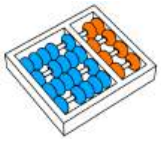


Avaliação de LLMs na Conversão de Fluxogramas para Código

M. G. Lozano *A. Santanchè* *P. D. P. Costa*

Relatório Técnico - IC-PFG-25-56
Projeto Final de Graduação
2025 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Avaliação de LLMs na Conversão de Fluxogramas para Código

Matheus Gasparotto Lozano André Santanchè*
Paula Dornhofer Paro Costa†

Resumo

A placa educacional BitDogLab visa democratizar o ensino de conceitos *STEAM* (*Science, Technology, Engineering, Arts, and Mathematics*) utilizando fluxogramas para mitigar barreiras sintáticas em programação. Este trabalho propõe o desenvolvimento e a validação de um assistente inteligente capaz de interpretar esses diagramas visuais e convertê-los automaticamente em código funcional (MicroPython), atuando como tutor virtual. O objetivo central consistiu na realização de um estudo comparativo (*benchmarking*) para identificar os Grandes Modelos de Linguagem (LLMs) e Multimodais (LMMs) mais aptos a operar nesta ferramenta, considerando restrições de *hardware* local (GPUs de consumo com 24 GiB de VRAM). Para a avaliação, foi desenvolvido integralmente um *dataset* proprietário "padrão-ouro" composto por 20 triplas alinhadas (fluxograma, pseudocódigo e código), cobrindo exaustivamente os periféricos da placa. A metodologia adotou métricas qualitativas de quatro níveis para analisar tanto a fidelidade da interpretação visual quanto a corretude funcional do código gerado. Os experimentos demonstraram que, para a etapa de visão, o modelo Qwen3-VL-Instruct (8B) apresentou desempenho superior na compreensão de topologia e OCR. Na etapa de codificação, o modelo NextCoder-14B destacou-se como a escolha ótima, oferecendo um equilíbrio crítico entre precisão sintática e eficiência de memória, em contraste com modelos maiores como o GPT-OSS-20B. Conclui-se que a orquestração híbrida entre Qwen3-VL e NextCoder viabiliza a execução simultânea dos agentes em ambiente local, proporcionando uma solução robusta para o ensino de lógica de programação.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
†Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, 13083-852 Campinas, SP

Sumário

1 Introdução

O cenário educacional brasileiro enfrenta desafios significativos no ensino de *STEAM* (*Science, Technology, Engineering, Arts, and Mathematics*). Barreiras como disparidades socioeconômicas, infraestrutura escolar inadequada e metodologias de ensino baseadas na memorização acabam por desestimular estudantes, tornando tais disciplinas pouco atrativas [1]. Como resposta a esse contexto, foi desenvolvida a BitDogLab, uma placa de *hardware* aberta e de baixo custo projetada para democratizar o acesso à tecnologia e ensinar conceitos de engenharia de forma prática e interativa [2]. A eficácia pedagógica da plataforma reside na tangibilidade do aprendizado: ao interagir fisicamente com atuadores (como matrizes de LED e *buzzers*) e sensores (botões, microfones), o estudante recebe *feedback* sensorial imediato sobre a execução de seus algoritmos, tornando o processo cognitivo mais concreto e engajante. Outro pilar pedagógico fundamental da BitDogLab é a mitigação da barreira sintática das linguagens de programação tradicionais. Para isso, incentiva-se o uso de fluxogramas como ferramenta introdutória, permitindo que o aluno desenvolva o raciocínio lógico e algorítmico visualmente antes de se preocupar com a codificação textual [3, 4].

Paralelamente aos avanços no *hardware* educacional, a inteligência artificial vive uma mudança de paradigma com a popularização dos Grandes Modelos de Linguagem (Large Language Models - LLMs). Baseados na arquitetura *Transformer* [5] e pré-treinados em vastos corpora textuais, esses modelos demonstram capacidades avançadas em tarefas de Processamento de Linguagem Natural (PLN) [6]. Uma aplicação de destaque é a geração automática de código, exemplificada por ferramentas como GitHub Copilot [7] e Cursor [8], que prometem elevar a produtividade no desenvolvimento de software. Entretanto, a adoção dessas ferramentas altera o papel do programador, que passa a atuar mais como um revisor do que como um escritor de código, o que levanta preocupações sobre a qualidade e a segurança das soluções geradas, especialmente quando utilizadas por aprendizes [9].

A interseção entre a metodologia visual da BitDogLab e a capacidade generativa dos LLMs oferece uma oportunidade para o ensino de programação: a criação de um assistente inteligente capaz de converter automaticamente os fluxogramas desenhados pelos alunos em código funcional para a placa BitDogLab. Dessa forma, o objetivo central deste trabalho não é apenas o desenvolvimento da ferramenta em si, mas a identificação e validação dos modelos de inteligência artificial mais adequados para compô-la. O estudo propõe uma avaliação comparativa (*benchmarking*) de diversos Modelos de Linguagem e Modelos Multimodais (LMMs), analisando seu desempenho na interpretação de diagramas visuais e na geração de código específico para sistemas embarcados. Busca-se, assim, determinar qual arquitetura de modelo oferece o melhor equilíbrio entre corretude lógica, adesão às restrições de *hardware* da BitDogLab e capacidade de instrução, viabilizando uma experiência de aprendizado confiável.

Como resultado principal, os experimentos evidenciaram que a orquestração de modelos especializados de médio porte supera as limitações de arquiteturas monolíticas em ambientes locais. Especificamente, a combinação entre o modelo visual Qwen3-VL e o gerador de código NextCoder-14B revelou-se a configuração ótima, garantindo a precisão na conversão dos artefatos sem exceder o orçamento de memória disponível, validando a viabilidade técnica da solução proposta.

O restante deste relatório está organizado da seguinte forma: a Seção 2 apresenta a fundamentação técnica, descrevendo as especificidades do *hardware* da BitDogLab e a arquitetura multiagentes da ferramenta. A Seção 3 detalha a metodologia experimental, incluindo a construção do *dataset* proprietário, os critérios de seleção e avaliação dos modelos e o ambiente de execução (Ollama e DSPy). A Seção 4 expõe os resultados quantitativos de desempenho e consumo de recursos computacionais e discute qualitativamente os padrões de erro e acerto observados, analisando as falhas de percepção visual e alucinações de *hardware*. Por fim, a Seção 5 sintetiza as conclusões e aponta direções para trabalhos futuros na expansão do *dataset* e refinamento do *pipeline* de inferência e a Seção 6 indica onde o código desenvolvido ao longo do projeto pode ser encontrado.

2 Cenário de Aplicação

O objetivo central deste estudo é identificar os modelos mais adequados para compor um assistente virtual capaz de gerar código a partir de fluxogramas fornecidos pelo usuário. A escolha desse modelo depende diretamente das características do *hardware* alvo e da estrutura do sistema proposto. Assim, a Seção 2.1 descreve o *hardware* BitDogLab e a Seção 2.2 detalha a arquitetura multiagentes proposta para a ferramenta, estabelecendo os requisitos técnicos que guiarão os testes realizados a seguir.

2.1 *Hardware* Alvo

A BitDogLab [10] é uma placa educacional baseada no microcontrolador Raspberry Pi Pico (nas versões H ou W), projetada para facilitar o aprendizado de programação e eletrônica. A placa possui diversos componentes essenciais para desenvolvimento de projetos interativos, sendo que os principais são 1 LED RGB, um *dislay* OLED com dimensões de 128x64 pixels, uma matriz 5x5 de LEDs, 1 microfone, 1 *joystick* analógico, dois *buzzers* e dois botões.

Para sua programação, a BitDogLab utiliza o MicroPython [11], uma implementação da linguagem de programação Python otimizada para execução em microcontroladores e sistemas embarcados, o que permite uma programação fácil e eficiente. A Figura 1 mostra o lado da frente da BitDogLab, onde os componentes relevantes para este projeto estão localizados.

2.2 Arquitetura Multiagentes do Assistente Virtual

Sistemas multiagentes (SMA) são um paradigma proposto por El Fallah Seghrouchni, Florea e Olaru [12] no qual diversos agentes com propósitos especializados colaboram para atingir um objetivo comum. Em SMAs, agentes podem ser programados para executar tarefas específicas e interagir entre si, de forma que tais sistema se destacam em ambientes em que as tarefas podem ser distribuídas entre agentes com especialidades variadas [13]. No contexto do projeto, o uso de tais sistemas é relevante à medida que o objetivo proposto pode ser facilmente decomposto nas tarefas de interpretação do fluxograma e geração do código, possibilitando a atuação coordenada de agentes especializados durante a interação com o usuário.

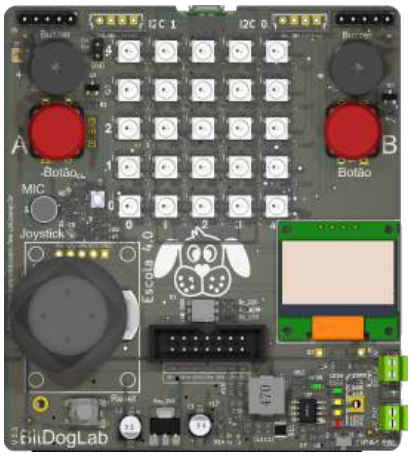


Figura 1: Frente da placa de *hardware* BitDogLab, versão 5.3

Ao realizar a implementação dos agentes, uma preocupação central foi garantir que o sistema fosse agnóstico a LLM. Conforme descrito por O'Neill [14], uma abordagem agnóstica a LLM significa construir um sistema de inteligência artificial sem depender de um modelo ou provedor específico, de forma que seja possível trocar ou adicionar diferentes modelos sem a necessidade de reescrever toda a infraestrutura. Considerando a alta frequência com que novos modelos, muitas vezes superiores aos existentes, são lançados por diferentes empresas e grupos de pesquisa, tal abordagem permite maior flexibilidade e proteção ao futuro, uma vez que é possível adotar novos modelos conforme surgem sem grande impacto na arquitetura existente.

O sistema é composto por dois agentes principais: um denominado "Leitor de Fluxograma", que recebe como entrada um arquivo de imagem contendo um fluxograma e retorna o pseudocódigo correspondente, e outro denominado "Gerador de Código", que recebe o pseudocódigo gerado e fornece como saída sua implementação em MicroPython. Além disso, também foi definido um agente "Coordenador", responsável por instanciar os outros agentes e definir, a partir da entrada recebida, qual deve ser invocado para processar tal entrada, controlando o fluxo de execução. Ao adicionar esta camada de abstração, a arquitetura escolhida permite uma abordagem agnóstica a LLM à medida que, para alterar o modelo utilizado, basta substituir a configuração do agente leitor de fluxograma pelo coordenador. O esquema geral da arquitetura do sistema é mostrado na Figura 2.

A ferramenta possui dois casos de uso principais: além da interpretação do fluxograma e geração do código, o usuário tem a opção de fornecer diretamente o pseudocódigo, ou mesmo um *prompt* em linguagem natural. Caso o usuário opte por enviar um arquivo de imagem, o agente leitor de fluxograma é chamado para interpretar a imagem e o pseudocódigo gerado é retornado para o usuário para validação. Se houver algum trecho incorreto, o usuário pode realizar alterações nesse pseudocódigo e, quando validar as alterações, o agente gerador de código é acionado para implementar o pseudocódigo em MicroPython. Por outro lado, caso o usuário escolha digitar o pseudocódigo ou envie um *prompt* em linguagem natural, apenas o agente gerador de código é executado e o código correspondente é gerado diretamente. A

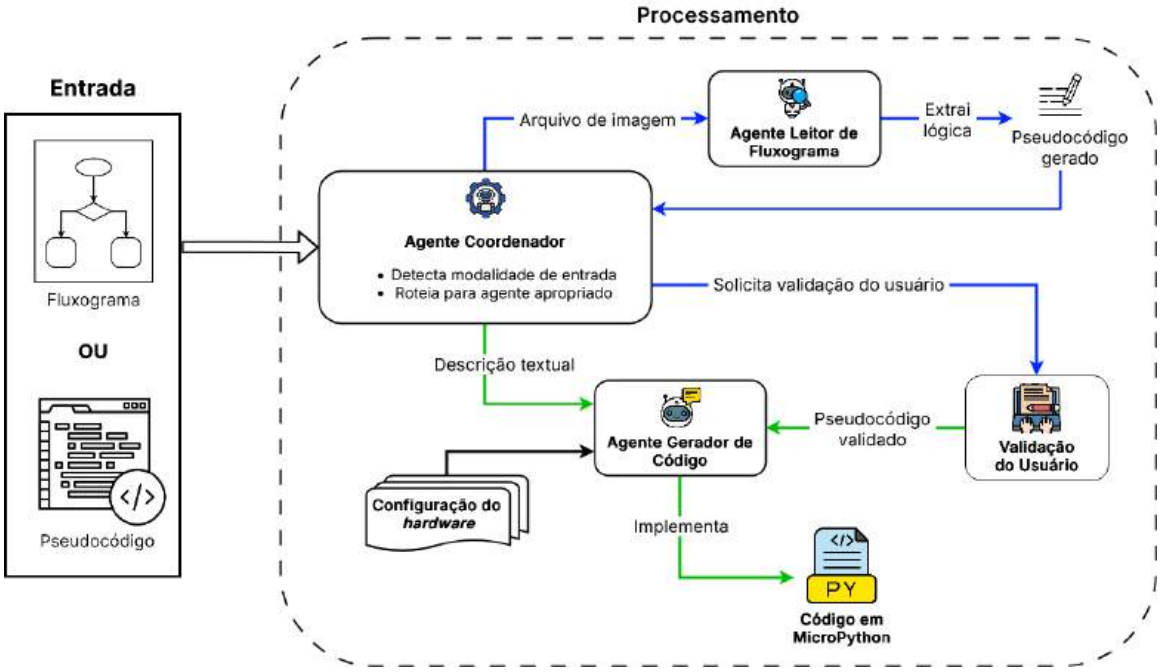


Figura 2: Arquitetura do sistema multiagentes desenvolvido.

Figura 3 exibe o fluxograma que representa a lógica de interação entre os agentes de acordo com o formato de entrada.

3 Metodologia

Esta seção detalha os procedimentos metodológicos e a infraestrutura experimental adotada para validar a eficácia da ferramenta de geração automática de código. A estratégia de avaliação foi estruturada para analisar, de forma isolada, as duas competências críticas do sistema: a interpretação visual de fluxogramas e a síntese de código funcional para sistemas embarcados. Inicialmente, a Seção 3.1 descreve o processo de construção do *dataset* de avaliação, desenvolvido especificamente para cobrir as especificidades da placa BitDogLab. Na Seção 3.2, são apresentados os critérios de seleção dos modelos de linguagem, fundamentados em *benchmarks* do estado da arte. As Seções 3.3 e 3.4 descrevem o ambiente de execução utilizado para execução dos modelos, detalhando a configuração do Ollama como servidor de inferência e do DSPy para gerenciamento programático dos *prompts* passados aos modelos, respectivamente. Por fim, detalham-se as métricas de pontuação definidas para a análise quantitativa dos resultados obtidos na Seção 3.5.

3.1 Conjunto de Dados

Para validar a eficácia dos modelos nas tarefas propostas, foi desenvolvido integralmente um conjunto de dados de avaliação composto por 20 instâncias de teste criadas para este

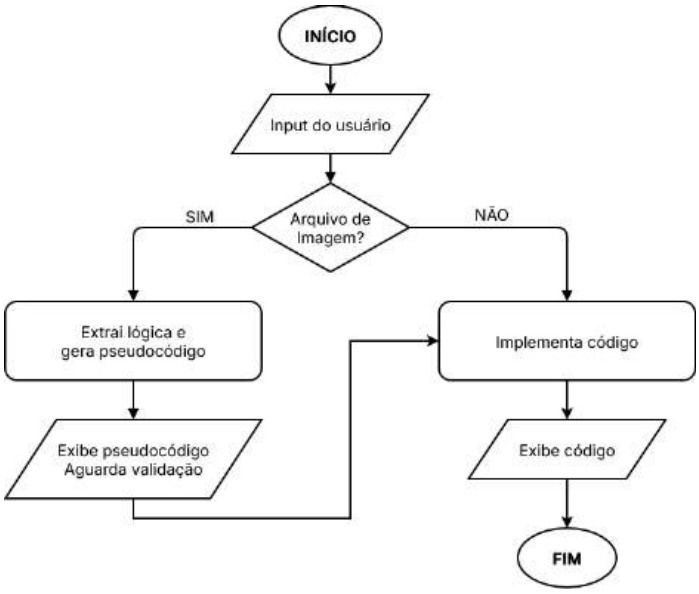


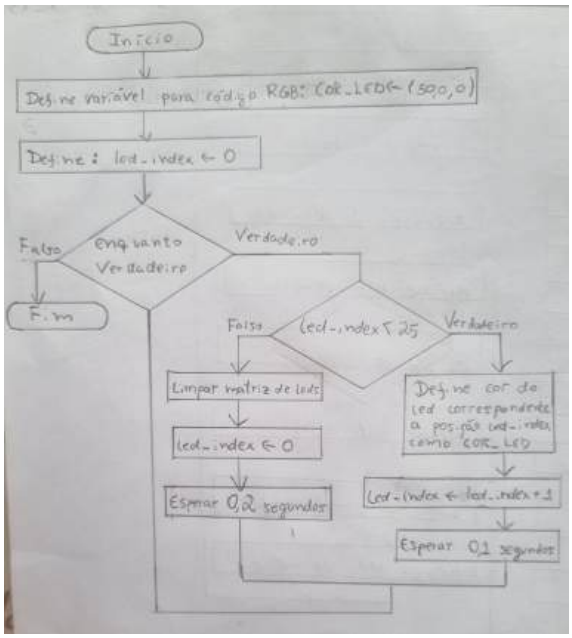
Figura 3: Fluxograma da interação entre agentes no sistema.

projeto. A decisão pela confecção manual dos dados visou garantir o controle absoluto sobre a complexidade lógica e a correção funcional dos exemplos. Este conjunto foi desenhado para atuar como um "padrão-ouro", garantindo que tanto a etapa de interpretação visual quanto a de geração de código fossem submetidas a cenários controlados e verificáveis.

O processo de elaboração dos artefatos seguiu uma abordagem reversa e sequencial, estruturada em três etapas para assegurar a consistência entre a imagem e o código. Inicialmente, os códigos-fonte em MicroPython foram implementados e testados diretamente na placa BitDogLab. Apenas após a confirmação de que o código executava a tarefa desejada sem erros (validação funcional), o exemplo era aprovado para compor o *dataset*. Baseando-se na lógica validada, os fluxogramas foram então desenhados manualmente utilizando lápis e papel. Essa escolha metodológica introduz um desafio adicional e realista aos modelos de visão: a necessidade de interpretar traços imperfeitos, variações de caligrafia e rascunhos, simulando o cenário de uso real previsto para os modelos. Por fim, foram redigidos os pseudocódigos correspondentes, servindo como o elo intermediário ideal entre a representação visual e o código estruturado.

Cada instância do *dataset* é, portanto, composta por uma tripla de artefatos alinhados, que representam o fluxo completo de transformação esperado pela ferramenta: fluxograma (entrada), pseudocódigo (referência intermediária) e o código MicroPython (referência final). A Figura 4 mostra um exemplo de entrada do *dataset*.

Um aspecto metodológico central na elaboração dos fluxogramas e pseudocódigos foi a omissão deliberada de detalhes de implementação de baixo nível, tais como diretivas de importação de bibliotecas e rotinas de configuração inicial dos pinos GPIO dos componentes. Essa decisão visa avaliar a capacidade inferencial do modelo gerador de código, forçando-o a deduzir a infraestrutura necessária a partir da documentação técnica fornecida via



(a) Lógica em fluxograma

```python
from machine import Pin
import neopixel
from utime import sleep

NUM_LEDS = 25
np = neopixel.NeoPixel(Pin(7), NUM_LEDS)

COR_LED = (50, 0, 0)
led_index = 0

while True:
    if led_index < NUM_LEDS:
        np[led_index] = COR_LED
        np.write()
        led_index += 1
        sleep(0.1)
    else:
        np.fill((0, 0, 0))
        np.write()
        led_index = 0
        sleep(0.2)
```

(b) Código em MicroPython

```
INÍCIO
DEFINE VARIÁVEL PARA CÓDIGO RGB: COR_LED <- (50, 0, 0)
DEFINE: LED_INDEX <- 0

ENQUANTO VERDADEIRO
    SE LED_INDEX < 25
        DEFINE COR DO LED CORRESPONDENTE A POSIÇÃO LED_INDEX COMO COR_LED
        LED_INDEX <- LED_INDEX + 1
        ESPERAR 0,1 SEGUNDOS
    SENÃO
        LIMPAR MATRIZ DE LEDS
        LED_INDEX <- 0
        ESPERAR 0,2 SEGUNDOS
FIM
```

(c) Descrição textual (pseudocódigo)

Figura 4: Exemplo de entrada do *dataset*.

contexto, em vez de apenas replicar instruções explícitas. Adicionalmente, adotou-se um nível de abstração elevado nas descrições lógicas: operações de manipulação de *hardware* complexas foram representadas em linguagem natural (por exemplo, "para cada *led* na sequência, acenda na cor vermelha e aguarde") em detrimento da sintaxe de programação, transferindo ao modelo a responsabilidade de mapear essas intenções semânticas para a estrutura de código correta.

A criação dos 20 casos de teste foi orientada pela necessidade de cobertura dos componentes principais presentes no *hardware* da BitDogLab. Os exemplos variam desde lógicas simples de acionamento único de LEDs até aplicações integradas que coordenam o uso de diversos componentes. A Tabela 1 apresenta a distribuição dos casos de teste em relação aos componentes acionados. Observa-se uma predominância de testes envolvendo Matrizes de LED e LEDs RGB, enquanto o *joystick* e o microfone foram os componentes menos representados.

Componente de Hardware	**N.º de Exemplos**	**Frequência Relativa**[1]
Matriz de LEDs (5x5)	8	40%
LED RGB	8	40%
Botões (A/B)	7	35%
Buzzer (Passivo)	7	35%
Display OLED (SSD1306)	4	20%
Joystick	2	10%
Microfone	2	10%

Tabela 1: Distribuição dos casos de teste por componente de *hardware* da BitDogLab.

3.2 Seleção de Modelos

Dada a vasta disponibilidade de LLMs no estado da arte, foi necessário estabelecer um critério de filtragem rigoroso para selecionar os candidatos mais aptos às tarefas de transcrição de fluxogramas em pseudocódigo e implementação em código. A seleção dos modelos avaliados neste trabalho baseou-se, primariamente, na análise de desempenho em *benchmarks* públicos que avaliam as competências necessárias para a realização de tais tarefas.

Concomitantemente, estabeleceu-se um limite superior de 20 bilhões de parâmetros aos candidatos. Esta decisão foi calculada para viabilizar a arquitetura multiagentes proposta: a GPU utilizada para hospedar o assistente virtual deve ser capaz de alocar, simultaneamente na VRAM, tanto o modelo de visão (leitor de fluxograma) quanto o modelo de código (gerador), permitindo a interação fluida entre os agentes sem a latência proibitiva de descarregamento e recarregamento de pesos (*model swapping*). Tal recorte alinha-se, ainda, à recente tendência da literatura onde modelos compactos otimizados têm demonstrado desempenho competitivo em domínios específicos, dispensando a onerosidade computacional de modelos massivos para a aplicação proposta [15].

[1]Nota: A soma das frequências supera 100% devido à natureza integradora dos testes, onde um único exemplo pode exigir a interação entre múltiplos componentes.

3.2.1 Critérios de Seleção dos Modelos de Interpretação Visual

A seleção dos modelos baseou-se na premissa de que a interpretação correta de um fluxograma exige a orquestração de múltiplas competências cognitivas. Para validar essas competências, foram selecionados *benchmarks* específicos que cobrem as seguintes dimensões:

Compreensão de Diagramas e Raciocínio Algorítmico Nesta categoria, buscou-se avaliar a capacidade dos modelos de compreender a estrutura topológica (nós e arestas) e a lógica sequencial implícita em representações visuais.

- AI2D [16]: Este é o *benchmark* de maior correlação direta com o objeto deste estudo. O AI2D avalia especificamente a compreensão de diagramas científicos e esquemáticos, exigindo que o modelo identifique constituintes visuais e suas relações semânticas. Sua inclusão justifica-se pela necessidade de o modelo distinguir corretamente entre blocos de processamento e setas de fluxo de dados.

- MathVista [17]: Embora focado em matemática, este *benchmark* é crucial por avaliar o raciocínio visual complexo e a resolução de problemas passo a passo (*chain-of-thought*). A habilidade de seguir uma sequência lógica em um problema matemático visual transfere-se diretamente para a capacidade de interpretar estruturas de controle (como laços e condicionais) em um algoritmo visual.

- ChartQA [18]: Focado na interpretação de dados em gráficos (barras, linhas), este *benchmark* avalia a precisão na extração de valores associados a elementos visuais. Sua relevância reside na verificação da capacidade do modelo de alinhar corretamente o texto (rótulos) com sua representação gráfica correspondente, essencial para não dissociar o conteúdo de uma caixa de fluxograma de sua posição no fluxo.

Reconhecimento Óptico de Caracteres (OCR) Dado que a lógica do código (nomes de variáveis, interação com componentes) reside no texto inscrito nas formas geométricas, a robustez do OCR é outro pré-requisito funcional. Um alto desempenho nestas métricas minimiza a perda de informações semânticas contidas nos nós do diagrama.

- OCRBench [19]: Por ser um agregador abrangente de diversas tarefas de reconhecimento de texto, o OCRBench serve como o indicador primário de legibilidade. Um baixo desempenho aqui inviabilizaria a transcrição correta da sintaxe contida nos blocos do fluxograma.

- TextVQA [20]: Este *dataset* de *Visual Question Answering* exige que o modelo não apenas leia o texto, mas o utilize para responder a perguntas sobre a imagem. Essa competência é fundamental para garantir que o modelo compreenda o texto em contexto (por exemplo, entender que o texto `X < 10` está dentro de um losango de decisão e não em um retângulo de processo).

Integridade Visual e Mitigação de Alucinações A geração de código funcional exige fidelidade absoluta à estrutura desenhada. A confiabilidade na identificação de conexões e fluxos de decisão é crítica, visto que a invenção de arestas ou nós inexistentes compromete a funcionalidade do código gerado.

- HallusionBench [21]: Diferentemente de métricas de alucinação focadas em objetos naturais, o HallusionBench avalia a consistência do raciocínio visual e a resistência a ilusões perceptivas. Sua inclusão é estratégica para filtrar modelos propensos a "inventar" conexões ou inverter a direção de setas em diagramas complexos, assegurando que a topologia do grafo gerado no código corresponda fielmente à imagem de entrada.

Modelos focados primariamente em imagens naturais ou descrição de cenas cotidianas foram preteridos em favor daqueles com especialização demonstrada nos domínios de documentos, diagramas e OCR denso, conforme os critérios supracitados.

3.2.2 Critérios de Seleção dos Modelos de Geração de Código

Analogamente à etapa visual, a seleção dos modelos para a fase de implementação não se baseou apenas na popularidade dos modelos, mas em métricas quantitativas de desempenho em tarefas de engenharia de software. Além disso, a geração de código para a BitDogLab impõe desafios que extrapolam a simples sintaxe correta: o modelo deve ser capaz de operar com o subconjunto de bibliotecas do MicroPython, respeitar restrições de idioma (variáveis em português) e aderir a diretrizes de formatação (comentários explicativos). Para validar essas capacidades, os modelos foram filtrados com base nos seguintes critérios de avaliação:

- HumanEval [22]: Considerado o padrão-ouro na avaliação de LLMs para codificação, este *benchmark* consiste em problemas de programação que exigem a implementação de corpos de funções a partir de assinaturas e *docstrings*. Dado que o MicroPython é uma implementação otimizada do Python 3, o desempenho no HumanEval serve como o indicador primário de que o modelo domina as estruturas de controle de fluxo, manipulação de dados e sintaxe da linguagem.

- MBPP (Mostly Basic Python Problems) [23]: Enquanto o HumanEval pode conter desafios algorítmicos complexos, o MBPP foca em conceitos fundamentais de programação e tarefas procedurais. Sua inclusão é estratégica para este trabalho, pois os *scripts* de controle para sistemas embarcados iniciantes (como acender LEDs ou ler botões) assemelham-se mais à lógica procedural direta avaliada no MBPP do que a algoritmos de competição avançados.

- IFEval (Instruction Following Evaluation) [24]: Este *benchmark* não avalia apenas a corretude do código, mas a fidelidade do modelo em seguir restrições verificáveis impostas pelo usuário (por exemplo, "não use bibliotecas externas" ou "gere comentários explicativos"). No contexto deste trabalho, um alto desempenho no IFEval é crucial para garantir que o modelo respeite as diretrizes de localização (uso de português em variáveis e comentários) e as especificidades da API do MicroPython, evitando a geração de código genérico que falharia na execução ou na compreensão pedagógica pelo usuário final.

3.2.3 Modelos Selecionados

A Tabela 2 consolida o conjunto final de arquiteturas selecionadas após a aplicação dos filtros de desempenho em *benchmarks* e eficiência computacional supracitados. Os modelos estão organizados conforme a tarefa atribuída no fluxo de processamento da ferramenta (interpretação visual ou geração de código), sendo caracterizados pela organização desenvolvedora, escala de parâmetros e respectiva referência técnica, visando assegurar a reprodutibilidade dos experimentos.

Modelo	**Desenvolvedor**	**Parâmetros**	**Tarefa**	**Fonte**
Qwen3-VL-Instruct	Alibaba Cloud	8 bilhões	Leitura de Fluxograma	[25]
InternVL3.5	OpenGVLab	8 bilhões	Leitura de Fluxograma	[26]
MiniCPM-V 4.5	OpenBMB	8 bilhões	Leitura de Fluxograma	[27]
Gemma 3-12b-it	Google DeepMind	12 bilhões	Leitura de Fluxograma	[28]
Kimi-VL-A3B	Moonshot AI	12 bilhões	Leitura de Fluxograma	[29]
Qwen2.5-Coder	Alibaba Cloud	14 bilhões	Codificação	[30]
MiniCPM4.1	OpenBMB	8 bilhões	Codificação	[31]
Phi-4	Microsoft	14 bilhões	Codificação	[32]
NextCoder-14B	Microsoft	14 bilhões	Codificação	[33]
GPT-OSS-20B	OpenAI	20 bilhões	Codificação	[34]

Tabela 2: Especificações técnicas e fontes dos modelos selecionados para avaliação.

3.3 Ambiente de Inferência Local

Para a execução local dos Grandes Modelos de Linguagem (LLMs) e Multimodais (LMMs), adotou-se a plataforma Ollama [35]. Esta ferramenta atua como um ambiente de execução de alto desempenho, projetado para simplificar a implantação de modelos de código aberto em infraestruturas locais, abstraindo as complexidades de configuração de bibliotecas de baixo nível (como PyTorch ou TensorFlow) e *drivers* de GPU.

A escolha do Ollama fundamenta-se, primariamente, na sua arquitetura otimizada baseada no *backend llama.cpp*. Essa base tecnológica permite a execução eficiente de modelos quantizados, reduzindo drasticamente os requisitos de memória VRAM, quando comparado a alternativas como HuggingFace Transformers [36] e vLLM [37], sem perdas significativas de precisão. Além disso, a plataforma oferece uma interface unificada via API REST, o que elimina a necessidade de escrever *scripts* de carregamento específicos para cada arquitetura de modelo, garantindo interoperabilidade e facilitando a troca rápida de modelos.

3.4 *Framework* DSPy

A construção de aplicações robustas baseadas em Modelos de Linguagem Grande (LLMs) enfrenta desafios significativos relacionados à estabilidade e reprodutibilidade dos *prompts*. A abordagem tradicional, frequentemente denominada "engenharia de *prompts*", depende

de ajustes manuais e empíricos em *strings* de texto para guiar o comportamento do modelo. Essa metodologia apresenta limitações críticas: é frágil a pequenas variações léxicas, dificilmente escalável e acopla rigidamente a lógica do programa à representação textual específica que um determinado modelo compreende melhor.

Para mitigar esses problemas, este projeto adotou o framework DSPy (*Declarative Self-improving Language Programs in Python*) [38]. Diferentemente de bibliotecas que apenas gerenciam *templates* de texto, o DSPy propõe uma mudança de paradigma ao tratar *prompts* não como cadeias estáticas, mas como parâmetros otimizáveis de um programa. O *framework* abstrai a interação com o LLM, separando o fluxo lógico da aplicação (o que o sistema deve fazer) da representação textual enviada ao modelo (como a instrução é formulada). A implementação do DSPy baseia-se em dois conceitos fundamentais que foram empregados na arquitetura deste trabalho: Assinaturas (*Signatures*) [39] e Módulos (*Modules*) [40].

As Assinaturas definem a especificação declarativa do comportamento de entrada e saída de uma transformação, abstraindo as instruções textuais de baixo nível. Analogamente às assinaturas de função em linguagens tipadas, uma *Signature* no DSPy declara o que o modelo deve realizar, especificando os campos de entrada e os campos de saída esperados, sem ditar o texto exato que deve ser passado ao modelo para atingir esse objetivo. No contexto deste trabalho, as assinaturas foram definidas para estruturar a conversão do fluxograma e a geração do código final, conforme mostrado na Listing 1.

Listing 1: Código Python definindo as *Signatures*

```python
class FlowchartToPseudocode(dspy.Signature):
    """
    Read a flowchart image and extract the logic in a pseudocode in
        Portuguese (Pt-BR).
    Keep the exact logic structure (conditionals, loops, start/end).
    """

    image: str = dspy.InputField(desc="Image of the flowchart encoded as
        base64")
    pseudocode: str = dspy.OutputField(
        desc="Pseudocode in Portuguese describing the logic extracted from
            the flowchart"
    )


class PseudocodeToMicroPython(dspy.Signature):
    """
    Convert a pseudocode in Portuguese to a valid MicroPython code.

    CRITICAL INSTRUCTIONS:
    1. Read the 'hardware_context' carefully.
    2. You MUST use the exact GPIO pins, libraries, and instantiation
        methods described in the 'hardware_context'.
    3. Do not invent pins or libraries not supported by the hardware
        documentation provided.
    """

```

```
    hardware_context: str = dspy.InputField(
        desc="Technical documentation containing GPIO mappings, required
            libraries, and how to instantiate components."
    )
    pseudocode: str = dspy.InputField(
        desc="The pseudocode in Portuguese describing the logic."
    )
    micropython_code: dspy.Code["python"] = dspy.OutputField(
        desc="The generated MicroPython code implementing the logic."
    )
```

A classe `FlowchartToPseudocode` encapsula a tarefa de percepção multimodal, recebendo a representação codificada da imagem e instruindo o modelo a preservar a fidelidade estrutural (condicionais e laços) na transcrição para pseudocódigo. Subsequentemente, a assinatura `PseudocodeToMicroPython` modela a etapa de síntese de código, incorporando explicitamente um mecanismo de ancoragem através do campo de entrada `hardware_context`, acompanhado de instruções críticas que restringem o espaço de geração às bibliotecas e pinagens válidas documentadas, visando mitigar alucinações de *hardware*.

Enquanto as Assinaturas definem a interface, os Módulos são as abstrações arquiteturais que implementam a estratégia de execução dessas assinaturas. Um módulo no DSPy pode encapsular técnicas complexas de *prompting*, como *Chain-of-Thought* [41] ou *ReAct* [42], de maneira transparente ao desenvolvedor. Ao instanciar um módulo parametrizado com uma assinatura específica, o *framework* gerencia automaticamente a construção do *prompt*, a inclusão de exemplos (se houver) e a formatação da saída.

Para a execução de ambas as tarefas, os módulos foram instanciados utilizando a estratégia `dspy.ChainOfThought`, induzindo o modelo a gerar passos intermediários de raciocínio antes da produção da resposta final, o que favorece a robustez na interpretação de diagramas complexos e na implementação de lógicas dependentes de contexto. A Listing 2 mostra a mensagem de sistema gerada automaticamente a partir da Assinatura definida para o modelo de geração de código, enquanto a Listing 3 mostra como a entrada fornecida pelo usuário é formatada e passada para o modelo no formato especificado.

Listing 2: *System message* gerada pelo DSPy para o módulo de geração de código

```
Your input fields are:
1. ‘hardware_context‘ (str): Technical documentation containing GPIO
    mappings, required libraries, and how to instantiate components.
2. ‘pseudocode‘ (str): The pseudocode in Portuguese describing the logic.
Your output fields are:
1. ‘reasoning‘ (str):
2. ‘micropython_code‘ (Code_python): The generated MicroPython code
    implementing the logic.
    Type description of Code_python: Code represented in a string,
        specified in the ‘code‘ field. If this is an output field, the code
        field should follow the markdown code block format, e.g.
‘‘‘python
{code}
‘‘‘ or
‘‘‘cpp
{code}
```

```
13 ```
14 Programming language: python
15 All interactions will be structured in the following way, with the
       appropriate values filled in.
16
17 [[ ## hardware_context ## ]]
18 {hardware_context}
19
20 [[ ## pseudocode ## ]]
21 {pseudocode}
22
23 [[ ## reasoning ## ]]
24 {reasoning}
25
26 [[ ## micropython_code ## ]]
27 {micropython_code}        # note: the value you produce must adhere to the
       JSON schema: {"type": "object", "properties": {"code": {"type":
       "string", "title": "Code"}}, "required": ["code"], "title":
       "Code_python"}
28
29 [[ ## completed ## ]]
30 In adhering to this structure, your objective is:
31         Convert a pseudocode in Portuguese to a valid MicroPython code.
32
33         CRITICAL INSTRUCTIONS:
34         1. Read the 'hardware_context' carefully.
35         2. You MUST use the exact GPIO pins, libraries, and instantiation
               methods described in the 'hardware_context'.
36         3. Do not invent pins or libraries not supported by the hardware
               documentation provided.
```

Listing 3: Formatação gerada pelo DSPy para a mensagem de usuário no módulo de geração de código

```
1 [[ ## hardware_context ## ]]
2 ...
3
4 [[ ## pseudocode ## ]]
5 ...
6
7 Respond with the corresponding output fields, starting with the field `[[
      ## reasoning ## ]]`, then `[[ ## micropython_code ## ]]` (must be
      formatted as a valid Python Code_python), and then ending with the
      marker for `[[ ## completed ## ]]`.
```

Durante a fase de integração experimental, identificaram-se obstáculos técnicos de interoperabilidade entre o *framework* DSPy e o servidor de inferência Ollama, especificamente no processamento de entradas multimodais. Observou-se uma incompatibilidade crítica nos protocolos de serialização de imagens: o formato de codificação utilizado pelo DSPy diverge do formato suportado pela API do Ollama, ocasionando falhas de execução. Embora correções preliminares tenham sido aplicadas para contornar esse obstáculo, persistiram inconsistências no *pipeline* de resposta, uma vez que a saída bruta gerada pelos modelos

no Ollama não aderia estritamente ao formato estruturado exigido pelos *parsers* de saída do DSPy. Diante dessas restrições e visando assegurar a estabilidade do sistema, optou-se por uma arquitetura de orquestração híbrida: o módulo de interpretação de fluxogramas foi desacoplado do DSPy e implementado via chamadas diretas à API do Ollama, reservando o uso do *framework* DSPy exclusivamente para a etapa de geração de código.

3.5 Métricas de Avaliação

Em sua revisão crítica sobre o tema, Paul, Zhu e Bayley [43] descrevem que as formas mais comuns para se avaliar a qualidade de um LLM no contexto de geração de código são a corretude funcional do código gerado (mensurada via execução de testes, verificando em quantos casos de teste o programa gerado produz a saída correta) e a proximidade sintática, utilizando métricas de similaridade derivadas da linguagem natural (por exemplo, BLEU [44], ROUGE [45], METEOR [46] e ChrF [47]) ou propostas especificamente para código (como RUBY [48] e CodeBLEU [49]), sendo que a maioria das avaliações de performance em geração de código emprega correção em testes.

Contudo, em cenários de geração de código voltados para sistemas embarcados, como é o caso da BitDogLab, a inexistência de simuladores ou ambiente de teste de *software* robustos impõe desafios significativos à validação da corretude funcional. A impossibilidade de emular interações com componentes físicos, como o acionamento de botões ou a resposta sonora de um *buzzer*, torna inviável a execução de testes automatizados. Diante dessa limitação, a alternativa convencional seria recorrer à análise de similaridade sintática em comparação a códigos de referência produzidos manualmente.

Entretanto, a literatura recente aponta severas restrições quanto à eficácia dessa abordagem. Um estudo conduzido por Evtikhiev et al. [50] demonstrou que mesmo as métricas desenvolvidas especificamente para esse propósito, como RUBY e CodeBLEU, não apresentam desempenho superior a métricas genéricas de tradução automática; dentre as avaliadas, a métrica ChrF foi a que mais se aproximou da avaliação humana, embora ainda distante de ser considerada ideal. Corroborando essa análise, Naik [51] investigou a capacidade de métricas baseadas em *embeddings*, como a CodeBERTScore [52], para mensurar corretude funcional, observando uma baixa correlação nos resultados. Consequentemente, Paul, Zhu e Bayley [43] concluem que, a despeito dos esforços de pesquisa, a avaliação automática de código via análise estática permanece um problema em aberto.

Considerando a inviabilidade técnica de testes automáticos, seja funcionais ou de similaridade sintática, este trabalho não empregou métricas automatizadas para a seleção dos modelos. Em vez disso, optou-se por realizar uma avaliação manual tanto dos pseudocódigos gerados pelo agente leitor de fluxograma quanto dos códigos produzidos pelo agente codificador. Para sistematizar a análise qualitativa e garantir a consistência da avaliação manual, foram estabelecidos critérios objetivos de pontuação para as duas etapas principais do fluxo de trabalho.

Na primeira etapa, referente à conversão de fluxogramas em pseudocódigo, foi elaborada uma escala de avaliação de quatro níveis (0 a 3), desenvolvida para capturar as nuances da interpretação visual de algoritmos e permitindo distinguir desde alucinações severas até inconsistências leves na lógica recuperada. Os critérios adotados são definidos a seguir:

- **Pontuação 0 (Dissociação Semântica):** O modelo apresenta alucinação severa ou fuga total do tema, gerando um texto que não possui nenhuma correlação com os elementos visuais ou lógicos presentes no fluxograma de entrada;
- **Pontuação 1 (Tangência):** O modelo identifica elementos isolados (como textos internos ou nós específicos), mas falha na reconstrução da topologia do grafo, resultando em uma estrutura lógica incoerente ou fragmentada;
- **Pontuação 2 (Consistência Parcial):** A estrutura global e o fluxo lógico principal foram compreendidos corretamente) e não houve alucinações, porém o pseudocódigo apresenta erros na lógica ou omissões;
- **Pontuação 3 (Correspondência Plena):** O pseudocódigo gerado reproduz fielmente a lógica, a estrutura e o conteúdo textual do fluxograma original, sem alucinações ou omissões.

Para a segunda etapa, que consiste na geração do código executável para a BitDogLab, foi desenvolvida uma rubrica específica de quatro níveis (0 a 3). Essa escala foi desenhada para distinguir erros de sintaxe de erros semânticos e, crucialmente, para isolar alucinações relacionadas às especificidades do *hardware* (como pinagem e bibliotecas):

- **Pontuação 0 (Erro de Sintaxe):** O código gerado é sintaticamente inválido, resultando em erro de execução;
- **Pontuação 1 (Erro de Configuração de Hardware):** O código é sintaticamente válido, mas apresenta erros na instanciação dos componentes físicos (como definição incorreta de pinos GPIO ou importação de bibliotecas incorretas);
- **Pontuação 2 (Erro de Lógica):** O código é sintaticamente válido e inicializa o *hardware* corretamente, mas a execução não produz o comportamento funcional esperado (erro semântico);
- **Pontuação 3 (Funcional):** O código está correto e executa a tarefa desejada na placa BitDogLab sem necessidade de correções.

4 Resultados e Discussão

Esta seção apresenta os resultados obtidos a partir da avaliação dos modelos selecionados no *dataset* de teste. Para cada um dos 20 casos de uso, foram coletadas métricas de pontuação (conforme as escalas de 0 a 3 definidas), tempo de execução da inferência e consumo máximo de memória de vídeo (VRAM). Todos os experimentos de *benchmarking*, bem como a execução da ferramenta final, foram conduzidos em uma estação de trabalho que utiliza como sistema operacional a distribuição Linux Ubuntu 24.04 equipada com uma unidade de processamento gráfico (GPU) modelo NVIDIA GeForce RTX 4090 dedicada, dispondo de 24 GiB de memória de vídeo (VRAM) e versão do CUDA 12.2.

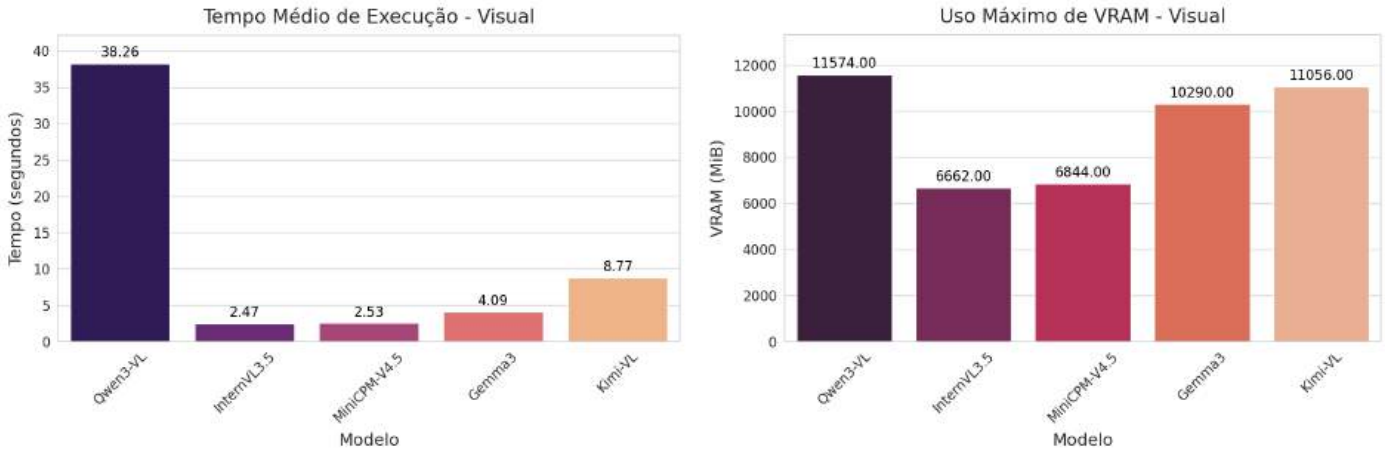


Figura 5: Comparativos entre os tempos médios de execução (à esquerda) e o uso máximo de VRAM (à direita) para os modelos visuais

A Tabela 3 sumariza o desempenho geral dos modelos, detalhando a distribuição percentual das notas atribuídas e a pontuação média final. Os modelos estão agrupados conforme a tarefa desempenhada: interpretação visual de fluxogramas e geração de código em MicroPython. Além disso, as Figuras 5 e 6 comparam os tempos médios de execução e o uso de VRAM de cada modelo.

Modelo	**Distribuição de Pontuação (%)**				**Média**
	0	**1**	**2**	**3**	
Tarefa: Leitura de Fluxograma					
Qwen3-VL-Instruct	0%	0%	50%	50%	**2,50**
Gemma 3-12b-it	0%	40%	45%	15%	1,75
MiniCPM-V 4.5	20%	25%	40%	15%	1,50
InternVL3.5	65%	10%	10%	15%	0,75
Kimi-VL-A3B	0%	0%	0%	0%	0,00*
Tarefa: Codificação					
GPT-OSS-20B	5%	5%	0%	90%	**2,75**
NextCoder-14B	10%	10%	10%	70%	2,40
Phi-4	15%	15%	20%	50%	2,05
Qwen2.5-Coder-14B	35%	5%	10%	50%	1,75
MiniCPM4.1	50%	35%	5%	10%	0,75

*O modelo Kimi-VL-A3B não produziu saídas válidas em nenhuma iteração.

Tabela 3: Distribuição de pontuações e média final dos modelos avaliados.

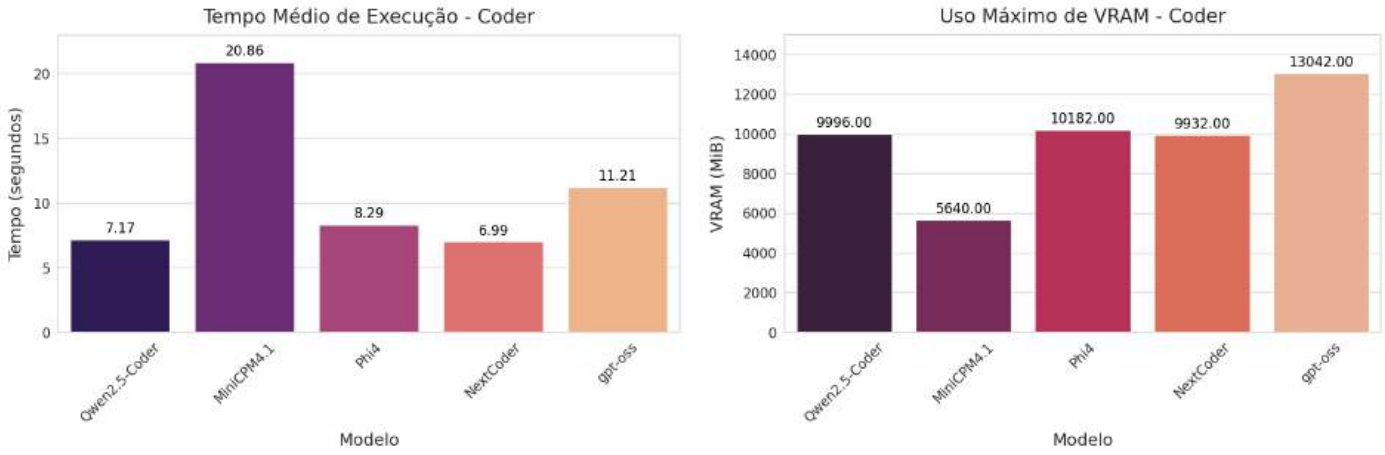


Figura 6: Comparativos entre os tempos médios de execução (à esquerda) e o uso máximo de VRAM (à direita) para os modelos de geração de código

4.1 Desempenho na Interpretação de Fluxogramas

No que tange à tarefa de extração de lógica visual, os resultados consolidados na Tabela 3 indicam uma disparidade significativa entre os candidatos. O modelo Qwen3-VL-Instruct obteve o melhor desempenho do grupo, com média de 2,50, sendo o único a não registrar pontuações 0 (fuga total) ou 1 (tangência), concentrando 100% de suas respostas nos estratos de alta fidelidade (notas 2 e 3). A análise manual revelou que a maioria de suas penalidades (nota 2) deveu-se a erros pontuais de OCR (leitura incorreta de valores numéricos), enquanto a estrutura lógica (condicionais e laços) manteve-se correta. Houve apenas um caso de erro lógico grave, o que reforça sua robustez para a tarefa de raciocínio espacial.

O Gemma 3-12b-it apresentou o segundo melhor desempenho (média 1,75), caracterizado por uma consistência na pontuação mediana (9 casos com nota 2), embora com menor capacidade de atingir a pontuação máxima (apenas 3 exemplos). Este modelo mostrou-se competente na captura da topologia do fluxograma, acertando a lógica do fluxo principal na maioria dos casos. Entretanto, apresentou dificuldades em dois pontos, que ocasionaram uma diminuição significativa de sua nota: erros frequentes de OCR, especialmente em listas de valores, e a omissão de procedimentos auxiliares definidos fora do fluxo principal, sendo este o mais crítico e, portanto, o principal causador da significativa redução de pontos.

Já os modelos MiniCPM-V 4.5 e InternVL3.5 demonstraram instabilidade, com médias de 1,50 e 0,75 respectivamente. O InternVL3.5, em particular, apresentou falha severa na maioria dos casos, recebendo pontuação 0 em 13 dos 20 casos testados devido a um comportamento característico de "colapso de alucinação": além de inventar lógicas inexistentes sem relação nenhuma com o conteúdo do fluxograma, o modelo frequentemente gerava respostas contendo caracteres em chinês, indicando uma falha grave na generalização para o domínio específico do *dataset* e possivelmente uma contaminação pelos dados de pré-treino.

O modelo MiniCPM-V 4.5, por outro lado, foi o que demonstrou a maior robustez em OCR, transcrevendo corretamente listas numéricas longas e densas. Contudo, falhou

criticamente no quesito *"instruction following"*: em diversos casos, gerou código Python diretamente em vez do pseudocódigo solicitado. Essa violação do formato de saída foi penalizada com nota 0, mascarando sua excelente capacidade de percepção visual e causando a baixa pontuação geral reportada.

Por fim, o modelo Kimi-VL-A3B, apesar de sua arquitetura promissora, apresentou uma falha sistêmica na ingestão dos dados visuais. Em todas as iterações, o modelo reportou incapacidade de ler a imagem ou alegou que nenhum arquivo havia sido fornecido, resultando em nulidade funcional.

4.2 Desempenho na Geração de Código MicroPython

Para a etapa de implementação, a Tabela 3 evidencia a liderança do modelo GPT-OSS-20B, que alcançou a maior média global do experimento (2,75). Este modelo atingiu a pontuação máxima (código correto e funcional) em 90% dos casos de teste. O NextCoder-14B também apresentou performance destacada, com média de 2,40 e 70% de acertos totais.

Os modelos Phi-4 e Qwen2.5-Coder-14B apresentaram desempenho intermediário, com médias de 2,05 e 1,75. Observa-se uma polarização no desempenho do Qwen2.5-Coder: embora tenha atingido a nota máxima em 50% dos casos, falhou completamente (nota 0) em 35% das tentativas. O MiniCPM4.1 obteve o menor desempenho do grupo (média 0,75), com metade das gerações resultando em código sintaticamente inválido e mais um terço, aproximadamente, apresentando erro na configuração do *hardware*, ou seja, 17 dos 20 casos testados resultaram em erro de execução.

A análise manual dos códigos gerados por cada modelo possibilitou identificar as dificuldades mais frequentes dos LLMs. A seguir, são apresentados os erros mais comuns observados em cada nível da hierarquia de pontuação:

- **Erros de Sintaxe e Dependências (Pontuação 0):** A falha mais recorrente nessa categoria foi a omissão de importações necessárias. Mesmo com o contexto fornecido, modelos menores como o MiniCPM4.1 frequentemente tentavam utilizar classes sem importá-las, gerando erros de execução imediatos.

- **Alucinação de *Hardware* e Configuração (Pontuação 1):** Erros classificados como falha de configuração revelaram a dificuldade dos modelos em respeitar restrições físicas. Um exemplo crítico foi a definição da frequência do *buzzer* como 0 Hz, o que levanta uma exceção `ValueError: freq too small` na biblioteca do MicroPython. Outro erro comum, embora silencioso (sem travar o código), foi a inversão dos pinos referentes à leitura das posições nos eixos X e Y do *joystick* ao instanciar o componente. Tais erros indicam que, embora o modelo gere código sintaticamente válido, ele falha em ancorar a lógica na documentação de *hardware* fornecida.

- **Erros Semânticos de Estado (Pontuação 2):** Nos casos onde o código executava e a configuração estava correta, o erro predominante foi a ausência do comando de atualização de estado, especificamente `np.write()` para a matriz de LEDs. Esse é um erro clássico em bibliotecas do tipo NeoPixel, onde o modelo altera o *buffer* de

memória mas "esquece" de enviar o sinal para o *hardware*, resultando em um programa que roda mas não produz resposta visual.

Em suma, o GPT-OSS-20B e o NextCoder-14B destacaram-se por evitar esses erros comuns, demonstrando uma capacidade superior de gerenciar tanto as dependências de biblioteca quanto as especificidades da BitDogLab.

4.3 Eficiência Computacional

A análise dos custos computacionais, apresentada nas Figuras 5 e 6, revela o *trade-off* entre tamanho do modelo, eficiência computacional e desempenho na tarefa.

Entre os modelos visuais, o Qwen3-VL, que obteve o melhor desempenho qualitativo, também registrou o maior tempo médio de execução (38,26 segundos) e o maior pico de tempo absoluto (cerca de 83 segundos). Além disso, consumiu aproximadamente 11,5 GiB de VRAM. Embora tenha se destacado no desempenho, sendo o melhor por uma grande margem, também foi o modelo mais lento, o que pode ser um ponto negativo em cenários onde a latência é crítica. No entanto, em casos onde o tempo de espera não seja um problema, o desempenho superior pode justificar a escolha.

Na tarefa de codificação, o GPT-OSS-20B, devido à sua grande escala paramétrica (20 bilhões de parâmetros), demandou o maior volume de memória (cerca de 13 GiB) e apresentou um tempo médio de inferência de 11,21 segundos. Esse número elevado de parâmetros, quando comparado aos demais modelos, pode ser uma possível justificativa para seu desempenho qualitativo superior. Por outro lado, o MiniCPM4.1, embora seja um modelo menor (o único desta categoria com apenas 8 bilhões de parâmetros), consistentemente apresentou uma anomalia no tempo de execução, com um pico de 309 segundos em um dos casos de teste. Somado ao fato dele ter sido o modelo com pior pontuação geral, isso sugere dificuldades de convergência ou laços de geração excessivos.

Os modelos Gemma 3-12B (na tarefa visual) e NextCoder-14B (na tarefa de codificação) apresentaram uma relação equilibrada de custo-benefício, operando na faixa de 10 GiB de VRAM e com tempos de resposta abaixo de 7 segundos. Ambos se destacaram como os segundos melhores em termos de pontuação média em suas respectivas categorias, oferecendo um bom equilíbrio entre desempenho e eficiência.

5 Conclusão e Trabalhos Futuros

O presente trabalho dedicou-se à validação experimental de uma arquitetura de inteligência artificial generativa aplicada ao ensino de lógica de programação através de um sistema embarcado. Diante do desafio de converter diagramas visuais em código funcional para a placa de *hardware* BitDogLab, o estudo buscou identificar, através de *benchmarking*, quais Modelos de Linguagem (LLMs) e Multimodais (LMMs) oferecem o equilíbrio ideal entre precisão cognitiva e viabilidade computacional em *hardware* local.

Os resultados obtidos demonstram a viabilidade técnica de utilizar modelos de médio porte (até 20 bilhões de parâmetros) para tarefas complexas de engenharia, refutando a premissa de que apenas modelos proprietários massivos seriam aptos a tais funções. Na etapa

de interpretação visual, o modelo Qwen3-VL-Instruct (versão com 8 bilhões de parâmetros) consolidou-se como a referência de desempenho, apresentando a maior robustez na compreensão de estruturas lógicas e topologias de fluxo, superando concorrentes em consistência estrutural. A análise qualitativa revelou que, para a tarefa de transcrição de algoritmos, a capacidade de raciocínio espacial e seguimento de instruções prevalece sobre a capacidade de OCR bruto.

No tocante à geração de código, o GPT-OSS-20B apresentou a maior taxa de acerto funcional. Contudo, considerando a restrição arquitetural imposta pelo ambiente de execução (uma GPU com 24 GiB de VRAM destinada a alocar ambos os agentes simultaneamente), a escolha ótima recai sobre o NextCoder-14B. Este modelo apresentou desempenho estatisticamente próximo ao líder, mas com uma eficiência de memória que permite sua coexistência com o agente visual no mesmo dispositivo, garantindo a latência operacional necessária para um assistente interativo. Portanto, a configuração final recomendada para a ferramenta assistiva da BitDogLab é a orquestração híbrida entre Qwen3-VL (Visão) e NextCoder-14B (Código).

Para a evolução do projeto, vislumbram-se oportunidades cruciais de aprimoramento tanto na base de dados quanto na engenharia de software. Primeiramente, recomenda-se a expansão do *dataset* de avaliação, atualmente restrito a uma única caligrafia e condições controladas; a inclusão de uma maior variedade de estilos de escrita e imagens com ruídos reais (baixa iluminação, desfoque) é fundamental para garantir a generalização da ferramenta em sala de aula. No âmbito da implementação, trabalhos futuros devem priorizar a correção da interoperabilidade entre o *framework* DSPy e os modelos visuais. A plena integração permitirá a utilização de otimizadores automáticos (compilação de *prompts*) e estratégias de aprendizado *few-shot*, além de viabilizar a construção de (*Chain-of-Thought*) de múltiplas etapas, refinando o raciocínio intermediário do agente.

Por fim, apesar do êxito na abordagem *zero-shot*, a persistência de erros de "alucinação de *hardware*" sugere que o fornecimento de contexto via *prompt* possui um limite de eficácia. Sugere-se, portanto, a combinação das melhorias de *pipeline* citadas acima com técnicas de ajuste fino supervisionado, especializando os pesos das redes na sintaxe específica da BitDogLab para atingir níveis de confiabilidade de produção.

6 Disponibilidade de Código e Reprodutibilidade

Visando fomentar a transparência científica e permitir a replicação dos experimentos detalhados neste trabalho, todos os artefatos de *software* desenvolvidos e instruções para seu uso foram disponibilizados em repositórios públicos.

O código-fonte completo da ferramenta assistiva, incluindo a implementação da arquitetura multiagentes e a interface de interação, encontra-se hospedado em `https://github.com/AI-Unicamp/BitDogLab-Chatbot`.

Paralelamente, os *scripts* de automação utilizados para o *benchmarking*, bem como o *dataset* proprietário e os resultados das avaliações dos modelos, estão disponíveis em: `https://github.com/Lozavival/BitDogLab-Benchmarking`.

Agradecimentos

A realização deste trabalho não seria possível sem a presença e o apoio daqueles que caminharam ao meu lado durante esta jornada.

Aos meus pais, Wiliam e Marcia Lozano, a minha eterna gratidão. Vocês são os pilares da minha vida e os principais responsáveis por eu ter chegado até aqui. Obrigado pelo amor incondicional, pelos sacrifícios feitos em prol da minha educação e por sempre acreditarem no meu potencial, mesmo quando eu hesitava. Esta conquista também é de vocês.

À minha família, pelo incentivo constante e pela compreensão nos momentos de ausência necessários para a dedicação aos estudos.

Aos meus orientadores, Prof. André Santanchè e Profª. Paula Dornhofer Paro Costa, agradeço imensamente pela orientação segura, pela paciência e pela generosidade em compartilhar seus conhecimentos. Suas correções e sugestões foram essenciais para o meu amadurecimento acadêmico e profissional.

Aos meus amigos e colegas de curso, agradeço pelo companheirismo, pelas trocas de experiência e pelos momentos de descontração que tornaram a caminhada mais leve.

Por fim, à minha namorada, Vitoria, meu carinho especial. Obrigado por estar ao meu lado em todos os momentos, pela paciência incansável durante as longas horas de estudo e por ser meu refúgio e incentivo diário. Sua presença tornou esta etapa muito mais feliz.

Referências

[1] Hugo Horta. "Education in Brazil: Access, quality and STEM". Em: *B. Freeman (Ed.), Consultant Report Securing Australia's Future STEM: Country comparison* (2013), pp. 28–29.

[2] Fabiano Fruett et al. "Empowering STEAM Activities With Artificial Intelligence and Open Hardware: The BitDogLab". Em: *IEEE Transactions on Education* (2024).

[3] Aryaman Darda e Reetu Jain. "Code Generation from Flowchart using Optical Character Recognition & Large Language Model". Em: *Authorea Preprints* (2024).

[4] Pratul Trivedi et al. "System model for syntax free coding". Em: *2019 Global Conference for Advancement in Technology (GCAT)*. IEEE. 2019, pp. 1–5.

[5] Ashish Vaswani et al. "Attention is all you need". Em: *Advances in neural information processing systems* 30 (2017).

[6] COLE STRYKER. *What are large language models (LLMs)?* Disponível em: `https://www.ibm.com/think/topics/large-language-models`. Acesso em: 13 out. 2025.

[7] GitHub. *GitHub Copilot*. Disponível em: `https://github.com/features/copilot`. Acesso em: 13 out. 2025.

[8] Anysphere. *Cursor - The AI Code Editor*. Disponível em: `https://cursor.com/`. Acesso em: 13 out. 2025.

[9] Christian Bird et al. "Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools". Em: *Queue* 20.6 (2022), pp. 35–57.

[10] BitDogLab. *BitDogLab*. Acesso em: 12 dez. 2025. URL: https://bitdoglab.webcontent.website/.

[11] MicroPython. *MicroPython*. Acesso em: 12 dez. 2025. URL: https://micropython.org/.

[12] Amal El Fallah Seghrouchni, Adina Magda Florea e Andrei Olaru. "Multi-agent systems: a paradigm to design ambient intelligent applications". Em: *Intelligent Distributed Computing IV: Proceedings of the 4th International Symposium on Intelligent Distributed Computing-IDC 2010, Tangier, Morocco, September 2010*. Springer. 2010, pp. 3–9.

[13] Zia Babar. *LLM-Based Multi-Agent Systems*. 2024. URL: https://medium.com/@zbabar/llm-based-multi-agent-systems-62fd8c47f678.

[14] Bill O'Neill. *What is an LLM Agnostic Approach to AI Implementation?* https://quiq.com/blog/llm-agnostic-ai/. [Acesso em: 29 jul. 2025]. 2025.

[15] Fali Wang et al. "A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness". Em: *ACM Transactions on Intelligent Systems and Technology* (2024).

[16] Aniruddha Kembhavi et al. "A diagram is worth a dozen images". Em: *European conference on computer vision*. Springer. 2016, pp. 235–251.

[17] Pan Lu et al. "Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts". Em: *arXiv preprint arXiv:2310.02255* (2023).

[18] Ahmed Masry et al. "Chartqa: A benchmark for question answering about charts with visual and logical reasoning". Em: *Findings of the association for computational linguistics: ACL 2022*. 2022, pp. 2263–2279.

[19] Yuliang Liu et al. "Ocrbench: on the hidden mystery of ocr in large multimodal models". Em: *Science China Information Sciences* 67.12 (2024), p. 220102.

[20] Amanpreet Singh et al. "Towards vqa models that can read". Em: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 8317–8326.

[21] Tianrui Guan et al. "Hallusionbench: an advanced diagnostic suite for entangled language hallucination and visual illusion in large vision-language models". Em: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 14375–14385.

[22] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[23] Jacob Austin et al. "Program synthesis with large language models". Em: *arXiv preprint arXiv:2108.07732* (2021).

[24] Jeffrey Zhou et al. "Instruction-following evaluation for large language models". Em: *arXiv preprint arXiv:2311.07911* (2023).

[25] Shuai Bai et al. *Qwen3-VL Technical Report*. 2025. arXiv: 2511.21631 [cs.CV]. URL: https://arxiv.org/abs/2511.21631.

[26] Weiyun Wang et al. *InternVL3.5: Advancing Open-Source Multimodal Models in Versatility, Reasoning, and Efficiency*. 2025. arXiv: 2508.18265 [cs.CV]. URL: https://arxiv.org/abs/2508.18265.

[27] Tianyu Yu et al. *MiniCPM-V 4.5: Cooking Efficient MLLMs via Architecture, Data, and Training Recipe*. 2025. arXiv: 2509.18154 [cs.LG]. URL: https://arxiv.org/abs/2509.18154.

[28] Gemma Team et al. *Gemma 3 Technical Report*. 2025. arXiv: 2503.19786 [cs.CL]. URL: https://arxiv.org/abs/2503.19786.

[29] Kimi Team et al. *Kimi-VL Technical Report*. 2025. arXiv: 2504.07491 [cs.CV]. URL: https://arxiv.org/abs/2504.07491.

[30] Binyuan Hui et al. *Qwen2.5-Coder Technical Report*. 2024. arXiv: 2409.12186 [cs.CL]. URL: https://arxiv.org/abs/2409.12186.

[31] MiniCPM Team et al. *MiniCPM4: Ultra-Efficient LLMs on End Devices*. 2025. arXiv: 2506.07900 [cs.CL]. URL: https://arxiv.org/abs/2506.07900.

[32] Marah Abdin et al. *Phi-4 Technical Report*. 2024. arXiv: 2412.08905 [cs.CL]. URL: https://arxiv.org/abs/2412.08905.

[33] Tushar Aggarwal et al. "NextCoder: Robust Adaptation of Code LMs to Diverse Code Edits". Em: *Forty-second International Conference on Machine Learning*. 2025.

[34] OpenAI et al. *gpt-oss-120b gpt-oss-20b Model Card*. 2025. arXiv: 2508.10925 [cs.CL]. URL: https://arxiv.org/abs/2508.10925.

[35] Ollama. *Ollama: A Tool for Building AI Apps*. Accessed: 08 nov. 2025. URL: https://ollama.com/.

[36] Hugging Face. *Transformers Documentation (em Português)*. Accessed: 12 dez. 2025. URL: https://huggingface.co/docs/transformers/pt/index.

[37] vLLM. *vLLM Documentation*. Accessed: 12 dez. 2025. URL: https://docs.vllm.ai/en/latest/.

[38] Omar Khattab et al. "Dspy: Compiling declarative language model calls into self-improving pipelines". Em: *arXiv preprint arXiv:2310.03714* (2023).

[39] DSpy.ai. *Signatures*. Accessed: 12 dez. 2025. URL: https://dspy.ai/learn/programming/signatures/.

[40] DSpy.ai. *Module*. Accessed: 12 dez. 2025. URL: https://dspy.ai/learn/programming/modules/.

[41] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". Em: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[42] Shunyu Yao et al. "React: Synergizing reasoning and acting in language models". Em: *The eleventh international conference on learning representations*. 2022.

[43] Debalina Ghosh Paul, Hong Zhu e Ian Bayley. "Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review". Em: *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. 2024, pp. 87–94. DOI: 10.1109/AITest62860.2024.00019.

[44] Kishore Papineni et al. "Bleu: a method for automatic evaluation of machine translation". Em: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[45] Chin-Yew Lin. "Rouge: A package for automatic evaluation of summaries". Em: *Text summarization branches out*. 2004, pp. 74–81.

[46] Satanjeev Banerjee e Alon Lavie. "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments". Em: *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 2005, pp. 65–72.

[47] Maja Popović. "chrF: character n-gram F-score for automatic MT evaluation". Em: *Proceedings of the tenth workshop on statistical machine translation*. 2015, pp. 392–395.

[48] Ngoc Tran et al. "Does BLEU score work for code migration?" Em: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 165–176.

[49] Shuo Ren et al. "Codebleu: a method for automatic evaluation of code synthesis". Em: *arXiv preprint arXiv:2009.10297* (2020).

[50] Mikhail Evtikhiev et al. "Out of the BLEU: How should we assess quality of the Code Generation models?" Em: *Journal of Systems and Software* 203 (2023), p. 111741. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111741. URL: https://www.sciencedirect.com/science/article/pii/S016412122300136X.

[51] Atharva Naik. "On the limitations of embedding based methods for measuring functional correctness for code generation". Em: *arXiv preprint arXiv:2405.01580* (2024).

[52] Shuyan Zhou et al. "Codebertscore: Evaluating code generation with pretrained models of code". Em: *arXiv preprint arXiv:2302.05527* (2023).