácia e legibilidade apresentaram correlações mais baixas, tanto

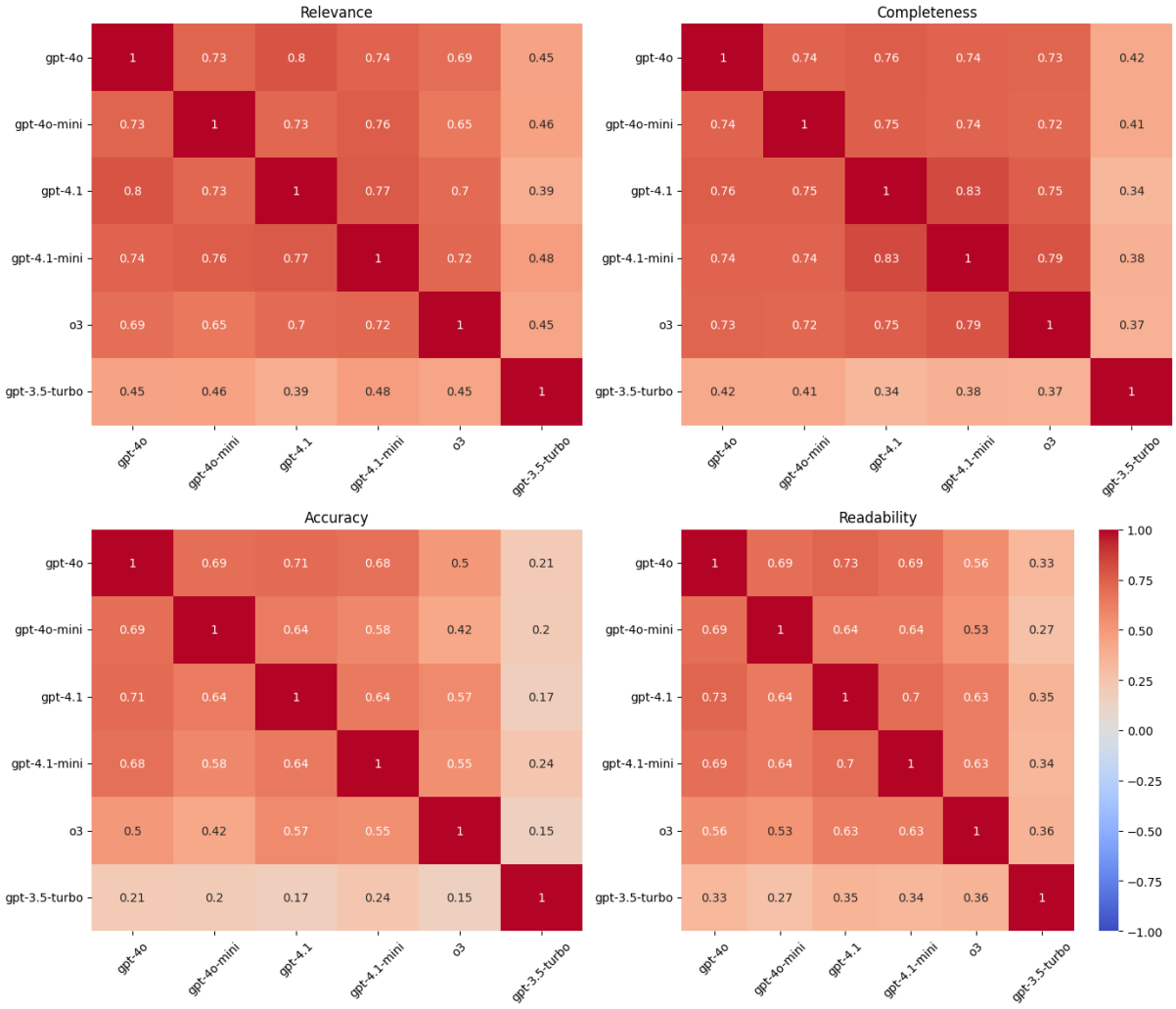

Figura 4 – Correlação entre modelos: prompt simples

entre os próprios modelos quanto em relação à avaliação humana. Esse resultado pode indicar que esses dois critérios não estão suficientemente bem definidos nos *prompts* utilizados.

3.3.4 Avaliação com definição das notas

Neste estágio, retomou-se a primeira estratégia adotada na fase exploratória: a inclusão das descrições detalhadas para cada pontuação, de um a cinco, em todos os critérios de avaliação. O *prompt* correspondente encontra-se descrito no Apêndice C. E a partir dessa etapa, o GPT-3.5-turbo deixou de ser utilizado como avaliador, em razão de sua correlação significativamente divergente em relação aos demais modelos.

Os resultados obtidos estão apresentados na Tabela 5, acompanhados por setas que indicam se houve melhoria ou piora em relação à abordagem apresentada na Seção 3.3.3.

Tabela 5 – Desempenho dos modelos por critério de avaliação: escala de pontuação

Modelo	Completude		Acurácia		Legibilidade		Relevância	
	%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
GPT-4.1	52.00 ↑	0.731 ↑	42.00 ↑	0.587 ↓	42.67 ↓	**0.568** ↓	**56.00** ↑	0.708 ↑
GPT-4.1-mini	53.33 ↑	**0.756** ↑	**56.67** ↑	0.494 ↓	45.33 ↓	0.481 ↓	44.00 ↑	0.655 ↓
o3	45.33 ↑	0.755 ↑	34.67 ↑	0.356 ↓	**48.00** ↑	0.483 ↓	42.67 ↓	0.688 ↑
GPT-4o	48.00 ↑	0.718 ↑	28.00 ↑	**0.671** ↓	24.00 ↓	0.484 ↓	32.00 ↑	**0.716** ↑
GPT-4o-mini	**59.33** ↑	0.705 ↑	53.33 ↑	0.432 ↓	44.67 ↑	0.411 ↓	38.00 ↑	0.565 ↑

Observou-se uma melhora nos resultados de completude e relevância, mas uma piora em legibilidade e na correlação de acurácia. A redução no desempenho relacionado à legibilidade sugere que a escala de pontuação adotada para esse critério não está adequadamente definida, indicando a necessidade de ajustes no *prompt*.

A Figura 5 mostra que, mesmo após a adoção da escala, as correlações em acurácia e legibilidade permanecem moderadas, indicando que os problemas de interpretação entre os modelos persistem.

Foram realizadas novas tentativas de aprimoramento nos critérios de acurácia e legibilidade, por meio da revisão das definições presentes no *prompt*. Uma das versões testadas encontra-se no Apêndice D, e os resultados correspondentes estão apresentados na Tabela 6. Nessa etapa, os testes foram realizados apenas com os modelos GPT-4.1 e GPT-4.1-mini, que haviam apresentado desempenho ligeiramente superior nas fases anteriores.

De modo geral, não foram observadas melhorias nos critérios de acurácia e legibilidade. E embora apenas esses dois critérios tenham sido modificados, pequenas variações de desempenho também foram identificadas nos critérios de completude e relevância.

Tabela 6 – Desempenho dos modelos por critério de avaliação: redefinição da escala

Modelo	Completude		Acurácia		Legibilidade		Relevância	
	%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
GPT-4.1	56.67 ↑	0.743 ↑	38.67 ↓	0.666 ↑	47.33 ↑	0.568 ↓	58.67 ↑	0.755 ↑
GPT-4.1-mini	54.00 ↑	0.759 ↑	62.67 ↑	0.648 ↑	49.33 ↑	0.461 ↓	42.00 ↓	0.666 ↓

3.3.5 Avaliação utilizando few-shot prompting

Por fim, ainda com o objetivo de aprimorar os resultados, foi adotada a estratégia de *few-shot prompting*. Foram incluídos três exemplos de *docstrings*: um de alta qualidade (com

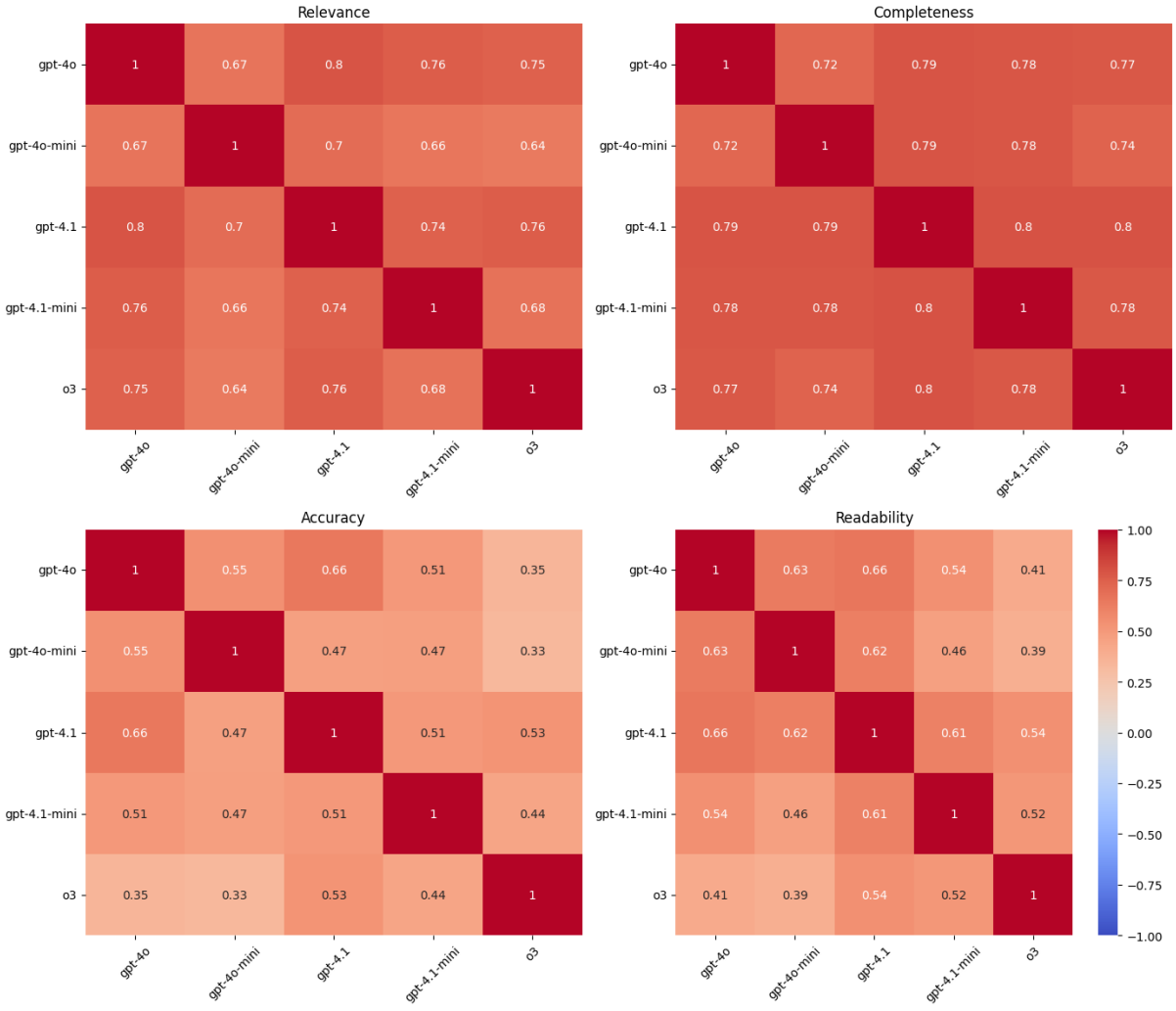

Figura 5 – Correlação entre modelos: escala de pontuação

notas entre 4 e 5), um de qualidade média (notas entre 2 e 4) e um de baixa qualidade (notas entre 1 e 3). O *prompt* utilizado está descrito no Apêndice C. Os resultados dessa abordagem estão apresentados na Tabela 7.

Tabela 7 – Desempenho dos modelos por critério de avaliação: few-shot prompting

Modelo	**Completude**		**Acurácia**		**Legibilidade**		**Relevância**	
	%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
GPT-4.1	52.67 ↑	0.711 ↑	46.00 ↑	**0.657** ↓	**57.33** ↑	**0.640** =	46.67 ↑	0.639 ↑
GPT-4.1-mini	46.67 ↑	0.721 ↑	**60.00** ↑	0.608 ↑	40.00 ↓	0.503 ↓	**48.67** ↑	**0.702** ↓
o3	29.33 ↑	**0.731** ↑	27.33 ↑	0.449 ↓	47.33 ↑	0.549 ↑	40.67 ↓	0.638 ↓
GPT-4o	42.67 ↑	0.686 ↑	29.33 ↑	0.651 ↓	38.00 ↓	0.533 ↓	35.33 ↑	0.681 ↑
GPT-4o-mini	**59.33** ↑	0.726 ↑	48.00 ↓	0.560 ↑	53.33 ↑	0.526 ↑	44.67 ↑	0.643 ↑

A estratégia resultou em melhorias, especialmente em completude. Em acurácia, houve

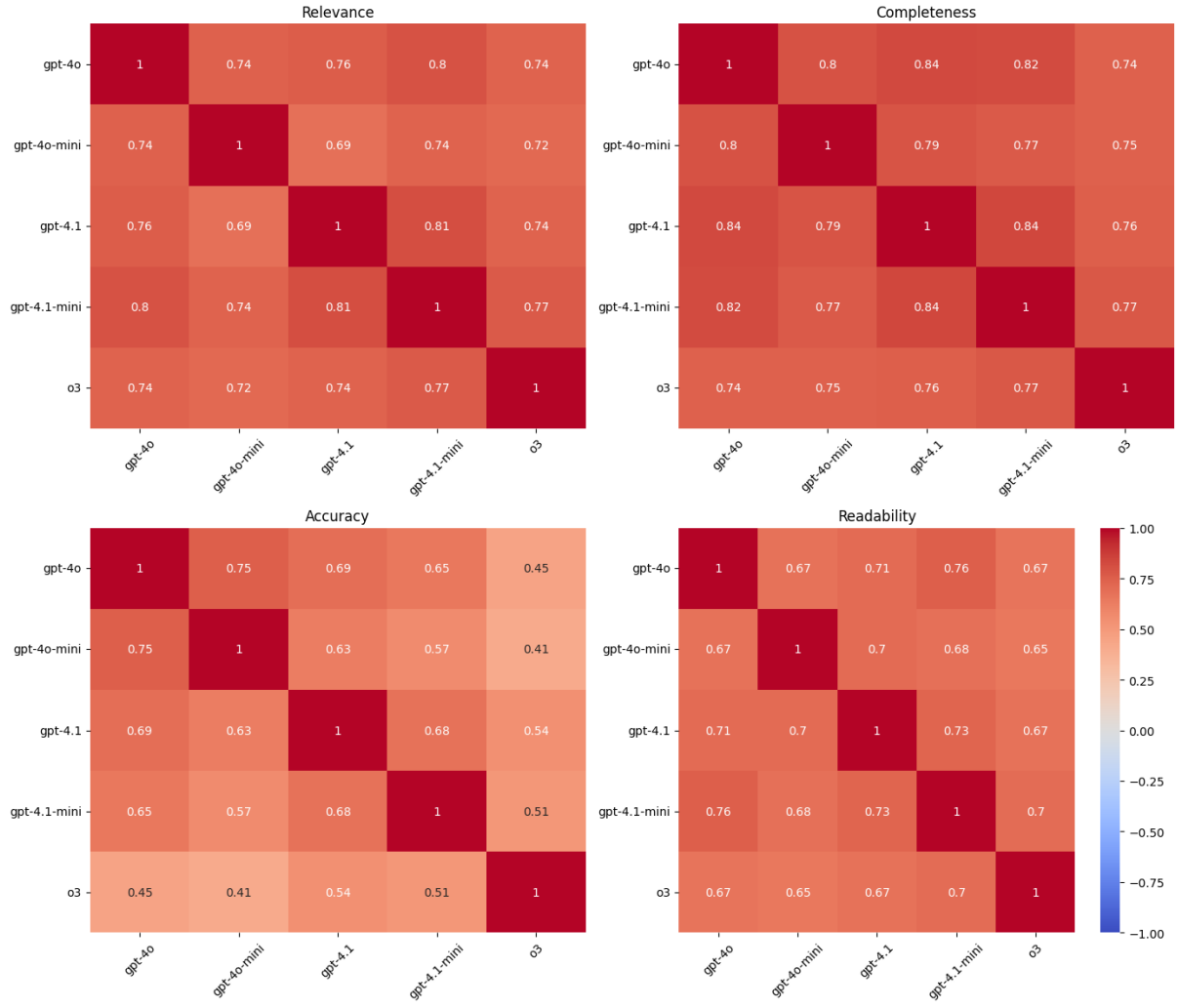

Figura 6 – Correlação entre modelos: few-shot prompting

aumento nos acertos exatos, mas leve queda na correlação. Legibilidade e relevância variaram conforme o modelo. Destacam-se uma redução de 0,063 na correlação de acurácia do modelo o3 e uma queda de 5,44% nos acertos de legibilidade do GPT-4.1-mini. Em contrapartida, observou-se melhoria superior a 10% em acertos e aumento de 0.09 na correlação em outros casos.

A Figura 6 revela que, no critério de legibilidade, os modelos apresentaram forte correlação entre si, mas apenas moderada em relação à avaliação humana, possivelmente devido ao uso de exemplos apenas com notas intermediárias. Em acurácia, o modelo o3 destoou dos demais. Já para legibilidade e relevância, a correlação entre modelos foi superior à observada em abordagens anteriores.

A partir disso, foram adicionados dois novos exemplos: uma *docstring* de péssima qualidade e outra de qualidade mediana. O resultado da avaliação com esse novo *prompt*, descrito no

Apêndice D, está apresentado na Tabela 8. A comparação de desempenho considera agora os resultados anteriores da Tabela 7 como referência.

Tabela 8 – Desempenho dos modelos por critério de avaliação: few-shot prompting com mais exemplos

Modelo	**Completude**		**Acurácia**		**Legibilidade**		**Relevância**	
	%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
GPT-4.1	60.54 ↑	0.740 ↑	46.26 ↑	0.630 ↓	52.38 ↓	0.619 ↓	53.74 ↑	0.690 ↑
GPT-4.1-mini	49.33 ↑	0.727 ↑	57.33 ↓	0.636 ↑	39.33 ↓	0.508 ↑	44.00 ↓	0.686 ↓

Embora o novo conjunto de exemplos tenha passado a incluir diferentes níveis de qualidade para o critério de legibilidade, não foram observadas melhorias nesse aspecto.

3.3.6 Avaliação com diferentes configurações

Nessas avaliações, o *prompt* simples da Seção 3.3.3 foi reaplicado ao modelo GPT-4.1, com a modificação apenas dos parâmetros de temperatura e *seed*, a fim de analisar o impacto dessas variações nos resultados. A Tabela 9 apresenta os resultados obtidos.

Tabela 9 – Desempenho do GPT-4.1 com ajustes em seed e temperatura

Seed	**Temp**	**Completude**		**Acurácia**		**Legibilidade**		**Relevância**	
		%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
42	0	44.00	0.709	**37.33**	0.668	**50.67**	**0.640**	44.33	**0.634**
42	0.6	42.00	0.704	30.67	0.648	**50.67**	0.602	43.33	0.626
42	1	44.97	0.705	33.56	0.581	42.95	0.532	42.28	0.618
4367	0	41.50	0.700	36.05	0.677	49.66	0.631	45.58	0.630
999	0.8	44.67	0.689	34.67	0.623	**50.67**	0.581	**48.00**	0.609
333	0.5	**47.33**	**0.736**	35.33	0.641	**50.67**	0.583	43.33	0.631
57	0.2	42.95	0.697	34.23	**0.686**	48.32	0.603	44.30	0.616

A escolha por variar exclusivamente esses dois parâmetros deve-se ao fato de que ambos afetam diretamente o grau de aleatoriedade na geração das respostas. Ao manter os demais parâmetros constantes, buscou-se garantir um experimento controlado, permitindo observar isoladamente a influência dessas variações.

De modo geral, não foram observadas mudanças significativas nos resultados apenas com o ajuste desses parâmetros.

3.4 GERAÇÃO E CORREÇÃO DE DOCSTRINGS

Partindo da premissa de que a avaliação de *docstrings* por LLMs apresenta proximidade suficiente com a avaliação humana, conforme indicado pelos resultados obtidos na seção anterior, o passo seguinte consistiu na geração de documentação para as mesmas 150 funções. Em seguida, essas *docstrings* foram avaliadas e, com base nos resultados das avaliações, submetidas a um processo de correção automatizada.

3.4.1 Geração

As *docstrings* foram geradas a partir da instrução simples "*Generate the docstring for the given function.*", utilizando dois modelos: GPT-4o e, para fins comparativos, GPT-3.5-Turbo. A avaliação das documentações geradas foi conduzida com o *prompt* da Seção 3.3.3, utilizando o modelo GPT-4.1.

A Figura 7 mostra a distribuição das notas atribuídas às *docstrings* geradas pelo GPT-3.5-Turbo. Apesar da predominância de avaliações positivas, notas inferiores ainda foram registradas, principalmente nos critérios de acurácia e completude.

E a Figura 8 mostra a distribuição das notas atribuídas às *docstrings* geradas pelo GPT-4o. No geral, a maior parte das respostas recebeu nota máxima, com poucas exceções.

Com o objetivo de compreender as principais limitações das *docstrings* geradas pelo GPT-4o, foi realizada uma análise qualitativa das justificativas do LLM associadas às notas inferiores a 5. Das 150 *docstrings* avaliadas, 31 receberam pelo menos uma nota abaixo do valor máximo.

De modo geral, os erros e omissões apontados foram de fato observados. No entanto, em alguns casos, o modelo avaliador adotou critérios mais rígidos, penalizando a *docstring* por ausência de informações supérfluas.

A seguir, são detalhadas as justificativas nos casos em que houve penalização no critério de acurácia:

- Penalização por informação incorreta, embora a *docstring* estivesse correta: seis casos;
- Penalização por informação incorreta, com a *docstring* de fato incorreta: oito casos;
- Penalização por omissão de informação considerada importante, mas que não comprometia a interpretação ou o uso da função: sete casos;

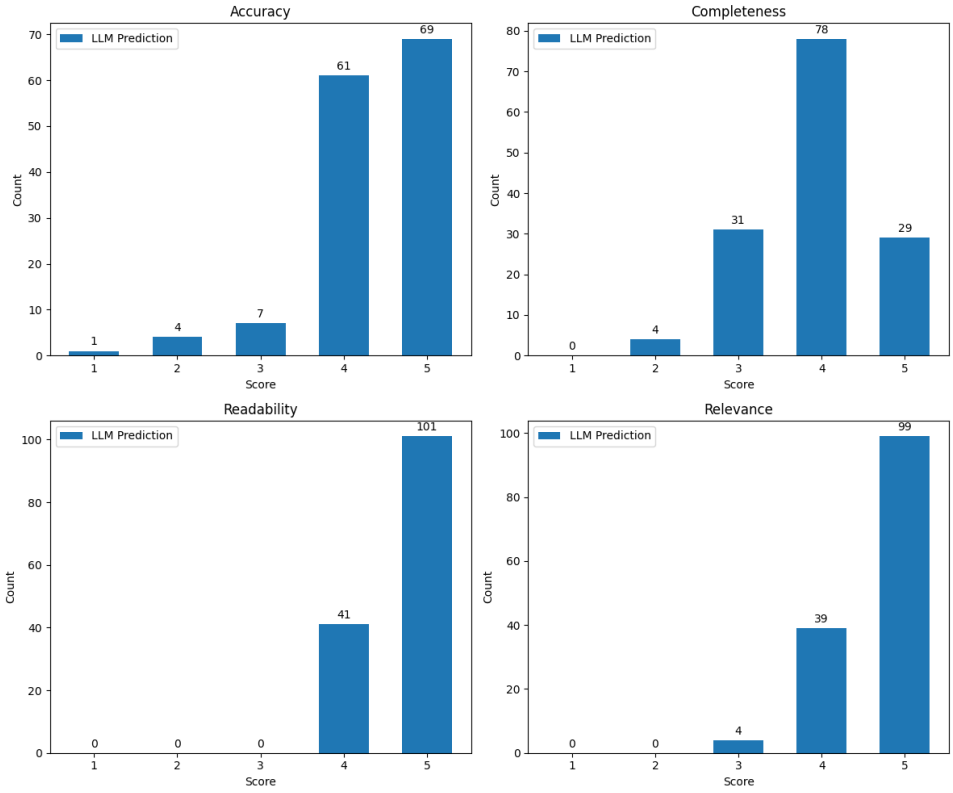

Figura 7 – Distribuição de notas para as docstrings geradas pelo GPT-3.5-Turbo

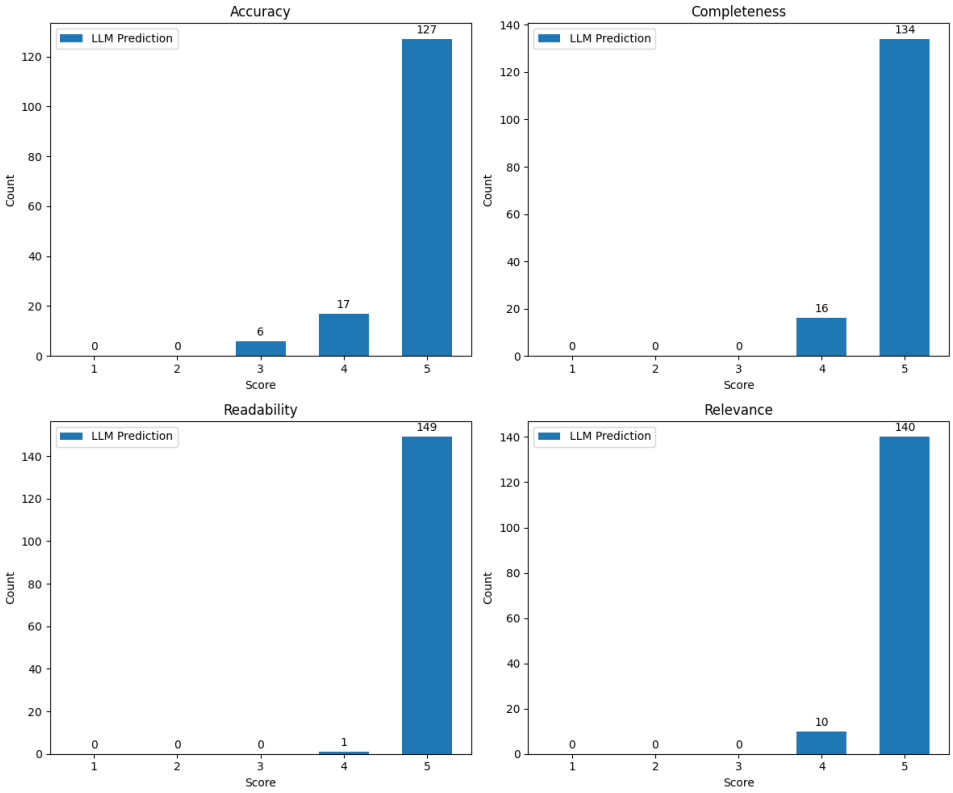

Figura 8 – Distribuição de notas para as docstrings geradas pelo GPT-4o

- Penalização por omissão de informação realmente relevante para o entendimento da função: um caso;

Dessa forma, considerando as justificativas analisadas, apenas nove casos apresentaram falhas que, de fato, justificavam a atribuição de nota inferior a 5.

3.4.2 Correções

O segundo teste teve como objetivo verificar a capacidade do LLM de corrigir *docstrings* com base nas justificativas das avaliações associadas às notas inferiores a 5. Para isso, foi realizada uma nova iteração de geração, na qual as justificativas foram incorporadas ao *prompt*, de modo que o modelo pudesse ajustar apenas os aspectos apontados como problemáticos.

O *prompt* utilizado foi: *"Update the docstring to fix the issues noted in the comments below. Focus only on the mentioned aspects and preserve the correctness of other parts. Respond with the revised docstring only, without any explanations or formatting."*. A essa instrução, foram adicionados o trecho de código, a *docstring* gerada inicialmente e os critérios avaliados negativamente, acompanhados de suas respectivas justificativas.

Inicialmente, a estratégia de correção foi aplicada às *docstrings* geradas com o modelo GPT-3.5-Turbo. Durante a geração original, oito *docstrings* não foram produzidas corretamente e apenas 23 receberam nota máxima em todos os critérios. Dessa forma, 119 *docstrings* apresentaram ao menos uma nota inferior a 5.

A Figura 9 apresenta a nova distribuição das notas após a etapa de correção. De modo geral, observou-se melhora na maioria dos critérios, com exceção de legibilidade, onde houve um aumento na quantidade de avaliações negativas. A análise dos casos com notas significativamente baixas indicou que, em alguns deles, o modelo não retornou uma *docstring* válida, limitando-se a repetir a função original ou a retorná-la acompanhada da *docstring*, o que comprometeu a validação automática posterior.

O mesmo procedimento foi aplicado às *docstrings* geradas com o modelo GPT-4o. A partir das 31 *docstrings* que haviam recebido pelo menos uma nota inferior a 5, foi realizada uma nova iteração de geração com base nas justificativas fornecidas.

A Figura 10 apresenta a distribuição das novas notas atribuídas após a correção. Observa-se uma melhora geral em todos os critérios, embora ainda tenham ocorrido alguns casos de piora. Após essa segunda iteração, apenas 17 *docstrings* continuaram apresentando ao menos

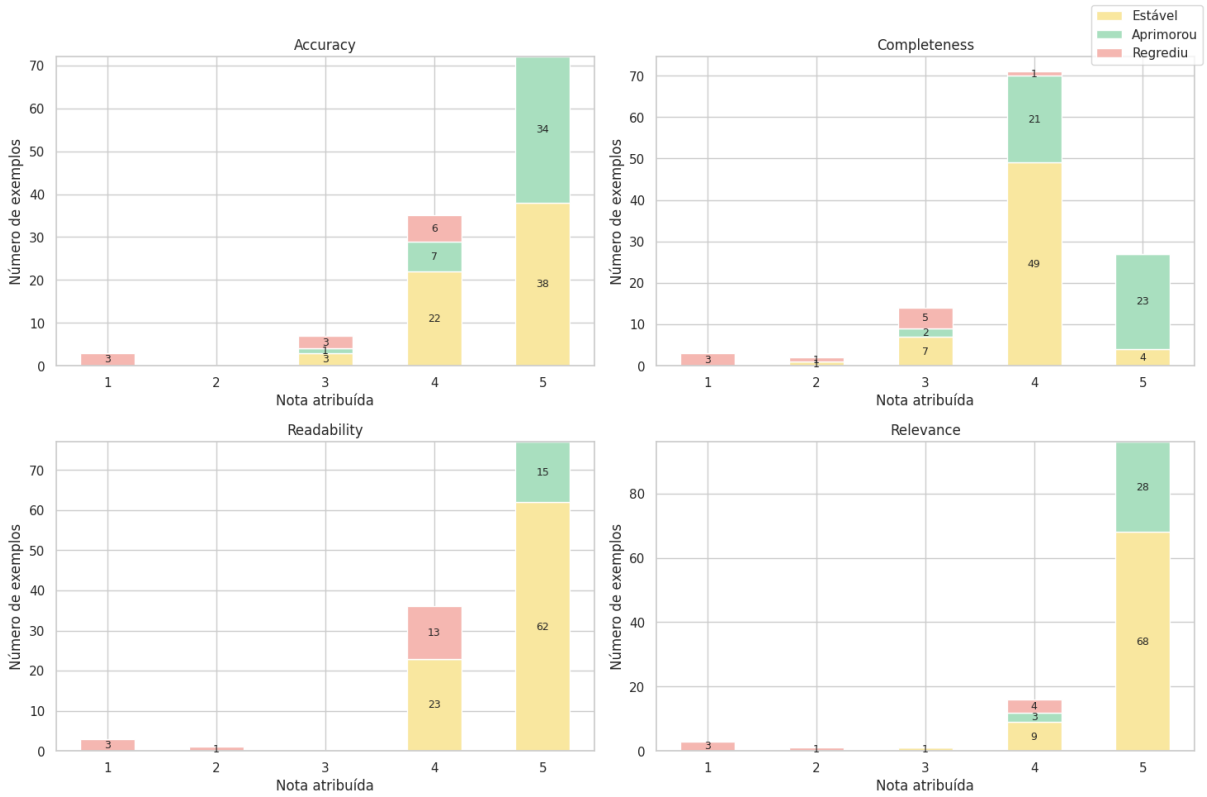

Figura 9 – Distribuição de notas para as docstrings corrigidas pelo GPT-3.5-turbo

uma nota abaixo de 5, sendo que apenas uma delas recebeu nota inferior a 4.

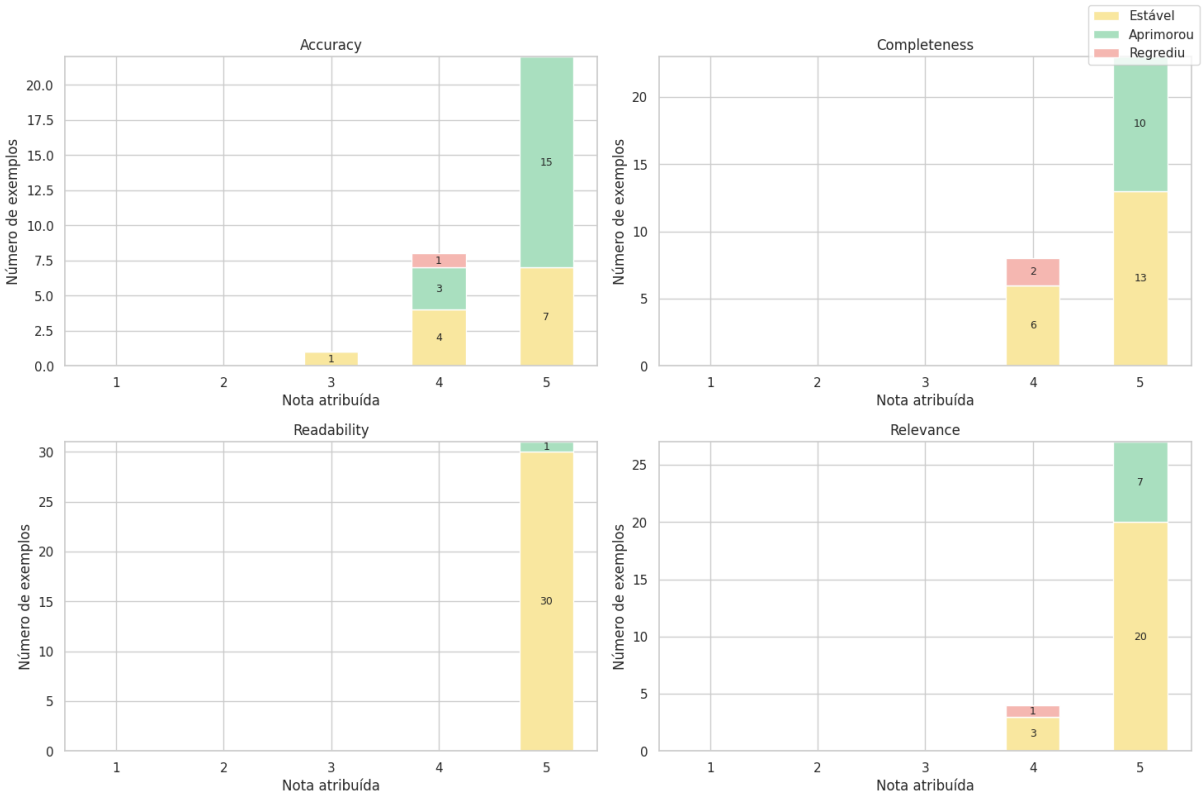

Figura 10 – Distribuição de notas para as docstrings corrigidas pelo GPT-4o

4 DISCUSSÃO

Nesta seção, são apresentados os principais resultados obtidos ao longo do estudo, acompanhados de suas respectivas interpretações.

A análise do coeficiente de Kendall (τ) revelou que os modelos mais recentes, como o GPT-4.1, apresentaram forte correlação com as avaliações humanas, indicando alinhamento consistente com os julgamentos de referência. Em contraste, o GPT-3.5-turbo destacou-se negativamente, com desempenho consideravelmente inferior em relação aos demais.

Dado o desempenho superior do GPT-4.1, este modelo foi adotado como base para a comparação entre as diferentes estratégias de *prompting* avaliadas. A Tabela 10 apresenta um panorama do desempenho do modelo considerando cada uma das abordagens ao longo dos experimentos.

Tabela 10 – Desempenho do GPT-4.1 com as diferentes estratégias

Estratégia	**Completude**		**Acurácia**		**Legibilidade**		**Relevância**	
	%Ex	τ	%Ex	τ	%Ex	τ	%Ex	τ
Simples	44.00	0.709	37.33	**0.668**	50.67	**0.640**	43.33	0.634
Escala	56.67	**0.743**	38.67	0.666	47.33	0.568	**58.67**	**0.755**
Few-shot	**60.54**	0.740	**46.26**	0.630	**52.38**	0.619	53.74	0.690

Considerando tanto os valores de correlação quanto a taxa de acertos exatos, os critérios de completude e relevância apresentaram os melhores resultados. No entanto, é importante destacar que os outros critérios analisados apresentaram correlação forte com a avaliação humana, com coeficientes superiores a 0.6.

A diferença de desempenho entre os critérios pode estar relacionada à natureza dos próprios conceitos. Completude e relevância envolvem aspectos mais objetivos e verificáveis: enquanto a completude está associada à presença ou ausência de informações essenciais, a relevância pode ser inferida pela comparação entre a *docstring* e os elementos já explicitados no código, permitindo avaliar se o comentário realmente agrega valor. Por isso, esses critérios tendem a ser captados de forma mais consistente pelos LLMs.

A legibilidade, por sua vez, envolve múltiplos aspectos da linguagem, como escolha de vocabulário, clareza das frases e organização do conteúdo, cuja avaliação pode variar dependendo da interpretação do modelo ou do avaliador humano.

O critério de acurácia combina elementos factuais, como a veracidade das informações, com aspectos mais sutis de precisão. Em alguns casos, o modelo penalizou *docstrings* por

omissões que, embora não comprometessem diretamente o entendimento, foram interpretadas como falhas relevantes.

Entre as três estratégias avaliadas, o *few-shot prompting* se destacou por apresentar a maior taxa de acertos. No entanto, os coeficientes de correlação foram semelhantes entre todas as abordagens, indicando que, independentemente da estratégia adotada, os modelos foram capazes de distinguir *docstrings* de melhor e pior qualidade de forma consistente. Isso sugere que a definição dos critérios já permite ao modelo captar diferenças relevantes. Ainda assim, para que as notas atribuídas se aproximem mais das avaliações humanas, é necessário nivelar a interpretação dos diferentes níveis de pontuação por meio de exemplos ou escalas explícitas.

A Figura 11 apresenta a distribuição dos erros em relação à diferença entre as notas atribuídas pelo modelo utilizando *few-shot prompting* e aquelas da avaliação humana. Observa-se que a maioria dos erros corresponde a uma diferença de apenas um ponto.

A predominância de erros com diferença de apenas um ponto sugere que os modelos podem ser confiáveis em contextos onde a precisão absoluta não é crítica, como revisão inicial de documentação gerada.

Em relação aos critérios, o modelo se mostrou mais rigoroso nas avaliações de relevância, completude e acurácia, enquanto, no critério de legibilidade, houve uma leve tendência a atribuir notas mais elevadas do que as atribuídas manualmente.

A Figura 12 apresenta a distribuição das notas atribuídas manualmente e das geradas pelo LLM. De modo geral, observa-se uma semelhança entre os dois conjuntos de avaliações. No entanto, nota-se que o modelo tende a ser mais conservador nos critérios de acurácia e relevância, evitando atribuir notas máximas com a mesma frequência que o avaliador humano.

Por fim, vale destacar que, embora as justificativas geradas pelos LLMs tenham sido coletadas durante os experimentos, não foi realizada uma análise aprofundada sobre sua qualidade. Uma avaliação cuidadosa dessas justificativas poderia fornecer informações relevantes para orientar ajustes nos *prompts* e aprimorar a interpretação dos critérios por parte dos modelos. No entanto, essa etapa foi omitida devido às restrições de tempo para a finalização do trabalho.

No que diz respeito à geração de *docstrings*, observou-se que modelos mais avançados, como o GPT-4o, foram capazes de produzir documentações de alta qualidade mesmo a partir de *prompts* simples. Apesar desse desempenho promissor, ainda foram identificados problemas, como omissões de informações relevantes, informações incorretas e, em alguns casos, descrições excessivamente longas.

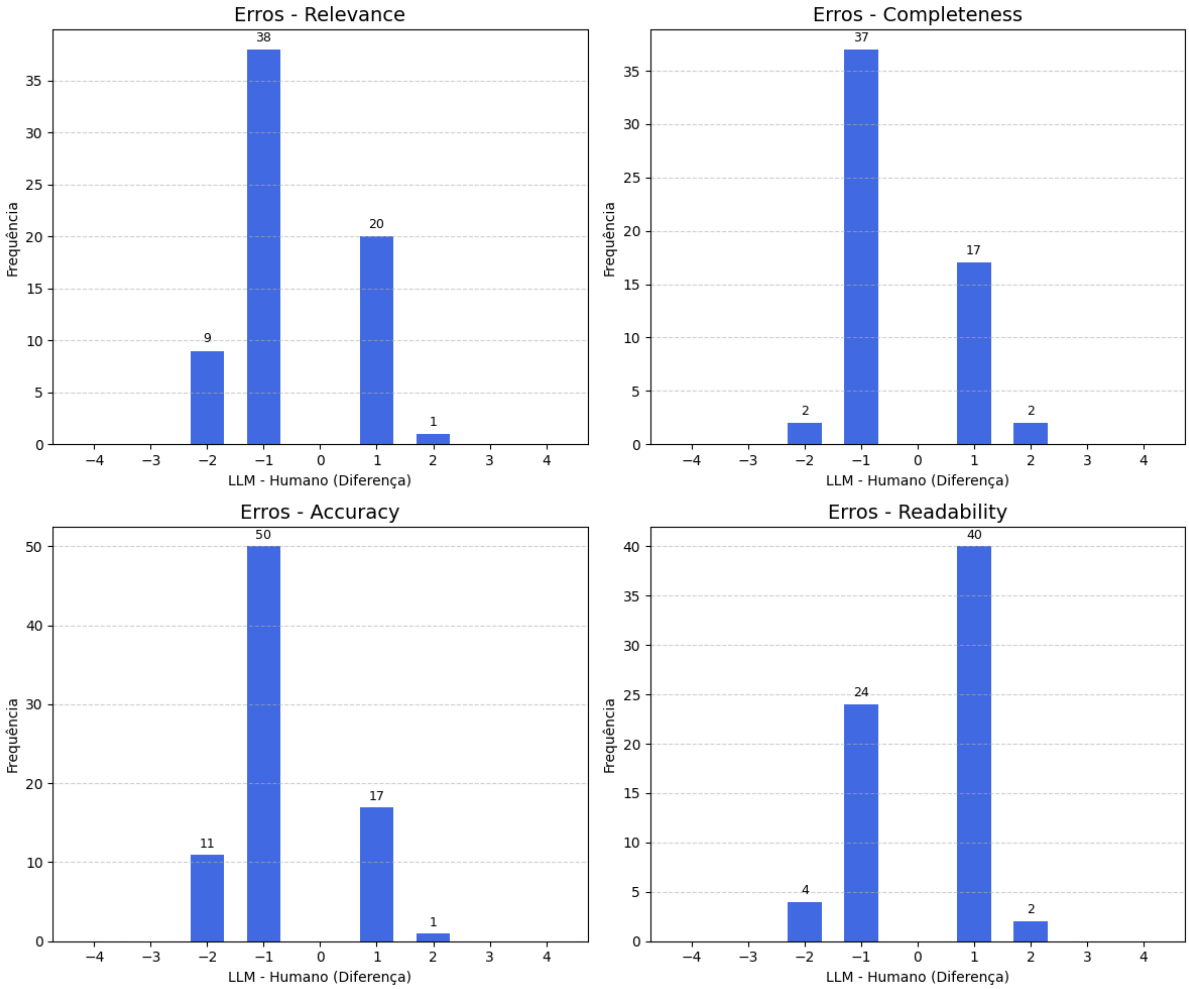

Figura 11 – Distribuição dos erros do modelo GPT-4.1 utilizando few-shot prompting

A análise das justificativas revelou que o LLM tende a penalizar a acurácia sempre que informações não estão explicitamente presentes na *docstring*, ainda que o comentário seja adequado em termos de completude e relevância. Esse rigor é coerente com os resultados quantitativos, que indicaram uma tendência do modelo a atribuir algumas notas ligeiramente mais baixas do que a avaliação humana, geralmente com diferença de um ponto.

A aplicação das correções resultou em uma melhora expressiva nas notas atribuídas, evidenciando a efetividade do processo de refinamento das *docstrings*.

No caso do *GPT-3.5-turbo*, a etapa de correção também resultou em melhorias, com mais 17 *docstrings* atingindo nota máxima em todos os critérios. No entanto, os ganhos foram menos consistentes quando comparados aos obtidos com o GPT-4o.

O modelo demonstrou dificuldade em aplicar corretamente as melhorias indicadas, o que levou à piora de diversas notas e, em alguns casos, à geração de saídas incompletas ou fora do

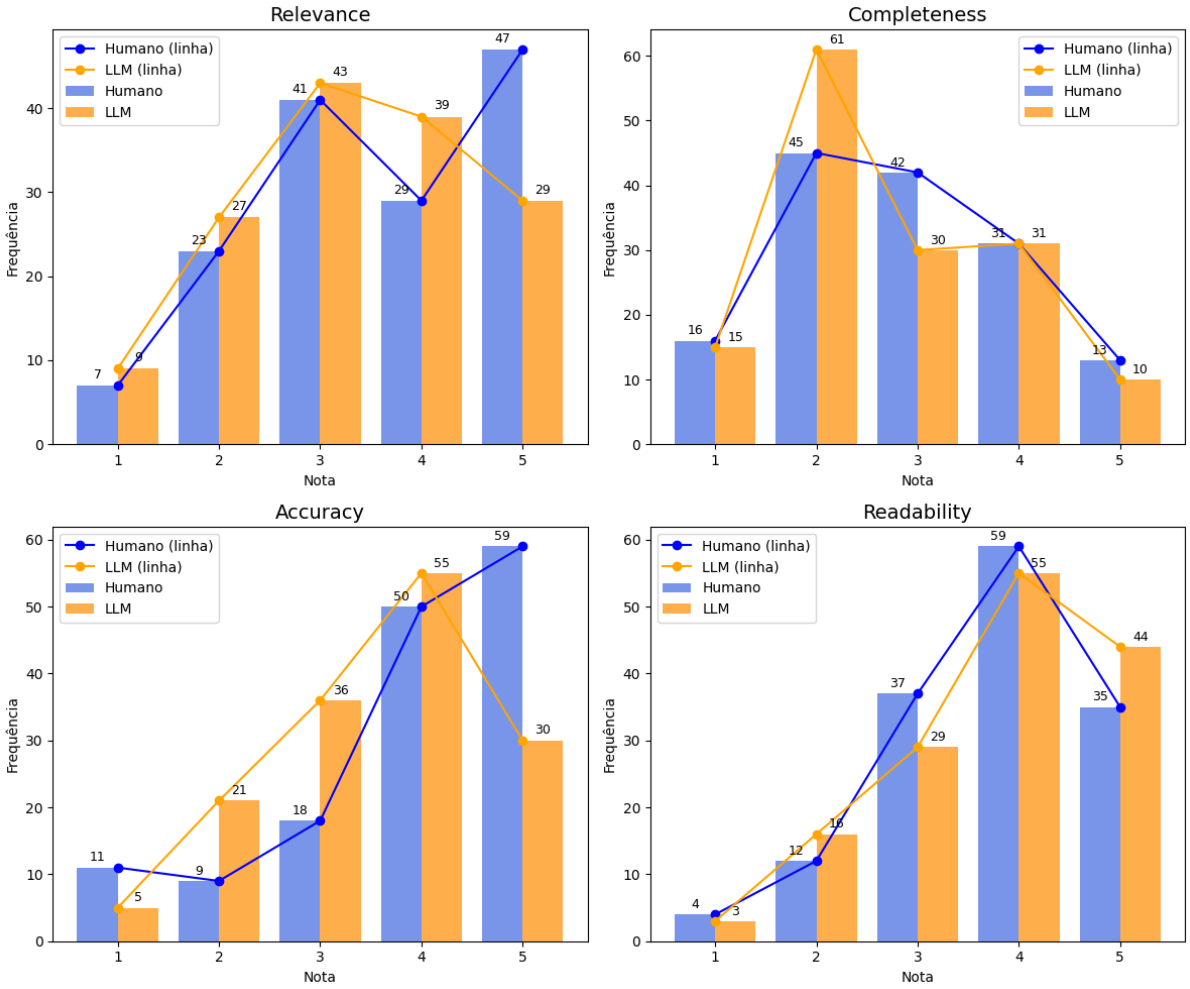

Figura 12 – Distribuição das notas do modelo GPT-4.1 utilizando few-shot prompting

formato esperado. Esses resultados evidenciam as limitações do modelo em tarefas que exigem interpretação refinada de instruções e ajustes pontuais.

Abaixo, comparamos alguns exemplos de *docstrings* geradas pelo GPT-4o que apresentaram melhora, piora ou permaneceram com a mesma nota após a correção.

No Código Fonte 2 a *docstring* se contradiz, diz que a função pode retornar o valor final ou falso, em seguida, ao descrever os parâmetros, informa que a função sempre retorna verdadeiro ou falso, nunca um valor. Esse comentário recebeu uma nota igual a três no quesito acurácia.

Código Fonte 2 – Avaliação com melhoria - antes da correção

```
/**
 * Recursively reads a nested property from an object using an array of property
     segments.
 *
 * This function attempts to access a nested property within an object by
     traversing
```

```
5  * through the object using the provided array of property segments. It checks if
       each
   * segment exists and continues to the next segment until the last segment is
       reached.
7  * If the property exists at each level, it returns the final value; otherwise,
       it returns false.
   *
9  * @param {Object} obj - The object from which the property is to be read.
   * @param {Array<string>} propertySegments - An array of strings representing the
       path to the property.
11 * @param {number} index - The current index in the propertySegments array being
       accessed.
   * @returns {boolean} - Returns true if the property exists and is accessible,
       otherwise false.
13 */
```

Já na versão corrigida, no Código Fonte 3, a contradição é corrigida, agora a descrição da função e do parâmetro descrevem corretamente que a função sempre retorna um booleano. Com todas as informações corretas, o comentário agora recebe nota máxima em acurácia.

Código Fonte 3 – Avaliação com melhoria - depois da correção

```
1  /**
   * Recursively checks the existence of a nested property within an object.
3  *
   * This function navigates through an object using a provided array of property
       segments,
5  * verifying the existence of each segment in sequence. It starts from the
       specified index
   * and continues until the last segment is reached. If all segments exist and are
       accessible,
7  * the function returns true; otherwise, it returns false.
   *
9  * @param {Object} obj - The object from which the property existence is to be
       verified.
   * @param {Array<string>} propertySegments - An array of strings representing the
       path to the property.
11 * @param {number} index - The current index in the propertySegments array being
       accessed.
   * @returns {boolean} - Returns true if the property exists and is accessible at
       each level, otherwise false.
13 */
```

No Código Fonte 4 a *docstring* é penalizada por não documentar um erro não tratado que pode acontecer durante a execução da função: *"...it does not mention that if the parent is not an ancestor of the child, the function may throw an error..."*. Antes da correção o comentário

recebe nota quatro em acurácia.

Código Fonte 4 – Avaliação com piora - antes da correção

```
/**
 * Calculates the cumulative scroll offset of a child element relative to a
    specified parent element.
 *
 * This function traverses up the DOM tree from the child element to the
    specified parent element,
 * summing the scrollTop values of each ancestor element. It returns the total
    scroll offset
 * from the child to the parent.
 *
 * @param {HTMLElement} parent - The parent element up to which the scroll offset
     is calculated.
 * @param {HTMLElement} child - The child element from which the scroll offset
    calculation begins.
 * @returns {number} The total scroll offset from the child element to the parent
     element.
 *
 * @example
 * // Assuming a DOM structure where 'childElement' is nested within '
    parentElement'
 * const scrollOffset = getScrollParent(parentElement, childElement);
 * console.log(scrollOffset); // Outputs the cumulative scroll offset
 */
```

Na versão corrigida, no Código Fonte 5, a *docstring* descreve que o código pode lançar um *TypeError*, dando a entender que o código lança o erro de forma explícita. O problema foi causado pela avaliação da documentação, pois esse erro não precisaria necessariamente ser descrito em uma *docstring*. Depois da correção, a nota em acurácia passou a ser três.

Código Fonte 5 – Avaliação com piora - depois da correção

```
/**
 * Calculates the cumulative scroll offset of a child element relative to a
    specified parent element.
 *
 * This function traverses up the DOM tree from the child element to the
    specified parent element,
 * summing the scrollTop values of each ancestor element. It returns the total
    scroll offset
 * from the child to the parent. If the specified parent is not an ancestor of
    the child,
 * the function will throw an error because it will attempt to access the
    scrollTop property
 * of a null element.
```

```
 *
 * @param {HTMLElement} parent - The parent element up to which the scroll offset
     is calculated.
 * @param {HTMLElement} child - The child element from which the scroll offset
    calculation begins.
 * @returns {number} The total scroll offset from the child element to the parent
     element.
 * @throws {TypeError} Throws an error if the parent is not an ancestor of the
    child.
 *
 * @example
 * // Assuming a DOM structure where 'childElement' is nested within '
    parentElement'
 * const scrollOffset = getScrollParent(parentElement, childElement);
 * console.log(scrollOffset); // Outputs the cumulative scroll offset
 */
```

Por último, analisando um caso em que não houve melhoria, na *docstring* apresentada no Código Fonte 6 o mapeamento das teclas está correto, mas foi avaliado como incorreto.

Código Fonte 6 – Avaliação sem melhoria - antes da correção

```
/**
 * Handles keyboard events to control movement within a plugin interface.
 *
 * This function checks if the current plugin is active and no modifier keys
 * (Alt, Ctrl, Shift, Meta) are pressed. If these conditions are met, it
    processes
 * specific key events to trigger movement actions. The function prevents the
    default
 * behavior of the keys and calls the `moveByKey` method to handle the movement.
 *
 * @param {Object} target - The target object, typically the context in which the
     function is executed.
 * @param {Object} e - The event object representing the keyboard event.
 * @returns {boolean|undefined} - Returns `true` if a movement key is processed,
    otherwise returns `undefined`.
 *
 * Key Mappings:
 * - 'S' (key code 83) and 'A' (key code 65) are mapped to move left or down.
 * - 'W' (key code 87) and 'D' (key code 68) are mapped to move right or up.
 * - Arrow keys are also mapped similarly:
 *   - DOWN (key code 40) and LEFT (key code 37) for moving left or down.
 *   - UP (key code 38) and RIGHT (key code 39) for moving right or up.
 */
```

Após a correção o comentário continuou apresentando o mapeamento correto, mas ainda foi penalizado. É válido destacar que o trecho de código correspondente apresentava comen-

tários em linha incorretos que podem ter impactado como o modelo avaliou a *docstring*.

Código Fonte 7 – Avaliação sem melhoria - depois da correção

```
/**
 * Handles keyboard events to control movement within a plugin interface.
 *
 * This function is designed to manage movement controls within a plugin
    interface by responding to specific keyboard events. It first checks if the
    current plugin is active and ensures that no modifier keys (Alt, Ctrl, Shift
    , Meta) are pressed. If these conditions are satisfied, it processes certain
     key events to trigger movement actions. The function prevents the default
    behavior of these keys and calls the `moveByKey` method to execute the
    movement.
 *
 * @param {Object} target - The target object, typically the context in which the
     function is executed. Note: This parameter is not utilized within the
    function.
 * @param {Object} e - The event object representing the keyboard event.
 * @returns {boolean|undefined} - Returns `true` if a movement key is processed,
    otherwise returns `undefined`.
 *
 * Key Mappings:
 * - 'A' (key code 65), 'S' (key code 83), DOWN (key code 40), and LEFT (key code
     37) are mapped to move in a negative direction (left or down) by calling `
    moveByKey(false)`.
 * - 'W' (key code 87), 'D' (key code 68), UP (key code 38), and RIGHT (key code
    39) are mapped to move in a positive direction (right or up) by calling `
    moveByKey(true)`.
 *
 * Note: The boolean argument passed to `moveByKey` indicates the direction of
    movement:
 * - `false` corresponds to moving left or down.
 * - `true` corresponds to moving right or up.
 */
```

A partir desses três exemplos, nota-se que avaliações incorretas vão resultar em correções ruins, ou até mesmo piorar a qualidade da *docstring*. Isso reforça a importância de realizar a análise qualitativa das avaliações.

5 LIMITAÇÕES

Apesar dos resultados promissores, é fundamental destacar as limitações da etapa de experimentação, que podem afetar a validade das conclusões deste estudo. A principal limitação diz respeito ao processo de avaliação manual, conduzido por apenas uma pessoa. Essa decisão foi motivada por restrições de tempo, dado o esforço necessário para classificar manualmente cada *docstring*. No entanto, a adoção de um único avaliador introduz um viés potencial nas notas de referência, especialmente considerando a natureza subjetiva de critérios como legibilidade e relevância.

Essa limitação torna-se ainda mais relevante no contexto da estratégia de *few-shot prompting*, em que os exemplos fornecidos podem ter refletido o estilo e as interpretações específicas do avaliador humano. Ainda assim, os bons resultados obtidos mesmo nas estratégias baseadas apenas na definição dos critérios, sem exemplos, sugerem que o modelo não se limitou a reproduzir esse viés, mas foi capaz de internalizar a lógica dos critérios propostos.

De toda forma, o fato de o modelo ter conseguido se alinhar com uma avaliação desejada não é, por si só, negativo. Ao contrário, indica que a técnica pode ser ajustada para diferentes interpretações ou estilos de julgamento, desde que os *prompts* e exemplos sejam adaptados adequadamente. Isso reforça o potencial dos LLMs como ferramentas configuráveis para contextos específicos de avaliação.

Outra limitação relevante, ainda que deliberada, refere-se ao escopo do conjunto de dados utilizado, composto exclusivamente por amostras em JavaScript. Essa escolha pode restringir a generalização dos resultados para outras linguagens de programação, sobretudo aquelas com estilos ou paradigmas distintos. No entanto, considerando que os modelos utilizados, todos da família GPT, foram treinados com grandes volumes de dados públicos extraídos da internet, incluindo repositórios em diversas linguagens populares, é razoável supor que seu desempenho se mantenha consistente em linguagens amplamente representadas, como Python, Java e C++.

Ainda assim, essa hipótese precisa ser validada empiricamente, visto que diferenças sutis na estrutura do código, estilo de escrita das *docstrings* e convenções específicas de cada linguagem podem influenciar a avaliação feita pelos modelos. Além disso, a inclusão de modelos distintos, com arquiteturas ou dados de treinamento diferentes, poderia fornecer uma visão mais ampla sobre o comportamento de LLMs frente às mesmas tarefas, permitindo avaliar até que ponto

os achados são específicos da família GPT ou generalizáveis a outras abordagens.

Também se destaca como limitação o fato de que as técnicas de *prompting* aplicadas neste estudo foram propositalmente simples, sem a utilização de abordagens mais sofisticadas, como *fine-tuning* ou estratégias de *chain-of-thought*. Essa escolha buscou favorecer a reprodutibilidade e reduzir a complexidade experimental, mas pode ter limitado o desempenho dos modelos em algumas etapas. Nesse sentido, a aplicação de modelos menores ou mais especializados, aliados a um processo de *fine-tuning* voltado especificamente para a tarefa de avaliação de *docstrings*, poderia resultar em ganhos expressivos de desempenho e maior aderência aos critérios definidos.

Por fim, os *prompts* foram ajustados com base exclusivamente em métricas quantitativas, sem considerar as justificativas textuais fornecidas pelos modelos. Isso limita a compreensão sobre o processo interpretativo dos LLMs: mesmo quando a nota atribuída coincide com a avaliação humana, não é possível saber se o modelo identificou os mesmos aspectos relevantes ou apenas chegou ao mesmo resultado por caminhos distintos. A ausência dessa análise também compromete a detecção de possíveis alucinações, como nos casos em que o modelo penaliza uma *docstring* por um erro inexistente. Investigar as justificativas poderia revelar se as avaliações refletem de fato uma compreensão adequada dos critérios ou se há interpretações equivocadas que passam despercebidas quando se observa apenas a nota final.

Apesar dessas limitações, os resultados obtidos se mantêm válidos no escopo testado e apontam caminhos realistas de adaptação e extensão.

6 CONCLUSÃO

Os resultados deste estudo indicam um potencial promissor no uso de LLMs para a avaliação automatizada de *docstrings*. As análises realizadas sobre funções em *JavaScript* mostraram que modelos mais recentes da família GPT, como o GPT-4.1 e o GPT-4.1-mini, foram capazes de produzir julgamentos alinhados aos de um avaliador humano.

Este trabalho também propôs uma sistematização dos atributos de qualidade mais relevantes na avaliação de documentação. A partir da revisão da literatura e da experimentação prática, concluiu-se que os critérios de completude, acurácia, relevância e legibilidade (ou clareza) são centrais para definir a utilidade de uma *docstring*, cobrindo tanto aspectos objetivos quanto perceptivos do conteúdo.

Embora esses critérios sejam aplicáveis a diferentes tipos de documentação, sua adequação pode variar de acordo com o contexto. Para reproduzir avaliações confiáveis em outros formatos, seria necessário revisar os atributos de qualidade considerados mais apropriados. Ainda assim, a separação por critérios permanece uma abordagem útil e transferível, contribuindo para a compreensão da documentação como um artefato multifacetado, cuja qualidade depende fortemente de sua finalidade e público-alvo.

Ao se considerar estilos de documentação mais amplos, como tutoriais, manuais de API ou guias de desenvolvimento, a forma de avaliação também precisaria ser adaptada. Avaliações eficazes nesses contextos exigiriam que o modelo tivesse acesso ao contexto completo do sistema, incluindo múltiplas funções, classes e até a base de código como um todo. A necessidade de fornecer esse contexto pode impactar o desempenho dos LLMs, especialmente em tarefas de maior escala. Portanto, não é possível generalizar diretamente os resultados obtidos neste estudo para todos os tipos de documentação, sendo necessário aprofundar as investigações para entender como esses modelos se comportam em cenários mais complexos.

A etapa de avaliação foi o foco principal deste estudo. Os experimentos conduzidos com diferentes estratégias de *prompting*, incluindo definições simples, escalas de pontuação e *few-shot prompting*, demonstraram que os modelos conseguem distinguir variações na qualidade das *docstrings* com boa consistência. Os critérios de completude e relevância apresentaram os melhores resultados, com correlações muito fortes. Mesmo em critérios mais subjetivos, como legibilidade e acurácia, a correlação com as avaliações humanas permaneceu forte.

Na etapa de geração, verificou-se que o modelo GPT-4o é capaz de produzir documentações

de alta qualidade a partir de instruções simples, embora ainda sejam observados alguns erros. Por meio da avaliação automatizada com LLMs, foi possível corrigir parte desses problemas, demonstrando a viabilidade de um ciclo de geração e refinamento assistido por modelos.

A análise preliminar das justificativas indicou que os LLMs ainda cometem erros interpretativos, sobretudo em critérios subjetivos. Apesar disso, mostraram-se confiáveis em muitos casos, configurando uma alternativa viável e escalável para revisar e validar *docstrings*. Assim, podem atuar como ferramentas de apoio à geração e avaliação de documentação em cenários onde a supervisão humana ainda é desejável.

As contribuições desse trabalho apontam para aplicações práticas como a integração desses modelos em fluxos de revisão de código, sugestões em tempo real durante a escrita e sua incorporação em pipelines de geração automática de documentação.

Embora os experimentos tenham se concentrado em um tipo específico de artefato, os resultados sugerem que abordagens semelhantes poderiam ser exploradas em outros contextos da engenharia de *software*. Isso inclui cenários em que a documentação textual exerce papel central, como especificações de requisitos, descrições de testes ou comentários de código. No entanto, generalizações ainda exigem investigações adicionais.

Trabalhos futuros devem aprofundar as limitações identificadas neste estudo. Entre os principais pontos, destacam-se a necessidade de avaliações manuais cruzadas, a análise detalhada das justificativas geradas pelos modelos e o uso dessas análises para orientar a correção automática de *docstrings*. Avançar nesse sentido pode viabilizar o desenvolvimento de pipelines mais robustos para geração, avaliação e refinamento de documentação técnica. Dessa forma, este trabalho contribui para aproximar o uso de LLMs das práticas reais de engenharia de software, especialmente no apoio à documentação automatizada.

REFERÊNCIAS

AMIDEI, J.; PIWEK, P.; WILLIS, A. Agreement is overrated: A plea for correlation to assess human evaluation reliability. In: DEEMTER, K. van; LIN, C.; TAKAMURA, H. (Ed.). *Proceedings of the 12th International Conference on Natural Language Generation*. Tokyo, Japan: Association for Computational Linguistics, 2019. p. 344–354. Disponível em: <https://aclanthology.org/W19-8642/>.

ATTIE, P.; OBEIDAT, A.; OH, N.; YELLE, I. *Code Documentation and Analysis to Secure Software Development*. 2024. Disponível em: <https://doi.org/10.48550/arXiv.2407.11934>.

BILLAH, M. M.; RAHMAN, M. S.; ROY, B. *Do Automatic Comment Generation Techniques Fall Short? Exploring the Influence of Method Dependencies on Code Understanding*. 2025. Disponível em: <https://doi.org/10.48550/arXiv.2504.19459>.

CHIANG, C.-H.; LEE, H.-y. Can large language models be an alternative to human evaluations? In: ROGERS, A.; BOYD-GRABER, J.; OKAZAKI, N. (Ed.). *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, 2023. p. 15607–15631. Disponível em: <https://doi.org/10.18653/v1/2023.acl-long.870>.

DIGGS, C.; DOYLE, M.; MADAN, A.; SCOTT, S.; ESCAMILLA, E.; ZIMMER, J.; NEKOO, N.; URSINO, P.; BARTHOLF, M.; ROBIN, Z.; PATEL, A.; GLASZ, C.; MACKE, W.; KIRK, P.; PHILLIPS, J.; SRIDHARAN, A.; WENDT, D.; ROSEN, S.; NAIK, N.; BRUNELLE, J. F.; THAKER, S. *Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation*. 2024. Disponível em: <https://doi.org/10.48550/arXiv.2411.14971>.

DVIVEDI, S. S.; VIJAY, V.; PUJARI, S. L. R.; LODH, S.; KUMAR, D. A comparative analysis of large language models for code documentation generation. In: *Proceedings of the 1st ACM International Conference on AI-Powered Software*. New York, NY, USA: Association for Computing Machinery, 2024. (AIware 2024), p. 65–73. ISBN 9798400706851. Disponível em: <https://doi.org/10.1145/3664646.3664765>.

GEIGER, R. S.; VAROQUAUX, N.; MAZEL-CABASSE, C.; HOLDGRAF, C. The types, roles, and practices of documentation in data analytics open source software libraries. *Computer Supported Cooperative Work (CSCW)*, v. 27, n. 3, p. 767–802, dez. 2018. ISSN 1573-7551. Disponível em: <https://doi.org/10.1007/s10606-018-9333-1>.

GUELMAN, I.; LEAL, A. G.; XAVIER, L.; VALENTE, M. T. *Using Large Language Models to Document Code: A First Quantitative and Qualitative Assessment*. 2024. Disponível em: <https://doi.org/10.48550/arXiv.2408.14007>.

HARGIS, G.; CAREY, M.; HERNANDEZ, A. K.; HUGHES, P.; LONGO, D.; ROUILLER, S.; WILDE, E. *Developing Quality Technical Information: A Handbook for Writers and Editors (2nd Edition)*. USA: Prentice Hall PTR, 2004. ISBN 0131477498.

oES, I. R. d. S. S.; VENSON, E. Evaluating source code quality with large language models: a comparative study. In: *Proceedings of the XXIII Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2024. (SBQS '24), p. 103–113. ISBN 9798400717772. Disponível em: <https://doi.org/10.1145/3701625.3701650>.

PATEL, J. Leveraging language models for code comment classification. In: *Forum for Information Retrieval Evaluation*. Índia: CEUR-WS.org, 2023. ISSN 1613-0073. CEUR Workshop Proceedings, Vol. 3681. Disponível em: <https://ceur-ws.org/Vol-3681/T7-9.pdf>.

PLöSCH, R.; DAUTOVIC, A.; SAFT, M. The value of software documentation quality. In: *2014 14th International Conference on Quality Software*. [s.n.], 2014. p. 333–342. Disponível em: <https://doi.org/10.1109/QSIC.2014.22>.

RANI, P.; BLASI, A.; STULOVA, N.; PANICHELLA, S.; GORLA, A.; NIERSTRASZ, O. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software*, v. 195, p. 111515, 2023. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121222001911>.

SANTOS, J. a.; CORREIA, F. F. A review of pattern languages for software documentation. In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. New York, NY, USA: Association for Computing Machinery, 2020. (EuroPLoP '20). ISBN 9781450377690. Disponível em: <https://doi.org/10.1145/3424771.3424786>.

SCHRECK, D.; DALLMEIER, V.; ZIMMERMANN, T. How documentation evolves over time. In: *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*. New York, NY, USA: Association for Computing Machinery, 2007. (IWPSE '07), p. 4–10. ISBN 9781595937223. Disponível em: <https://doi.org/10.1145/1294948.1294952>.

SILVA, L.; UNTERKALMSTEINER, M.; WNUK, K. Towards identifying and minimizing customer-facing documentation debt . In: *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 72–81. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/TechDebt59074.2023.00015>.

SOUZA, S. C. B. de; ANQUETIL, N.; OLIVEIRA, K. M. de. A study of the documentation essential to software maintenance. In: *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. New York, NY, USA: Association for Computing Machinery, 2005. (SIGDOC '05), p. 68–75. ISBN 1595931759. Disponível em: <https://doi.org/10.1145/1085313.1085331>.

STEIDL, D.; HUMMEL, B.; JUERGENS, E. Quality analysis of source code comments. In: *2013 21st International Conference on Program Comprehension (ICPC)*. [s.n.], 2013. p. 83–92. Disponível em: <https://doi.org/10.1109/ICPC.2013.6613836>.

SUN, W.; MIAO, Y.; LI, Y.; ZHANG, H.; FANG, C.; LIU, Y.; DENG, G.; LIU, Y.; CHEN, Z. *Source Code Summarization in the Era of Large Language Models*. 2024. Disponível em: <https://doi.org/10.48550/arXiv.2407.07959>.

TREUDE, C.; MIDDLETON, J.; ATAPATTU, T. Beyond accuracy: assessing software documentation quality. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1509–1512. ISBN 9781450370431. Disponível em: <https://doi.org/10.1145/3368089.3417045>.

WOODFIELD, S. N.; DUNSMORE, H. E.; SHEN, V. Y. The effect of modularization and comments on program comprehension. In: *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, 1981. (ICSE '81), p. 215–223. ISBN 0897911466. Disponível em: <https://dl.acm.org/doi/10.5555/800078.802534>.

YANG, D.; SIMOULIN, A.; QIAN, X.; LIU, X.; CAO, Y.; TENG, Z.; YANG, G. DocAgent: A multi-agent system for automated code documentation generation. In: MISHRA, P.; MURESAN, S.; YU, T. (Ed.). *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Vienna, Austria: Association for Computational Linguistics, 2025. p. 460–471. ISBN 979-8-89176-253-4. Disponível em: <https://aclanthology.org/2025.acl-demo.44/>.

ZHI, J.; GAROUSI-YUSIFOğLU, V.; SUN, B.; GAROUSI, G.; SHAHNEWAZ, S.; RUHE, G. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, v. 99, p. 175–198, 2015. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121214002131>.

ZHOU, S.; ALON, U.; XU, F. F.; WANG, Z.; JIANG, Z.; NEUBIG, G. *DocPrompting: Generating Code by Retrieving the Docs*. 2022. Disponível em: <https://doi.org/10.48550/arXiv.2207.05987>.

APÊNDICE A – DESCRIÇÃO DA ESCALA DE PONTUAÇÃO

```
# Task
Evaluate the quality of the docstring for the given code based on four criteria: completeness, accuracy, clarity, and relevance.
Each criterion must be rated on a scale from 1 to 5, as described below.

## Evaluation Criteria
Please evaluate the docstring description across the four aspects below.

### Completeness: assesses whether the comment covers all the essential elements of the code.

1 - Incomplete. Does not explain any element of the code.
2 - Very incomplete. Superficially touches on one or more elements.
3 - Partial. Satisfactorily describes some elements.
4 - Nearly complete. Covers almost all key points well.
5 - Complete. Fully addresses all key aspects.

### Accuracy: assesses whether the comment is correct.

1 - Incorrect or lacks sufficient information for validation.
2 - Unreliable. Contains many errors or major omissions.
3 - Partially correct. Mostly accurate but has major omissions.
4 - Nearly accurate. Reliable, but with minor inaccuracies or omissions.
5 - Fully accurate. Precisely reflects the current behavior of the code.

### Clarity: assesses whether the comment is written clearly, directly, and understandably.

1 - Confusing or missing. Vague, poorly written, or disorganized.
2 - Hard to understand. Poorly written, vague, or overly technical.
3 - Understandable, though parts may be vague or too cluttered.
4 - Clear and fluid. Not vague and well written, though it may have minor structural flaws.
5 - Extremely clear. Concise, well-structured and uses simple, effective language.

### Relevance: assesses whether the comment adds practical value to understanding and using the code.

1 - Irrelevant. Merely repeats the code or is off-topic.
2 - Slightly relevant. Contains several redundant or unrelated pieces of information.
3 - Moderately relevant. Somewhat useful, but adds little value and may include unnecessary details.
```

```
4 - Relevant. Offers useful explanations, even if it includes some minor unnecessary details.
5 - Highly relevant. Complements the code with explanations that enhance understanding.

## Output Format
Respond with a valid JSON object using only integers from 1 to 5, following this structure:

{
  "evaluation": {
    "completeness": number,
    "accuracy": number,
    "clarity": number,
    "relevance": number
  }
}

Do not include any markdown formatting, code block markers, or explanation.
```

APÊNDICE B – AVALIAÇÃO INDIVIDUAL POR CRITÉRIO

Código Fonte 8 – Prompt de avaliação de completude

```
# Task
Evaluate the completeness of the provided docstring based on the following criteria.

# Evaluation Criteria - Completeness (1 to 5):
This measures how well the docstring explains the code's key elements: purpose, mechanism, inputs, and outputs.

1 - Missing: No explanation of any element.
2 - Incomplete: Mentions one or two elements superficially.
3 - Partial: Adequately explains some elements, but others are lacking.
4 - Nearly Complete: Covers most essential elements, with minor gaps.
5 - Complete: Thoroughly describes all key aspects.

# Output Format
Respond with a single integer (1 to 5) representing the completeness score. Do not include explanations, comments, or formatting.
```

Código Fonte 9 – Prompt de avaliação de acurácia

```
# Task
Evaluate the accuracy of the provided docstring based on the following criteria.

# Evaluation Criteria - Accuracy (1 to 5):
Assesses whether the comment is correct. It does not penalize for omissions, only for factual mistakes, misrepresentations, or misleading generalizations.

1 - Incorrect or lacks sufficient information for validation.
2 - Largely incorrect. Several inaccuracies or key misrepresentations.
3 - Partially correct. Mostly accurate but has some inaccuracies.
4 - Nearly accurate. Reliable, but with minor inaccuracies or omissions.
5 - Fully accurate. Precisely reflects the current behavior of the code.

# Output Format
Respond with a single integer (1 to 5) representing the accuracy score. Do not include explanations, comments, or formatting.
```

Código Fonte 10 – Prompt de avaliação de clareza

```
# Task
Evaluate the clarity of the provided docstring based on the following criteria.
```

```
# Evaluation Criteria - Clarity (1 to 5):
Assesses how clearly and effectively the docstring communicates its intended meaning.

1 - Very unclear. Confusing, vague, or poorly written.
2 - Hard to follow. Poor written, structure, or overly technical.
3 - Understandable, but may be vague, wordy, or disorganized.
4 - Clear and readable, with minor structural or stylistic issues.
5 - Exceptionally clear. Well-written, concise, and easy to understand.

# Output Format
Respond with a single integer (1 to 5) representing the clarity score. Do not include explanations, comments, or formatting.
```

Código Fonte 11 – Prompt de avaliação de relevância

```
# Task
Evaluate the relevance of the provided docstring based on the following criteria.

# Evaluation Criteria - Relevance (1 to 5):
Assesses whether the comment adds practical value to understanding and using the code. It does not evaluate correctness.

1 - Irrelevant. Merely repeats the code or is off-topic.
2 - Slightly relevant. Contains several redundant or unrelated pieces of information.
3 - Moderately relevant. Somewhat useful, but adds little value and may include unnecessary details.
4 - Relevant. Offers useful explanations, even if it includes some minor unnecessary details.
5 - Highly relevant. Complements the code with explanations that enhance understanding.

# Output Format
Respond with a single integer (1 to 5) representing the relevance score. Do not include explanations, comments, or formatting.
```

APÊNDICE C – PROMPTS FINAIS

Código Fonte 12 – Prompt sem escala de pontuação

```
# Task
Evaluate the quality of the provided docstring for the corresponding code, assigning each of the following criteria a score from 1 (lower) to 5 (highest):

1. Relevance: Is the docstring concise and adds value? It should avoid irrelevant, overly detailed, unrelated information, or information that is obvious from reading the code.
2. Completeness: Does the docstring provide enough information for a developer to understand and use the function? It should cover purpose, inputs, outputs, and any behaviorally relevant aspects.
3. Accuracy: Is the documentation precise, factually correct, and aligned with the actual behavior of the code? Factual errors, vague descriptions of critical behavior, and unverifiable content should be penalized.
4. Readability: Is the text well-structured, well-written, and easy to understand? It should use simple language, be grammatically correct, and follow a logical organization.

Carefully read both the code and the docstring before scoring. Justify each score based only on the evidence provided.
```

Código Fonte 13 – Prompt com escala

```
# Task
Evaluate the quality of the docstring for the given code based on four criteria: completeness, accuracy, readability, and relevance. Each criterion must be rated independently on a scale from 1 to 5.

## Evaluation Criteria

### Completeness: assesses whether the docstring explains all essential elements of the code (purpose, mechanism, inputs, outputs).

1 - Missing. Does not explain any element of the code.
2 - Incomplete. Mentions one or two elements, very briefly.
3 - Partial. Satisfactorily describes some elements.
4 - Nearly complete. Covers most essential points, with minor omissions.
5 - Complete. Fully addresses all key aspects.

### Accuracy: assesses whether the comment is correct. It does not penalize for omissions, only for factual mistakes, misrepresentations, or misleading generalizations.

```

```
1 - Incorrect or lacks sufficient information for validation.
2 - Largely incorrect. Several inaccuracies or key misrepresentations.
3 - Partially correct. Mostly accurate but has some inaccuracies.
4 - Nearly accurate. Reliable, but with minor inaccuracies or omissions.
5 - Fully accurate. Precisely reflects the current behavior of the code.

### Readability: assesses how clearly and effectively the docstring communicates.

1 - Very poor. Disorganized, with grammar issues, fragmented sentences or
    incomplete.
2 - Poor. Difficult to follow due to awkward phrasing, inconsistent structure, or
     overly technical language.
3 - Fair. Generally understandable, but includes vague wording, or minor sentence
    -level issues.
4 - Good. Readable, with small lapses in grammar, style, or organization.
5 - Excellent. Easy to read, fully coherent, with proper grammar, complete
    sentences, and strong structure.

### Relevance: assesses whether the comment adds practical value to understanding
     and using the code. It does not evaluate correctness.

1 - Irrelevant. Merely repeats the code or is off-topic.
2 - Slightly relevant. Contains several redundant or unrelated pieces of
    information.
3 - Moderately relevant. Somewhat useful, but adds little value and may include
    unnecessary details.
4 - Relevant. Offers useful explanations, even if it includes some minor
    unnecessary details.
5 - Highly relevant. Complements the code with explanations that enhance
    understanding.
```

Código Fonte 14 – Prompt com exemplos

```
# Task
Evaluate the quality of the provided docstring for the corresponding code,
    assigning each of the following criteria a score from 1 (lower) to 5 (highest
    ):

1. Relevance: Is the docstring concise and adds value? It should avoid irrelevant
    , overly detailed, unrelated information, or information that is obvious from
     reading the code.
2. Completeness: Does the docstring provide enough information for a developer to
     understand and use the function? It should cover purpose, inputs, outputs,
    and any behaviorally relevant aspects.
3. Accuracy: Is the documentation precise, factually correct, and aligned with
    the actual behavior of the code? Factual errors, vague descriptions of
    critical behavior, and unverifiable content should be penalized.
4. Readability: Is the text well-structured, well-written, and easy to understand
```

```
    ? It should use simple language, be grammatically correct, and follow a
    logical organization.

Carefully read both the code and the docstring before scoring. Justify each score
     based only on the evidence provided.

# Examples
Below are three examples of code, docstrings, and corresponding evaluations.

## Example 1
### Code
function addonsManager_getCategoryId(aSpec) {
  var spec = aSpec || { };
  var category = spec.category;

  if (!category)
    throw new Error(arguments.callee.name + "": Category not specified."");

  return category.getNode().id;
}

### Docstring
Get the ID of the given category element

@param {object} aSpec
Information for getting a category
Elements: category - Category to get the id from

@returns Category Id
@type {string}

{
  "evaluation": {
    "completeness": {
      "score": 4,
      "justification": "The comment addresses most of the relevant aspects but
          omits the possibility that an error may be thrown."
    },
    "accuracy": {
      "score": 5,
      "justification": "All information in the comment is factually accurate and
          consistent with the code. There are no omissions that would impair
          understanding of the function."
    },
    "readability": {
      "score": 4,
      "justification": "The parameter description could be better structured to
```

```
          clarify that 'category' is a property of the object, rather than using
          the phrasing 'Elements: category.'"
    },
    "relevance": {
      "score": 4,
      "justification": "The docstring is concise and focused on the function's
          behavior. However, it could be more relevant by mentioning the
          potential for raised errors, which are part of the function's
          observable behavior."
    }
  }
}

## Example 2
### Code
function makeAddressToAUTFrame(w, frameNavigationalJSexpression)
{
    if (w == null)
    {
        w = top;
        frameNavigationalJSexpression = ""top"";
    }

    if (w == selenium.browserbot.getCurrentWindow())
    {
        return frameNavigationalJSexpression;
    }
    for (var j = 0; j < w.frames.length; j++)
    {
        var t = makeAddressToAUTFrame(w.frames[j], frameNavigationalJSexpression
            + "".frames["" + j + ""]"");
        if (t != null)
        {
            return t;
        }
    }
    return null;
}

### Docstring
construct a JavaScript expression which leads to my frame (i.e., the frame
    containing the window in which this code is operating)

### Evaluation
{
  "evaluation": {
    "completeness": {
```

```
      "score": 2,
      "justification": "The comment briefly states the function's purpose but
          does not describe the parameters. It also lacks important details, such
           as the fact that it starts from the top window if no window is found,
          and how the frame path is structured."
    },
    "accuracy": {
      "score": 4,
      "justification": "While the explanation is high-level and lacks depth, it
          correctly describes the core behavior of the function."
    },
    "readability": {
      "score": 4,
      "justification": "The comment is written in clear and simple language, but
          the explanation of which frame is targeted could be clearer."
    },
    "relevance": {
      "score": 3,
      "justification": "The comment lacks relevant information such as parameter
          descriptions and path structure. Because of these omissions, it is not
          particularly relevant on its own, as the user still needs to read the
          code to fully understand the function."
    }
  }
}


## Example 3
### Code
function count(req, res, next) {
  User.count(function(err, count){
    if (err) return next(err);
    req.count = count;
    next();
  })
}

### Docstring
this approach is cleaner, less nesting and we have the variables available on the
     request object

### Evaluation
{
  "evaluation": {
    "completeness": {
      "score": 1,
      "justification": "The comment highlights the benefits of using the function
```

```
            but does not explain what the function actually does or describe its
            parameters. It lacks all essential information."
        },
128     "accuracy": {
          "score": 2,
130       "justification": "The statement is partially accurate, the variables are
              available on the request object, but it is poorly explained.
              Additionally, claims like 'cleaner approach' and 'less nesting' are
              subjective and unverifiable without a reference point."
        },
132     "readability": {
          "score": 3,
134       "justification": "The comment is written in clear and simple language, but
              its phrasing could be improved."
        },
136     "relevance": {
          "score": 1,
138       "justification": "Although the comment adds information not present in the
              code, it does not help the user understand or use the function.
              Therefore, it is not relevant."
        }
140   }
    }
```

APÊNDICE D – PROMPTS FINAIS - TENTATIVA DE MELHORIA

Código Fonte 15 – Prompt com escala

```
# Task
Evaluate the quality of the docstring for the given code based on four criteria: completeness, accuracy, readability, and relevance. Each criterion must be rated independently on a scale from 1 to 5.

## Evaluation Criteria

### Completeness: assesses whether the docstring explains all essential elements of the code (purpose, mechanism, inputs, outputs).

1 - Missing. Does not explain any element of the code.
2 - Incomplete. Mentions one or two elements, very briefly.
3 - Partial. Satisfactorily describes some elements.
4 - Nearly complete. Covers most essential points, with minor omissions.
5 - Complete. Fully addresses all key aspects.

### Accuracy: assesses whether the comment is correct and precise. If the comment has information that can't be validated against the code or unrelated information, it should be penalized.

1 - Completely incorrect, unrelated to the code, or it is impossible to verify by only checking the code.
2 - Largely incorrect. Several inaccuracies or omissions, can have unverifiable information.
3 - Partially correct. Mostly accurate, but includes minor inaccuracies or major omissions.
4 - Correct. Reliable, with minor omissions that affect precision.
5 - Accurate. Precisely reflects the current behavior of the code.

### Readability: assesses if the comment is easy to read, it assesses only grammar, format, structure and language.

1 - Very Poor. Disorganized, with highly incomplete, fragmented, and confusing sentences.
2 - Poor. Hard to follow, with improper language, confusing or fragmented sentences, or poor structure.
3 - Fair. Generally readable, with long or fragmented sentences and possible minor structural issues.
4 - Good. Readable, with only minor lapses in grammar, wording, or organization.
5 - Excellent. Easy to read and well-organized, with proper grammar and complete, well-formed sentences.

### Relevance: assesses whether the comment adds practical value to understanding
```

```
     and using the code. It does not evaluate correctness.

1 - Irrelevant. Merely repeats the code or is off-topic.
2 - Slightly relevant. Contains several redundant or unrelated pieces of
    information.
3 - Moderately relevant. Somewhat useful, but adds little value and may include
    unnecessary details.
4 - Relevant. Offers useful explanations, even if it includes some minor
    unnecessary details.
5 - Highly relevant. Complements the code with explanations that enhance
    understanding.
```

Código Fonte 16 – Prompt com exemplos

```
# Task
Evaluate the quality of the provided docstring for the corresponding code,
    assigning each of the following criteria a score from 1 (lower) to 5 (highest
    ):

1. Relevance: Is the docstring concise and adds value? It should avoid irrelevant
    , overly detailed, unrelated information, or information that is obvious from
     reading the code.
2. Completeness: Does the docstring provide enough information for a developer to
     understand and use the function? It should cover purpose, inputs, outputs,
    and any behaviorally relevant aspects.
3. Accuracy: Is the documentation precise, factually correct, and aligned with
    the actual behavior of the code? Factual errors, vague descriptions of
    critical behavior, and unverifiable content should be penalized.
4. Readability: Is the text well-structured, well-written, and easy to understand
    ? It should use simple language, be grammatically correct, and follow a
    logical organization.

Carefully read both the code and the docstring before scoring. Justify each score
     based only on the evidence provided.

# Examples
Below are three examples of code, docstrings, and corresponding evaluations.

## Example 1
### Code
function addonsManager_getCategoryId(aSpec) {
  var spec = aSpec || { };
  var category = spec.category;

  if (!category)
    throw new Error(arguments.callee.name + "": Category not specified."");

  return category.getNode().id;
```

```
}

### Docstring
Get the ID of the given category element

@param {object} aSpec
Information for getting a category
Elements: category - Category to get the id from

@returns Category Id
@type {string}

{
  "evaluation": {
    "completeness": {
      "score": 4,
      "justification": "The comment addresses most of the relevant aspects but
          omits the possibility that an error may be thrown."
    },
    "accuracy": {
      "score": 5,
      "justification": "All information in the comment is factually accurate and
          consistent with the code. There are no omissions that would impair
          understanding of the function."
    },
    "readability": {
      "score": 4,
      "justification": "The parameter description could be better structured to
          clarify that 'category' is a property of the object, rather than using
          the phrasing 'Elements: category.'"
    },
    "relevance": {
      "score": 4,
      "justification": "The docstring is concise and focused on the function's
          behavior. However, it could be more relevant by mentioning the
          potential for raised errors, which are part of the function's
          observable behavior."
    }
  }
}

## Example 2
### Code
function makeAddressToAUTFrame(w, frameNavigationalJSexpression)
{
    if (w == null)
    {
```

```
        w = top;
        frameNavigationalJSexpression = ""top"";
    }

    if (w == selenium.browserbot.getCurrentWindow())
    {
        return frameNavigationalJSexpression;
    }
    for (var j = 0; j < w.frames.length; j++)
    {
        var t = makeAddressToAUTFrame(w.frames[j], frameNavigationalJSexpression
            + "".frames["" + j + ""]"");
        if (t != null)
        {
            return t;
        }
    }
    return null;
}

### Docstring
construct a JavaScript expression which leads to my frame (i.e., the frame
    containing the window in which this code is operating)

### Evaluation
{
  "evaluation": {
    "completeness": {
      "score": 2,
      "justification": "The comment briefly states the function's purpose but
          does not describe the parameters. It also lacks important details, such
           as the fact that it starts from the top window if no window is found,
          and how the frame path is structured."
    },
    "accuracy": {
      "score": 4,
      "justification": "While the explanation is high-level and lacks depth, it
          correctly describes the core behavior of the function."
    },
    "readability": {
      "score": 4,
      "justification": "The comment is written in clear and simple language, but
          the explanation of which frame is targeted could be clearer."
    },
    "relevance": {
      "score": 3,
      "justification": "The comment lacks relevant information such as parameter
```

```
          descriptions and path structure. Because of these omissions, it is not
          particularly relevant on its own, as the user still needs to read the
          code to fully understand the function."
    }
  }
}


## Example 3
### Code
function count(req, res, next) {
  User.count(function(err, count){
    if (err) return next(err);
    req.count = count;
    next();
  })
}

### Docstring
this approach is cleaner, less nesting and we have the variables available on the
     request object

### Evaluation
{
  "evaluation": {
    "completeness": {
      "score": 1,
      "justification": "The comment highlights the benefits of using the function
           but does not explain what the function actually does or describe its
          parameters. It lacks all essential information."
    },
    "accuracy": {
      "score": 2,
      "justification": "The statement is partially accurate, the variables are
          available on the request object, but it is poorly explained.
          Additionally, claims like 'cleaner approach' and 'less nesting' are
          subjective and unverifiable without a reference point."
    },
    "readability": {
      "score": 3,
      "justification": "The comment is written in clear and simple language, but
          its phrasing could be improved."
    },
    "relevance": {
      "score": 1,
      "justification": "Although the comment adds information not present in the
          code, it does not help the user understand or use the function.
```

```
          Therefore, it is not relevant."
    }
  }
}

## Example 4
### Code
function render() {
  const DvaContainer = require('./DvaContainer').default;
  ReactDOM.render(
    React.createElement(
      DvaContainer,
      null,
      React.createElement(require('./router').default)
    ),
    document.getElementById('root')
  );
}

### Docstring
render

### Evaluation
{
  "evaluation": {
    "completeness": {
      "score": 1,
      "justification": "The comment merely repeats the function name."
    },
    "accuracy": {
      "score": 1,
      "justification": "While the function does perform a render operation, the
          comment is vague and not precise."
    },
    "readability": {
      "score": 1,
      "justification": "The comment consists of a single word and lacks any
          grammatical or structural form."
    },
    "relevance": {
      "score": 1,
      "justification": "Adds no value beyond the function name."
    }
  }
}

## Example 5
```

```
### Code
function cachePush(value) {
    var cache = this.cache,
        type = typeof value;

    if (type == 'boolean' || value == null) {
      cache[value] = true;
    } else {
      if (type != 'number' && type != 'string') {
        type = 'object';
      }
      var key = type == 'number' ? value : keyPrefix + value,
          typeCache = cache[type] || (cache[type] = {});

      if (type == 'object') {
        (typeCache[key] || (typeCache[key] = [])).push(value);
      } else {
        typeCache[key] = true;
      }
    }
  }

### Docstring
Adds a given value to the corresponding cache object.

@private
@param {*} value The value to add to the cache.

### Evaluation
{
  "evaluation": {
    "completeness": {
      "score": 3,
      "justification": "The comment briefly explains the function's purpose and
          documents the parameter, making it partially complete. It lacks details
           about the parameter format and how different types are handled."
    },
    "accuracy": {
      "score": 4,
      "justification": "The description is correct, but omitting type-specific
          behavior may lead to misuse."
    },
    "readability": {
      "score": 5,
      "justification": "The comment is clear, concise, and well-written."
    },
    "relevance": {
```

```
      "score": 3,
      "justification": "The comment is relevant but provides limited value beyond
          what is evident from the function signature."
    }
  }
}
```