Automatização de Documentação de Código com LLMs Auto-Hospedáveis

Daniel Augusto de Sousa Mendes[1], Ângelo de Carvalho Paulino[2], Angelo Passaro[1]
[1]Instituto de Estudos Avançados (IEAv), São José dos Campos/SP – Brasil
[2] Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos/SP – Brasil

***Resumo* – Este trabalho demonstra um pipeline totalmente local para documentação automatizada de software com Modelos de Linguagem de Grande Escala (do inglês, Large Language Models — LLMs). Usando a interface HyperGPT e o LM Studio como servidor para LLMs Auto-Hospedáveis em recursos computacionais próprios, são geradas *docstrings* em formato Doxygen para códigos em Python e C/C++. As documentações foram avaliadas com base em clareza, precisão e completude. As docstrings foram compiladas em HTML pelo Doxygen além de serem reconhecidas pelo IntelliSense do VS Code, comprovando sua utilidade prática. Resultados mostram que *pipelines* utilizando ferramentas locais podem se igualar ou até mesmo superar serviços em nuvem na geração de *docstrings* de qualidade, preservando sigilo institucional e eliminando custos recorrentes.**

***Palavras-Chave* – Inteligência Artificial, LLMs, Documentação de Código.**

I. INTRODUÇÃO

Nas últimas décadas, a Inteligência Artificial (IA) tem experimentado uma evolução exponencial, impulsionada pelo aprimoramento das arquiteturas de modelos e pela disponibilidade de poder de processamento, além da oferta crescente de grandes volumes de dados. Nesse cenário, os Modelos de Linguagem de Grande Escala (*Large Language Models* — LLMs) se consolidaram como o estado da arte da IA, mostrando elevada eficácia em tarefas de compreensão e geração de linguagem natural. Aplicações como tradução automática, sumarização de textos, sistemas de perguntas e respostas e até mesmo geração de código ilustram esse potencial [1]. Tais modelos também exibem desempenho notável mesmo em cenários de *zero-shot* e *few-shot learning* [2], ou seja, cenários em que nenhum ou poucos exemplos são apresentados. Aplicações amplamente adotadas — como ChatGPT, Gemini, Copilot e DeepSeek — são sustentadas por esses modelos e proporcionam funcionalidades avançadas através de interfaces amigáveis.

No desenvolvimento de software, a documentação é fundamental para garantir a compreensibilidade, a manutenção e a evolução de projetos. Entretanto, sua elaboração e atualização costumam envolver tarefas repetitivas que, com alguma frequência, são negligenciadas pelos desenvolvedores. Esse descuido eleva a curva de aprendizado de novos colaboradores, reduz a confiança de usuários e parceiros no sistema e compromete a manutenção, a colaboração e o aprimoramento contínuo do código. Nessa conjuntura, os LLMs oferecem uma oportunidade de automação capaz de tornar o processo de documentação mais ágil e robusto [3]-[4].

Este trabalho apresenta resultados iniciais da aplicação de LLMs à automação da geração de documentação de códigos genéricos para o Instituto de Estudos Avançados (IEAv). Para tal, foi empregada a interface HyperGPT — desenvolvida internamente ao IEAv, em HTML/JavaScript — que se conecta a um servidor do LM Studio [5] hospedado localmente em uma *workstation* do IEAv. A principal vantagem desse ambiente reside no processamento inteiramente local dos dados, o que assegura que nenhuma informação sensível seja transmitida para fora da instituição.

Neste contexto, este trabalho tem por objetivo investigar a viabilidade de automatizar a documentação de software em um ambiente 100% local, onde tanto os LLMs quanto a interface de interação residem na infraestrutura interna do IEAv. Para isso, (i) comparamos sistematicamente LLMs de uso geral e modelos especializados em programação na tarefa de criar docstrings para códigos Python e C/C++; (ii) desenvolvemos um *prompt* unificado capaz de abranger múltiplas linguagens com um único comando; e (iii) construímos um *script* auxiliar que extrai arquivos-fonte diretamente de repositórios e os converte em PDF, permitindo uma integração transparente ao HyperGPT. A principal contribuição é a entrega de um pipeline replicável e seguro que combina LM Studio, HyperGPT e o script de extração, permitindo gerar documentação técnica de alta qualidade sem expor código sigiloso a serviços externos, ao mesmo tempo em que fornece métricas e procedimentos para avaliar a clareza, a coerência e a robustez das *docstrings* produzidas.

II. FUNDAMENTAÇÃO TEÓRICA

A. Conceitos de LLM e documentação

A documentação de software tem como propósito central facilitar a compreensão, o uso e a manutenção do código-fonte por desenvolvedores — e, eventualmente, por usuários finais. Para atender a esse objetivo, inserem-se blocos de texto denominados *docstrings* diretamente no código, em pontos como funções, classes, módulos e cabeçalhos de arquivos [6]. Esses blocos podem ter diferentes padrões de formatação, como o *reStructuredText*, *Google Style* ou *Doxygen Style*. Normalmente, incluem seções de parâmetros (*Args* ou *Parameters*), valores de retorno (*Returns*) e exceções (*Raises*). Com o código devidamente comentado, ferramentas como Doxygen [7] e Sphinx [8] — conhecidas como geradores de documentação — extraem essas informações e produzem visualizações interativas em HTML que funcionam como manuais de referência do software.

Os LLMs já vêm sendo estudados como instrumentos para a geração automática de documentação [4], [9]-[10]. Esses modelos operam a partir da divisão do texto/código em *tokens* — pequenas unidades linguísticas que podem representar palavras, partes de palavras ou caracteres isolados [2]. Para executar uma tarefa, o LLM recebe um *prompt* (texto de entrada em linguagem natural) que descreve as instruções desejadas.

Diversas soluções exploram esse potencial. Ferramentas como o DocAider [3], AutoDoc [11] e RepoAgent [4], operam

D. A. S. Mendes, adsmendesdaniel@gmail.com; Â. de C. Paulino, angelo-acp@fab.mil.br; A. Passaro, angelopassaro@gmail.com.
Este trabalho foi parcialmente financiado pelo Projeto PROCAD-Defesa 2019 (Edital 15/2019/CAPES), Processo: 88881.387753/2019-01.

em ambientes on-line, integrando-se a plataformas como o Github por meio de uma Interface de Programação de Aplicações (do inglês, *Application Programming Interface* —APIs) de terceiros — por exemplo, os serviços da OpenAI — para produzir documentação automaticamente [3]-[4], [11]. Outras, como "lmdocs" e "doc-comments-ai" [12]-[13], permitem execução totalmente local, empregando modelos hospedados em ambientes como o LM Studio; dessa forma, dispensam conexão com serviços externos e preservam a privacidade do código. A maioria dessas ferramentas foca apenas em Python, embora algumas ofereçam suporte adicional a linguagens como JavaScript, TypeScript e Go, e raramente C/C++.

Na literatura, as metodologias adotadas pelas ferramentas de documentação variam: algumas exigem apenas o caminho do repositório, enquanto outras incorporam *prompts* detalhados, elaborados manualmente. A avaliação da documentação gerada é majoritariamente humana — levando em conta critérios como clareza e precisão técnica — ainda que existam propostas de métricas automáticas ou de comparação com documentação de referência [4], [9]-[10].

No presente estudo, impunha-se documentar códigos em Python e C/C++ sem recorrer a APIs externas nem armazenar o código em repositórios públicos. Nesse sentido, considerou-se o uso do HyperGPT para interagir com o LM Studio. Um exemplo de função em Python documentada com o uso destas ferramentas é mostrado na Fig. 1.

```
def Runge_Kutta_4a_Ordem(dydt, t, y, h):
    """
    @brief Realiza uma iteração do método de Runge-Kutta de 4ª ordem para resolver equações diferenciais ordinárias (EDO).

    O método de Runge-Kutta de 4ª ordem é um método numérico para resolução de EDOs. Ele combina vários esquemas de Euler modificados e é conhecido por sua precisão e estabilidade.

    @param dydt: Função que representa a taxa de mudança da variável dependente em relação à variável independente. Deve ser uma função lambda ou definida como tal.
    @param t: Valor atual da variável independente (geralmente tempo).
    @param y: Valor atual da variável dependente.
    @param h: Tamanho do passo temporal (intervalo de tempo).

    @return: Novo valor da variável dependente após aplicar o método de Runge-Kutta de 4ª ordem.

    @details: O método calcula quatro coeficientes (k1, k2, k3, k4) que são usados para prever a mudança na variável dependente y. Eles são calculados iterativamente com base nos valores atuais de t e y, bem como no tamanho do passo h. Finalmente, o método retorna uma nova estimativa de y usando uma combinação desses coeficientes ponderada pelo fator 1/6.

    @note: Este método é amplamente utilizado em simulações científicas e engenharias devido à sua precisão e eficiência computacional.
    """

    k1 = dydt(t, y)
    k2 = dydt(t + h / 2, y + k1 * h / 2)
    k3 = dydt(t + h / 2, y + k2 * h / 2)
    k4 = dydt(t + h , y + k3 * h)

    return y + (k1 + 2 * k2 + 2 * k3 + k4) * h / 6
```

Fig. 1. Algoritmo de Runge-Kutta de 4ª Ordem documentado por LLM.

III. METODOLOGIA

A. Ambiente computacional e ferramentas

Os experimentos foram conduzidos, majoritariamente, em uma *workstation* Linux (Ubuntu) que hospeda o LM Studio [5]. A máquina dispõe de um processador Intel Xeon Gold 6348 @ 2,6 GHz (56 núcleos) e 1 TB de RAM, recursos suficientes para carregar e servir LLMs de médio porte em execução puramente local.

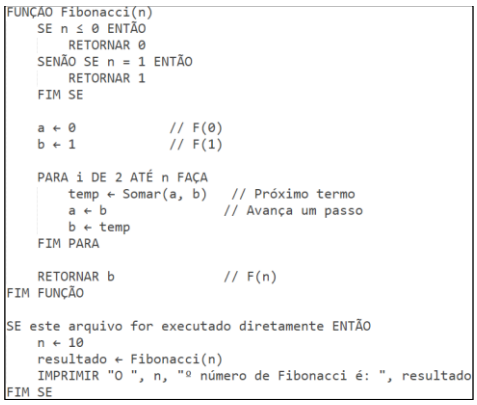
O LM Studio é um agregador gratuito de LLMs de código aberto provenientes do Hugging Face [14]. Uma vez baixado o modelo, ele pode ser acessado remotamente por meio da interface HyperGPT. A versão v0.7.7 (Fig. 2), de 09/04/2025, aceita dois tipos de entrada: texto digitado diretamente ou arquivos PDF. Todos os testes de validação de modelos usaram *prompts* textuais; o envio de PDFs foi utilizado para os testes com códigos completos. Durante a preparação do estudo, contribuiu-se com correções de *bugs* e testes que resultaram na versão atual do HyperGPT.

Fig. 2. Tela do HyperGPT v0.7.7.

B. Conjunto de códigos-exemplo e construção de prompts

Para avaliar e escolher um modelo mais adequado, seria necessário analisar o desempenho sobre um código representativo da aplicação-alvo. Contudo, utilizar diretamente um código-alvo completo — que no caso deste trabalho possui aproximadamente 3.700 linhas (equivalente a mais de 35.000 tokens) — seria impraticável nesta fase inicial, tanto pela complexidade quanto pelo custo computacional. Para contornar essa limitação, foram desenvolvidos trechos sintéticos em Python e C/C++, os quais implementam a sequência de Fibonacci em três níveis de complexidade: (i) **básico**, com uma única função; (ii) **intermediário**, com múltiplas funções, conforme ilustra o pseudocódigo da Fig. 3; e (iii) **avançado**, utilizando orientação a objetos.

Optou-se pelo formato de *docstrings* Doxygen Style, compatível tanto com C/C++ quanto com Python, permitindo-se gerar a documentação HTML com o Doxygen em ambas as linguagens. O *prompt* inicial foi produzido pelo GPT-4o a partir de uma descrição detalhada da tarefa ("*prompt-para-gerar-prompt*"). A primeira versão do *prompt* mostrou-se insuficiente para os efeitos desejados usando modelos locais; Assim, solicitou-se ao próprio GPT-4o que o refinasse iterativamente, resultando em um *prompt* padrão adotado em todos os testes (Fig. 4).



```
FUNÇÃO Fibonacci(n)
    SE n ≤ 0 ENTÃO
        RETORNAR 0
    SENÃO SE n = 1 ENTÃO
        RETORNAR 1
    FIM SE

    a ← 0            // F(0)
    b ← 1            // F(1)

    PARA i DE 2 ATÉ n FAÇA
        temp ← Somar(a, b)   // Próximo termo
        a ← b                // Avança um passo
        b ← temp
    FIM PARA

    RETORNAR b               // F(n)
FIM FUNÇÃO

SE este arquivo for executado diretamente ENTÃO
    n ← 10
    resultado ← Fibonacci(n)
    IMPRIMIR "O ", n, "º número de Fibonacci é: ", resultado
FIM SE
```

Fig. 3. Pseudocódigo (intermediário) usado para testes em Python e C/C++.

```
Você é um assistente para automatizar a documentação de código. Seu objetivo é transformar o código fornecido em um código documentado com comentários e strings de documentação compatíveis com Doxygen, seguindo boas práticas de clareza e organização técnica.

### Instruções gerais:

1. **Insira um cabeçalho de arquivo no topo do código**, no formato Doxygen, com as seguintes informações:
    - @file: nome do arquivo (use um nome genérico se não for fornecido)
    - @brief: descrição geral e concisa do propósito do arquivo
    - @author: (opcional)
    - @date: (opcional)
    - @version: 1.0

2. **Adicione documentação Doxygen para todas as funções e classes**, incluindo:
    - @brief: breve descrição do que a função faz
    - @param: para cada parâmetro, indicando tipo e propósito
    - @return: o que a função retorna
    - @details: (opcional) explicações mais aprofundadas, se necessário
    - @note: (opcional) observações relevantes

3. **Regras para posicionamento da documentação**:
    - Em **Python**, use **docstrings entre `"""` como a primeira linha dentro da função** (não acima da definição).
      Exemplo:
      ```python
      def minha_funcao(x):
          """
          @brief Realiza uma operação.
          @param x Valor de entrada.
          @return Resultado da operação.
          """
      ```
    - Em **C ou C++**, use comentários Doxygen (`/** ... */`) **imediatamente acima da função**.

4. **Adicione comentários explicativos em português** nas partes importantes do código, como:
    - Blocos algorítmicos
    - Estruturas de repetição e decisão
    - Cálculos relevantes ou etapas não triviais

5. **Não altere a lógica ou a estrutura do código original**. Apenas insira documentação e comentários técnicos úteis.

6. **Toda a documentação e comentários devem estar em português do Brasil**.

7. O objetivo é tornar o código facilmente compreensível e pronto para ser processado por geradores de documentação como o Doxygen.

Código:
```

Fig. 4. *Prompt* padrão utilizado nos testes de documentação.

C. Seleção e avaliação de modelos

O HyperGPT conta com diversos modelos disponíveis no LM Studio, abrangendo tanto modelos de uso geral quanto especializados em programação. Entre eles estão o *granite-3.2-8b-instruct*, um LLM open-source da IBM para uso geral; o *Qwen2.5-coder-7b-instruct*, desenvolvido pela Alibaba Cloud; e dois modelos da DeepSeek — *deepseek-r1-distill-qwen-7b@q6_k*, e o *deepseek-coder-v2-lite-instruct*, focado especificamente em tarefas de programação.

Considerando que testes iniciais indicaram uma correlação positiva entre o número de parâmetros e a qualidade da documentação gerada, o conjunto de modelos foi ampliado com versões de maior porte, incluindo mais modelos especializados em programação. Foram incorporados *Qwen2.5-coder-14b-instruct* e *Qwen2.5-coder-32b-instruct*, que são versões mais robustas do modelo *Qwen2.5* utilizado anteriormente, além de *zyh-llm-qwen2.5-14b-v4* e *nxcode-cq-7b-orpo*, ambos projetados especificamente para tarefas de programação. O *GPT-4o* e *GPT-4o-mini* também foram testados por meio do site ChatGPT para que fosse possível traçar um paralelo das ferramentas hospedadas localmente com aquelas acessíveis ao público geral. A Tabela 1 apresenta a consolidação dos modelos avaliados, informando o número de parâmetros (em bilhões), o limite de *tokens* permitido na entrada e na saída, bem como o espaço ocupado em disco após o *download*.

TABELA I. QUANTIDADE DE PARÂMETROS DE CADA LLM, QUANTIDADE MÁXIMA DE TOKENS E ESPAÇO EM DISCO NECESSÁRIO

LLM	***Parâmetros (bilhões)***	***Quantidade máxima de tokens***	***Espaço em disco***
granite-3.2-8b-instruct	8	131.072	6,71 GB
deepseek-r1-distill-qwen-7b@q6_k	7	131.072	6,25 GB
deepseek-coder-v2-lite-instruct	16	163.840	10,36 GB
Qwen2.5-coder-7b-instruct	7	32.768	4,68 GB
GPT-4o	~200	Não informado	Não informado
GPT-4o mini	~8	Não informado	Não informado
Qwen2.5-coder-14b-instruct	14	131.072	8,99 GB
Qwen2.5-coder-32b-instruct	32	131.072	19,85 GB
zyh-llm-qwen2.5-14b-v4	14	1.010.000	8,57 GB
nxcode-cq-7b-orpo	7	65.536	4,41 GB

Os testes foram feitos mesclando-se os códigos ao *prompt* padrão, de modo a serem testados na interface de mensagens do HyperGPT — um código por vez — e sempre utilizando uma nova aba para cada código, a fim de que o histórico das mensagens anteriores não impactasse os resultados. Para cada documentação gerada por LLM, foi feita uma avaliação qualitativa que as classificava em três categorias distintas: "**Ótimo**", quando a documentação gerada poderia ser adicionada ao código original sem modificações; "**Satisfatório**", quando as *docstrings* geradas necessitariam de pequenas modificações manuais para poderem ser adicionadas ao código original, e "**Ruim**" quando os resultados foram muito diferentes do que foi instruído pelo *prompt* padrão e/ou necessitariam de muitas modificações manuais a ponto de comprometer o propósito da automatização de geração de documentação. Além disso, a cada uma das classificações, foi atribuída uma pontuação correspondente: 2 para "**Ótimo**"; 1 para "**Satisfatório**"; e 0 para "**Ruim**". Somando-se os pontos obtém-se a nota global (0 a 6) de cada LLM, viabilizando a comparação de desempenho.

D. Geração da documentação do código-alvo completo

Concatenou-se o *prompt* padrão ao código completo, separando cada código com o delimitador: "--- FILE: nome_do_arquivo.py ---", correspondente a cada arquivo ".py" do repositório. Para facilitar a compreensão dos códigos pelo modelo, eles também foram organizados, aproximadamente, na ordem de sua execução. Orquestrados dessa forma, o *prompt* e os códigos totalizaram exatas 3.797 linhas a serem utilizadas como mensagem de entrada na interface HyperGPT, o que ultrapassa 35 mil tokens, mas que permanece dentro do limite de *tokens* da maioria dos LLMs disponíveis. Subsequentemente os LLMs podem ser empregados para gerar *docstrings* bem

estruturadas, efetivas para a criação da documentação em HTML ou PDF. Em IDEs como o VSCode [15], é possível empregar ferramentas IntelliSense [16], como o "Quick Info" [17], também testada. Assim, o usuário pode obter descrições de funções e variáveis ao posicionar o cursor sobre elas, desde que documentadas corretamente. No caso particular deste trabalho, as descrições serão todas geradas por LLMs.

IV. RESULTADOS E DISCUSSÕES

A. Desempenho dos LLMs: códigos-teste

Seguindo a metodologia descrita na seção anterior, obtiveram-se os resultados sumarizados nas Tabelas 3 e 4. No caso dos códigos em Python, a maioria das saídas foi classificada como satisfatória. As docstrings, em geral, estavam corretas, mas frequentemente exigiam ajustes pontuais, como a inclusão de comentários em trechos-chave ou a adição de *tags* como *@note* e *@details* nos cabeçalhos. Os erros considerados graves, que levaram à classificação ruim, envolveram: (i) uso de sintaxe de comentários própria do C++ em arquivos Python; (ii) quebras de formatação; e (iii) inserção de trechos bilíngues (inglês + chinês) em *docstrings* que deveriam estar exclusivamente em português.

Dentre os modelos, aquele que apresentou a maior quantidade de erros foi o *deepseek-r1-distill-qwen-7b@q6_k*, destacando-se negativamente tanto pela baixa qualidade dos comentários quanto pelo custo computacional elevado. Em contraste, o *deepseek-coder-v2-lite-instruct* se mostrou significativamente mais eficiente, errando apenas um caso relacionado à ausência de aspas na *docstring* do cabeçalho — erro trivialmente corrigível. Este modelo apresentou, portanto, a melhor relação entre qualidade e custo computacional no conjunto avaliado para Python.

TABELA III. RESULTADOS QUALITATIVOS DE DOCUMENTAÇÃO COM OS LLMS UTILIZANDO OS CÓDIGOS TESTE EM PYTHON

LLM	***Básico***	***Intermediário***	***Avançado***	***Pontos***
*granite-3.2-**8b**-instruct*	Ruim	Ruim	Satisfatório	1
*deepseek-r1-distill-qwen-**7b**@q6_k*	Ruim	Ruim	Ruim	0
deepseek-coder-v2-lite-instruct	Ótimo	Satisfatório	Ótimo	5
*Qwen2.5-coder-**7b**-instruct*	Satisfatório	Satisfatório	Satisfatório	3
GPT-4o	Ótimo	Satisfatório	Satisfatório	4
GPT-4o mini	Ruim	Satisfatório	Satisfatório	2
*Qwen2.5-coder-**14b**-instruct*	Satisfatório	Satisfatório	Satisfatório	3
*Qwen2.5-coder-**32b**-instruct*	Satisfatório	Satisfatório	Satisfatório	3
*zyh-llm-qwen2.5-**14b**-v4*	Satisfatório	Ótimo	Ruim	3
*nxcode-cq-**7b**-orpo*	Satisfatório	Satisfatório	Satisfatório	3

Grande parte dos modelos de maior porte, como *Qwen2.5-coder-14b-instruct* e *Qwen2.5-coder-32b-instruct*, entregou documentação adequada, ainda que não tenham sido classificados como "ótimos" por não adicionarem comentários em pontos considerados chave. Uma exceção negativa relevante foi observada no *nxcode-cq-7b-orpo*, que produziu comentários fora de formatação e, mais criticamente, dividiu indevidamente o código Python em dois arquivos ".py" diferentes. Diante dos resultados, o ***deepseek-coder-v2-lite-instruct*** foi considerado o melhor modelo, com 5 pontos.

Reaplicando o mesmo protocolo aos códigos em C/C++ (Tabela 4) observou-se uma tendência geral de melhora nas pontuações em relação às obtidas nos testes com Python. Essa diferença é atribuída ao fato de que o padrão de comentários no estilo Doxygen é historicamente mais frequente e consolidado em bases de código C/C++, o que possivelmente aumentou sua presença nos dados de treinamento dos LLMs. Tal hipótese pode, inclusive, ajudar a explicar o comportamento de alguns modelos que geraram comentários em sintaxe C/C++ mesmo quando aplicados a scripts Python.

Entre os erros recorrentes nos códigos C/C++, destacam-se: ausência de cabeçalhos de documentação, alterações acidentais no código-fonte e falhas de ortografia. No entanto, novamente, o *deepseek-coder-v2-lite-instruct* se destacou, obtendo o máximo possível na avaliação, sendo consistente tanto na qualidade dos comentários quanto na preservação do código original.

Modelos de maior capacidade, como *Qwen2.5-coder-14b-instruct*, *Qwen2.5-coder-32b-instruct* e *zyh-llm-qwen2.5-14b-v4*, também apresentaram bom desempenho, demonstrando maior robustez na geração de documentação e na integridade do código-fonte. A exceção ficou por conta do *nxcode-cq-7b-orpo*, que, nesta linguagem, obteve as piores avaliações, gerando documentação de baixa qualidade e com formatação deficiente — pior ainda que seu desempenho em Python.

TABELA IV. RESULTADOS QUALITATIVOS DE DOCUMENTAÇÃO COM OS LLMS UTILIZANDO OS CÓDIGOS TESTE EM C/C++

LLM	***Básico***	***Intermediário***	***Avançado***	***Pontos***
*granite-3.2-**8b**-instruct*	Satisfatório	Ótimo	Ruim	3
*deepseek-r1-distill-qwen-**7b**@q6_k*	Ótimo	Satisfatório	Ruim	3
deepseek-coder-v2-lite-instruct	Ótimo	Ótimo	Ótimo	6
*Qwen2.5-coder-**7b**-instruct*	Satisfatório	Satisfatório	Satisfatório	3
GPT-4o	Ótimo	Satisfatório	Ótimo	5
GPT-4o mini	Satisfatório	Ótimo	Ótimo	5
*Qwen2.5-coder-**14b**-instruct*	Satisfatório	Ótimo	Satisfatório	**4**
*Qwen2.5-coder-**32b**-instruct*	Ótimo	Ótimo	Satisfatório	**5**
*zyh-llm-qwen2.5-**14b**-v4*	Ótimo	Ótimo	Satisfatório	**5**
*nxcode-cq-**7b**-orpo*	Ruim	Ruim	Ruim	0

De forma geral, os resultados reforçam que modelos especializados ou de maior porte (14B ou 32B) tendem a oferecer melhor equilíbrio entre qualidade da documentação e robustez do código, embora o custo de processamento permaneça um aspecto relevante a ser monitorado.

Adicionalmente, destaca-se que, embora a avaliação qualitativa tenha sido baseada em critérios claros e objetivos, ela carrega inerentemente um grau de subjetividade relativo a um único avaliador humano. Este aspecto pode ser mitigado, como já sugerido na literatura ([4], [9]-[10]), por meio de abordagens como uso de múltiplos avaliadores, listas de verificação (*checklists*) e protocolos de avaliação cega, que representam caminhos recomendados para pesquisas futuras.

B. Desempenho dos LLMs: código-alvo completo

Com a metodologia descrita na Subseção 3.D, foram conduzidos testes de documentação completa de um código-alvo em Python, proprietário do IEAv. Os resultados dos testes anteriores dos LLMs indicaram uma certa correlação entre desempenho e quantidade de parâmetros. Portanto, além do modelo com a melhor pontuação para os códigos-teste (*deepseek-coder-v2-lite-instruct),* foi adicionalmente escolhido o modelo mais robusto disponível, o *Qwen2.5-coder-32b-instruct,* com 32 bilhões de parâmetros, para os testes de documentação completa, a fim de avaliar seu desempenho. O envio do código completo para o LLM foi feito a partir de uma compilação em PDF do código completo, com upload feito via HyperGPT.

A documentação total do código se mostrou uma tarefa de várias horas de processamento. Na versão v0.7.7 do HyperGPT, não foi possível fazer um monitoramento deste tempo com precisão já que a interface apenas indica o horário no momento do recebimento *prompt,* e não do fim da impressão do código comentado. Após a impressão de todo o código pelo HyperGPT, as *docstrings* foram adicionadas manualmente uma a uma, ao código completo. Esta etapa manual visou evitar alguns problemas, como eventuais adições de *docstrings* fora de formatação, modificações indesejadas no código-fonte, além da revisão acurada da parte teórica impressa nas *docstrings*, tais como aquelas que mencionam unidades físicas. Cabe ressaltar que, com o uso do *Qwen2.5-coder-32b-instruct* ("Qwen") e *deepseek-coder-v2-lite-instruct* ("DeepSeek"), **não foi obtida nenhuma *docstring* fora de formatação, nem edições significativas nos códigos**. Ou seja, a documentação completa foi bem-sucedida com o uso de ambos os modelos.

Tomando como exemplo a documentação gerada pelo "Qwen", os arquivos finais alcançaram o total de 4.269 linhas (somando código e *docstrings)*, das quais cerca de 585 foram geradas pelo LLM. O uso de tokens ultrapassou 71.000 tokens — limiar que inviabiliza a utilização de modelos menores ou máquinas com menos recursos. Desse modo e nessas condições, torna-se essencial aperfeiçoar os *prompts* de forma que façam o LLM imprimir apenas as docstrings (sem repetir o código), a fim de "economizar" *tokens*.

TABELA V. DESEMPENHO MÉDIO EM FUNÇÃO DO TEMPO EM NOVE TESTES.

LLM	***Média de Tokens/s***	***Desvio Padrão***	***Tempo médio até o primeiro token (s)***	***Desvio Padrão***
deepseek-coder-v2-lite-instruct	0,759	0,059	3,361	1,095
*Qwen2.5-coder-**32b**-instruct*	0,339	0,038	4,169	1,819

Foram realizadas medidas de desempenho em função do tempo de geração de tokens. Apesar de ambos os modelos terem sido bem sucedidos na documentação do código-alvo completo, algumas diferenças de desempenho merecem destaque: Na Tabela V, vê-se que o modelo "DeepSeek" apresentou um tempo de resposta menor — mais que duas vezes mais rápido — com média de 0,759 tokens/segundo, quando comparado à média de 0,339 tokens/segundo do "Qwen". Além disso, o "DeepSeek" também obteve desempenho superior na geração do primeiro token, isto é, no processamento completo do *prompt* de entrada. Esses dois fatores indicam que nem sempre é mais vantajoso utilizar modelos mais robustos (maior quantidade de parâmetros); a documentação do "Qwen" se ateve a conclusões/observações óbvias como "@brief Calcula o fluxo de massa" para a função "fluxo_de_massa()" enquando o "DeepSeek" redigiu "@brief Calcula o fluxo de massa **a partir do empuxo e impulso específico**", demonstrando uma melhor compreensão das relações entre variáveis e funções. Entretanto, o *deepseek-coder-v2-lite-instruct* mostrou-se mais instável ao gerar a documentação completa, pois eventualmente pausava a geração de tokens aguardando um *feedback* do usuário, comportamento este que não foi observado com o *Qwen2.5-coder-32b-instruct.* A investigação do motivo desse comportamento é sugerida para trabalhos futuros.

C. Integração com Doxygen e IntelliSense

Com os códigos devidamente documentados, exploraram-se recursos de documentação externos. O Doxygen 1.13.2 gerou com sucesso uma interface HTML navegável, validando a consistência sintática das *docstrings*. No Visual Studio Code [15], o plugin IntelliSense — especificamente a funcionalidade Quick Info [17] — recuperou automaticamente as descrições geradas pelos LLMs. Nas Figs. 5 e 6 é possível ver uma aplicação da *feature* "Quick Info" do IntelliSense disponível no VSCode. Ao passar o cursor do mouse sobre uma função (Fig. 5), é possível ver uma descrição contida na documentação produzida pelo LLM. Já na Fig. 6 é possível ver a descrição de uma variável. Note-se que, no caso da descrição da função (Fig. 5) o LLM não se limitou a documentar de maneira genérica a função "*simular()*", mas extraiu intuições de todo o código e inferiu sobre **o que** a função simulava, sem ser instruída sobre do que se tratava o código.

Tais resultados sugerem que LLMs podem não só automatizar tarefas mecânicas, mas também fornecer *insights* adicionais, enriquecendo a base documental do *software*. Ainda assim, as versões iniciais requerem refinamento — por exemplo, inclusão de referências bibliográficas e equações em LaTeX às *docstrings* e criação de um arquivo README que sirva de página inicial para a documentação em HTML.

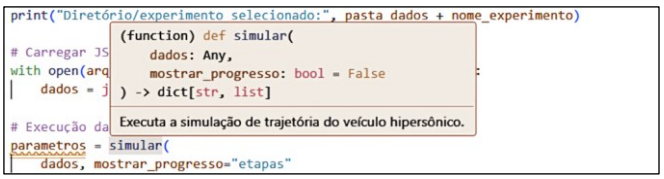


Fig. 5. Detalhe da função *simular*(): o modelo não apenas descreveu argumentos e retornos, mas inferiu o fenômeno físico simulado a partir do contexto global do projeto, acrescentando informações úteis, além do óbvio.

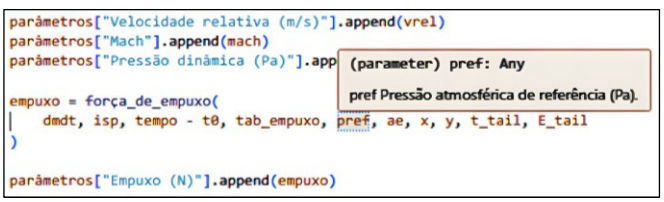


Fig. 6. Descrição de variável interna: demonstra a utilidade imediata de *docstrings* corretas no fluxo de desenvolvimento diário.

Em síntese, os resultados confirmam que: (i) modelos especializados em programação podem entregar melhor qualidade de documentação do que LLMs genéricos de porte semelhante; (ii) até um limite, há impacto da quantidade de parâmetros na qualidade das *docstrings*, embora às custas de maior demanda computacional; e (iii) a integração com ferramentas consagradas, como Doxygen e IntelliSense, potencializa o valor prático da documentação gerada. Investigações futuras devem focar na validação com múltiplos avaliadores, na medição precisa do tempo de execução e em estratégias de *prompt engineering* que reduzam o volume de *tokens* sem prejuízos.

V. CONCLUSÃO

Este estudo se diferencia das soluções já descritas na literatura por deslocar **todo** o processo de geração de documentação Python e C/C++ para um ambiente **100% local**, combinando LLMs de código-aberto executados no LM Studio com a interface proprietária HyperGPT, sem recorrer a APIs em nuvem, repositórios públicos ou serviços de terceiros. Enquanto alguns trabalhos anteriores dependem de integrações on-line (por exemplo, GitHub Actions acopladas à OpenAI) e, portanto, expõem o código-fonte a servidores externos, nossa implementação comprova que é tecnicamente viável produzir *docstrings* em formato *Doxygen Style* — em português e para linguagens distintas (Python e C/C++) — **mantendo sigilo institucional e eliminando custos recorrentes**. A metodologia cobre desde a engenharia de *prompts* até a avaliação comparativa de modelos genéricos e especializados, estabelecendo um pipeline replicável que atende a requisitos de confidencialidade. Os resultados confirmam o potencial de diminuir o esforço humano em tarefas tradicionalmente demoradas, sem expor dados sensíveis a serviços externos.

Os testes comparativos evidenciaram uma correlação positiva entre o número de parâmetros do modelo e a qualidade das *docstrings*. Em particular, o **deepseek-coder-v2-lite-instruct** (16 B) mostrou o melhor desempenho global nos cenários sintéticos, superando inclusive referências em nuvem (GPT-4o/GPT-4o mini). Já na documentação completa de um projeto proprietário de ~3.700 linhas, tanto o **deepseek-coder-v2-lite-instruct** quanto o **Qwen 2.5-coder-32B-instruct** produziram documentações tecnicamente corretas, sem quebras de formatação nem alterações de lógica, mas, o modelo do "DeepSeek" demonstrou uma melhor qualidade nos comentários e maior velocidade de resposta. Estes experimentos comprovam que, mesmo em diante de códigos volumosos, LLMs locais de porte intermediário podem atingir padrões de qualidade satisfatórios — desde que alocados em *hardware* compatível com a demanda.

Embora promissores, os resultados também expuseram limitações práticas. Primeiro, a geração integral da documentação exigiu horas de processamento e intervenção manual posterior para inserir *docstrings* no repositório, eliminar redundâncias e revisar unidades de medida ou terminologias. Ainda, a versão atual da interface HyperGPT (v0.7.7) não registra o tempo efetivo de resposta, dificultando métricas de desempenho sistemáticas. Esses aspectos reforçam a necessidade de métricas automáticas complementares e de instrumentação mais fina do pipeline.

Como desdobramentos futuros, recomenda-se: (i) implementar chamadas diretas à API do LM Studio para eliminar o estágio manual de conversão para PDF e permitir *streaming* de *docstrings* diretamente para o repositório; (ii) adotar *prompt engineering* focado em suprimir a reprodução do código original, reduzindo significativamente a contagem de *tokens*; (iii) desenvolver um arquivo de configuração padrão do Doxygen, garantindo uniformidade de estilo para todos os projetos de interesse; e (iv) disponibilizar um conjunto de códigos de referência, cuidadosamente documentados por especialistas, para que um LLM possa aprender os padrões do estilo documental antes de executar a tarefa em larga escala.

Ao mostrar que LLMs locais podem gerar documentação de alta qualidade, mantendo a confidencialidade dos dados e integrando-se facilmente a ferramentas consagradas, estabelece-se uma base para a adoção institucional, inclusive para códigos de interesse de outras instituições. Contudo, a plena incorporação ao ciclo de desenvolvimento exigirá refinamento adicional dos *prompts*, ampliação da avaliação humana e otimização de desempenho — desafios que configuram uma agenda fértil para pesquisas futuras.

REFERÊNCIAS

[1] B. Wei, "Requirements Are All You Need: From Requirements to Code With LLMs," arXiv, Cornell University, Jun. 14, 2024.

[2] T. Brown et al., "Language Models Are Few-Shot Learners," arXiv, Cornell University, v. 4, May 28, 2020.

[3] J. Dias et al., "DocAider: Automated Documentation Maintenance for Open-source GitHub Repositories," Microsoft Tech Community Blog, 2024. Disponível em: <https://techcommunity.microsoft.com/blog/educatordeveloperblog/docaider-automated-documentation-maintenance-for-open-source-github-repositories/4245588>. Acesso em: 06/05/2025.

[4] Q. Luo et al., "RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation," arXiv, 2024. Disponível em: <https://arxiv.org/abs/2402.16667>. Acesso em: 06/04/2025.

[5] LM Studio Team, "LM Studio – Discover and run local LLMs," 2025. Disponível em: <https://lmstudio.ai/>. Acesso em: 06/05/2025.

[6] Doxygen, "Documenting the Code," 2025. Disponível em: <https://www.doxygen.nl/manual/docblocks.html>. Acesso em: 16/06/2025.

[7] Doxygen, "Doxygen: Doxygen," 2025. Disponível em: <https://www.doxygen.nl/>. Acesso em: 09/05/2025.

[8] Sphinx, "Overview — Sphinx documentation," 2025. Disponível em: <https://www.sphinx-doc.org/en/master/index.html>. Acesso em: 09/05/2025.

[9] S. Dvivedi et al., "A Comparative Analysis of Large Language Models for Code Documentation Generation," 2023. Disponível em: <https://arxiv.org/abs/2312.10349>. Acesso em: 13/05/2025.

[10] A. Zhao, "Evaluating an LLM Code Documentation Generation Application," 2025. Disponível em: <https://medium.com/gft-engineering/evaluating-an-llm-code-documentation-generation-application-719b57f801e5>. Acesso em: 13/05/2025.

[11] Context-Labs, "GitHub – context-labs/autodoc: Experimental toolkit for auto-generating codebase documentation using LLMs," 2023. Disponível em: <https://github.com/context-labs/autodoc>. Acesso em: 09/05/2025.

[12] MananSoni42, "GitHub – MananSoni42/lmdocs: Generate python documentation using LLMs," 2025. Disponível em: <https://github.com/MananSoni42/lmdocs>. Acesso em: 09/05/2025.

[13] Fynnfluegge, "GitHub – fynnfluegge/doc-comments-ai: LLM-powered code documentation generation," 2025. Disponível em: <https://github.com/fynnfluegge/doc-comments-ai>. Acesso em: 09/05/2025.

[14] Hugging Face, "Hugging Face – On a mission to solve NLP, one commit at a time," 2025. Disponível em: <https://huggingface.co/>. Acesso em: 06/05/2025.

[15] Microsoft, "Visual Studio Code," 2025. Disponível em: <https://code.visualstudio.com/Download>. Acesso em: 07/05/2025.

[16] Microsoft, "IntelliSense – Visual Studio Code," 2025. Disponível em: <https://code.visualstudio.com/docs/editing/intellisense>. Acesso em: 07/05/2025.

[17] TERRYGLEE et al., "Parameter info, list members, and quick info in IntelliSense – Visual Studio (Windows)," 2022. Disponível em: <https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022>. Acesso em: 07/05/2025.