

Agentes Inteligentes e Desenvolvimento de Software: Avaliação de um Sistema Multi-Agente com e sem Intervenção Humana

Intelligent Agents and Software Development: Evaluation of a
Multi-Agent System With and Without Human Intervention

Matheus Felipe de Morais Barros

mymb1@discente.ifpe.edu.br

Heitor Negromonte de Oliveira Charles

hnoc@discente.ifpe.edu.br

Flávio Rosendo da Silva Oliveira

flavio.oliveira@paulista.ifpe.edu.br

RESUMO

Com o avanço dos Modelos de Linguagem de Grande Escala, tornou-se possível delegar etapas do desenvolvimento de software a agentes inteligentes. No entanto, limitações como alucinações e erros lógicos ainda comprometem a confiabilidade desses sistemas. Este trabalho avalia um sistema multi-agente baseado em Inteligência Artificial Generativa em dois cenários: atuação autônoma e atuação com intervenção humana. Dois estudos de caso foram realizados, aplicando três agentes especializados — Validador de Requisitos, Desenvolvedor e Testador — a tarefas de programação. Métricas como complexidade ciclomática, Halstead, índice de manutenibilidade e cobertura de testes foram computadas e analisadas. A avaliação dos resultados obtidos indicou que a intervenção humana melhorou significativamente a qualidade do código, com destaque para o aumento da manutenibilidade e a redução do esforço cognitivo para entendê-lo. Conforme a avaliação realizada a partir dos estudos de caso, a capacidade humana de lidar com ambiguidade, ajustar desvios semânticos e aplicar decisões contextuais ainda são fatores úteis para complementar a atuação de agentes aplicados ao desenvolvimento de software.

Palavras-chave: Desenvolvimento de software, Agentes Inteligentes, Grandes Modelos de Linguagem, *Human-In-The-Loop*

ABSTRACT

The advancement of Large Language Models has made it possible to delegate stages of software development to intelligent agents. However, limitations such as hallucinations and logical errors still compromise the reliability of these systems. This study evaluates a multi-agent system based on Generative Artificial Intelligence in two scenarios: autonomous operation and operation with human intervention. Two case studies were conducted, applying three specialized agents — Requirements

Validator, Developer, and Tester — to programming tasks. The analysis considered metrics such as cyclomatic complexity, Halstead measures, maintainability index, and test coverage.

The results indicated that human intervention significantly improved code quality, especially by increasing maintainability and reducing the cognitive effort required for code comprehension. This improvement is attributed to the human ability to handle ambiguity, correct semantic deviations, and make context-based decisions — aspects still beyond the full autonomy of generative agents.

Keywords: Software Development, Intelligent Agents, Large Language Models, Human-In-The-Loop

1 INTRODUÇÃO

O desenvolvimento de software é uma atividade que exige esforço humano intensivo, tanto em conhecimento técnico quanto em tempo, conforme destacam (Pressman e Maxim, 2020). Com o avanço dos Modelos de Linguagem de Grande Escala, ou *Large Language Models (LLMs)*, como o GPT-4 e suas variantes otimizadas (OpenAI, 2024), surgiu uma nova possibilidade — delegar tarefas complexas, como a geração e validação de código, a agentes inteligentes autônomos. Esses agentes, ao operarem de forma cooperativa dentro de um sistema multi-agente, podem simular um ciclo completo de desenvolvimento de software, abrindo caminhos para a automação parcial ou total deste processo.

Entretanto, a eficácia desses sistemas ainda é objeto de investigação. Embora demonstrem capacidade para resolver problemas e gerar código funcional, muitos desses agentes apresentam limitações, como erros lógicos recorrentes e alucinações — isto é, resultados das inferências que parecem plausíveis, mas possuem incoerências ou estão incorretas (JI et al., 2022; Chen et al., 2021). Esse fenômeno é amplamente documentado na literatura e representa um dos principais desafios para a confiabilidade dos modelos de linguagem natural, especialmente em tarefas que exigem precisão, tal como programação.

Isso desperta reflexões importantes: até que ponto esses agentes conseguem, de fato, operar de maneira autônoma? Qual é o real impacto da intervenção humana nesse tipo de sistema? Embora este trabalho não tenha como objetivo esgotar tais questões, elas ajudam a contextualizar a relevância do conceito de *Human-in-the-Loop (HITL)*, abordagem que insere a supervisão humana estratégica durante o funcionamento do sistema (Amershi et al., 2014).

Este trabalho propõe a criação e avaliação de um sistema multi-agente baseado em Inteligência Artificial Generativa (GenAI), composto por três agentes

especializados — Validador de Requisitos, Desenvolvedor e Testador — operando em dois contextos: com e sem intervenção humana. Para isso, foram realizados dois estudos de caso envolvendo desafios de programação. O primeiro consistiu na implementação de uma calculadora com interface gráfica em Python. Já o segundo caso, propôs a implementação de um serviço de carrinho de compras, focado na manipulação de listas conforme regras específicas de adição e remoção de elementos.

A organização deste trabalho está dividida da seguinte forma: a Seção 2 apresenta o referencial teórico, abordando os conceitos que fundamentam a análise neste estudo, tais como sistemas multiagentes, modelos de linguagem generativos, a abordagem *HITL* e métricas de avaliação da qualidade de código adotadas. A Seção 3 apresenta os trabalhos relacionados à aplicação de agentes baseados em LLMs em tarefas de desenvolvimento de software, e que exploram autonomia versus supervisão humana. A Seção 4 detalha a metodologia adotada, incluindo a definição dos agentes, a implementação do sistema multi-agente com CrewAI e os dois cenários experimentais propostos. A Seção 5 apresenta os estudos de caso e os resultados obtidos, com base nas métricas definidas para comparar os dois cenários propostos. Por fim, a Seção 6 apresenta as conclusões do estudo e sugestões para pesquisas futuras.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Large Language Models

LLMs são redes neurais treinadas com grandes volumes de texto para prever a próxima palavra em uma sequência, o que as torna altamente eficazes na geração de linguagem natural. Esses modelos se baseiam em arquiteturas do tipo *Transformer*, que permitem o aprendizado de dependências contextuais complexas (Vaswani et al., 2017).

Além de aplicações em tradução, resposta a perguntas e chatbots, os LLMs vêm sendo utilizados para geração automática de código. Apesar de sua sofisticação, LLMs ainda apresentam limitações como alucinação de informações e produção de resultados incorretos mesmo quando parecem plausíveis (Ji et al., 2022).

2.2 GPT-4o-mini

O modelo GPT-4o-mini é uma variante compacta e otimizada da linha GPT-4 desenvolvida pela OpenAI. Ele mantém a capacidade de compreensão e geração contextual de linguagem, com destaque para sua eficiência computacional, o que o torna ideal para sistemas multi-agentes em ambientes com recursos limitados (OpenAI, 2024). Embora menos potente que o GPT-4 *full*, o GPT-4o-mini apresenta

desempenho competitivo em tarefas de raciocínio lógico e geração de código, especialmente quando usado com *prompts* estruturados. Além disso, seu custo de uso é significativamente menor. Por essas razões, o GPT-4o-mini foi adotado neste trabalho como base para a implementação dos agentes.

2.3 Agentes Inteligentes e Sistemas Multi-Agentes

Agentes inteligentes são sistemas computacionais capazes de perceber o ambiente, processar informações e tomar decisões de forma autônoma com base em objetivos definidos (Wooldridge, 2009). Quando organizados em sistemas multi-agentes, esses agentes interagem entre si para resolver tarefas complexas de forma distribuída.

Segundo Russell e Norvig (2013), os agentes podem ser classificados em diferentes tipos, conforme seu grau de sofisticação. Entre os principais, estão: (i) agentes reativos, que respondem diretamente a estímulos do ambiente; (ii) agentes baseados em modelo, que mantêm uma representação interna do ambiente; (iii) agentes baseados em objetivos, que escolhem ações com base em metas definidas; e (iv) agentes de aprendizado, que ajustam seu comportamento com base na experiência.

Neste trabalho, adotamos a abordagem de agentes baseados em objetivos, pois ela permite que os agentes tomem decisões orientadas a metas específicas. Essa escolha também facilita a análise comparativa entre os cenários, já que os resultados podem ser avaliados com base no grau de sucesso no cumprimento desses objetivos.

2.4 Métricas de Avaliação de Código

A avaliação da qualidade do código gerado pelos agentes inteligentes foi realizada com base em métricas objetivas, amplamente consolidadas na Engenharia de Software. Tais métricas permitem mensurar aspectos como complexidade, esforço de compreensão, manutenibilidade, cobertura de testes e taxa de sucesso na execução de testes unitários. Para este trabalho, foram utilizadas as seguintes métricas: complexidade ciclomática, métricas de Halstead, o índice de manutenibilidade (*Maintainability Index* – MI), a cobertura de testes e a taxa de sucesso dos testes unitários. Essas métricas foram selecionadas para possibilitar uma avaliação abrangente, considerando diferentes aspectos dos experimentos realizados, e serão detalhadas nas subseções a seguir.

2.4.1 Complexidade Ciclométrica

A complexidade ciclométrica foi proposta por McCabe (1976) como uma forma de mensurar o número de caminhos independentes existentes em um algoritmo. Essa métrica está relacionada ao número de decisões condicionais no código-fonte

como estruturas *if*, *while*, *for*, entre outras, e é amplamente utilizada para estimar o grau de dificuldade associado à testabilidade e compreensão de um programa.

Quanto maior o valor da complexidade ciclomática, mais difícil será testar, compreender e manter o código. A fórmula utilizada para o cálculo pode ser vista na equação 1, na qual: $V(G)$ representa a complexidade ciclomática do grafo de controle de fluxo, E é o número de arestas, N é o número de nós e P é o número de componentes conectados.

Equação 1 – Fórmula da complexidade ciclomática:

$$V(G) = E - N + 2P$$

2.4.2 Métricas de Halstead

As métricas de Halstead, desenvolvidas por Halstead (1977), baseiam-se na contagem de operadores e operandos presentes no código-fonte, com o objetivo de estimar o esforço cognitivo necessário para compreender um programa. Essas métricas também auxiliam na previsão de erros e na análise da manutenibilidade do software, e seus elementos estão descritos na Tabela 1. Nesse caso, n_1 é o número de operadores distintos, n_2 é o número de operandos distintos, N_1 é o total de operadores e N_2 é o total de operandos.

Tabela 1 – Fórmulas das Métricas de Halstead

Métrica	Fórmula
Vocabulário	$n = n_1 + n_2$
Comprimento	$N = N_1 + N_2$
Volume	$V = N \times \log_2(n)$
Dificuldade	$D = (n_1 / 2) \times (N_2 / n_2)$
Esforço	$E = D \times V$
Bugs estimados	$B = E^{(2/3)} / 3000$

2.4.3 Índice de Manutenibilidade

O índice de manutenibilidade (*Maintainability Index* – MI), introduzido por Oman (1994), foi concebido com o propósito de reduzir os custos associados à manutenção de software, oferecendo uma forma prática de mensurar a facilidade de manutenção de um programa. Posteriormente, Coleman et al. (1994) realizaram uma calibração desse índice, validando sua aplicação em ambientes industriais de desenvolvimento de software. O cálculo do MI é baseado em uma fórmula polinomial que considera diversos fatores por módulo, incluindo: a média da métrica de

Halstead (aveV), a média de linhas de código (aveLOC), a média da complexidade ciclomática estendida de McCabe (aveV(g')) e, de forma opcional, a proporção média de comentários por linha de código (perCM). A fórmula geral é expressa da na Equação 2

Equação 2 – Cálculo do índice de manutenibilidade (MI):

$$MI = 171 - (5,2 \times \ln(\text{aveV})) - (0,23 \times \text{aveV}(g')) - (16,2 \times \ln(\text{aveLOC})) + (50 \times \sin(\sqrt{2,4} \times \text{perCM}))$$

2.4.4 Cobertura de Testes e Taxa de Sucesso

A cobertura de testes é uma métrica utilizada para medir a proporção do código-fonte exercitada por testes automatizados. Essa análise permite identificar trechos não verificados e orientar melhorias na qualidade do software. Entre os tipos mais comuns, destacam-se a cobertura de instruções, de ramos, de caminhos e de condições (Ammann; Offutt, 2017).

Por sua vez, a taxa de sucesso representa a porcentagem de testes que produzem o resultado esperado, sendo um indicador direto da conformidade do sistema com os requisitos definidos.

3 TRABALHOS RELACIONADOS

O uso de agentes inteligentes no desenvolvimento de software tem ganhado destaque nos últimos anos, especialmente com a popularização dos *LLMs*. Diversos estudos têm buscado compreender a eficácia desses sistemas, tanto em contextos totalmente autônomos quanto na presença de supervisão humana, com o objetivo de aumentar a eficiência e a qualidade do processo de desenvolvimento.

3.1 Sistemas Multiagentes e Automação de Tarefas de Desenvolvimento

A proposta de utilizar sistemas multi-agentes para dividir etapas do desenvolvimento de software não é recente. Wooldridge e Jennings (2009) definem agentes como sistemas capazes de perceber seu ambiente e agir de maneira autônoma em direção a metas definidas. Quando estruturados em sistemas multi-agentes, esses agentes atuam de forma coordenada e distribuída na resolução de problemas complexos.

Entre as metodologias mais consolidadas para o desenvolvimento de sistemas multiagentes está a metodologia Gaia, proposta por Wooldridge, Jennings e Kinny (2000), que fornece uma estrutura formal para análise e design de sistemas compostos por múltiplos agentes com papéis bem definidos. Outra metodologia relevante é a Prometheus, desenvolvida por Padgham e Winikoff (2002), que descreve um processo sistemático desde a especificação dos requisitos até a implementação dos agentes.

Com o avanço dos *LLMs*, novas abordagens vêm sendo propostas para automatizar o ciclo de desenvolvimento utilizando agentes colaborativos. O *ChatDev* (Qin et al., 2023), por exemplo, simula uma empresa composta por agentes com diferentes papéis — como CEO (*Chief Executive Officer*), engenheiro de software e testadores — que interagem iterativamente para gerar, revisar e testar código. Já o *CodePori* (Rasheed et al., 2024) apresenta um sistema que prioriza dinamicamente os agentes conforme a tarefa, otimizando a divisão do trabalho com base em estratégias de coordenação. Esses trabalhos demonstram que a integração de *LLMs* com arquiteturas multiagentes pode simular estruturas organizacionais e automatizar etapas do desenvolvimento, aproximando-se de práticas reais da Engenharia de Software.

3.2 Integração de Human-in-the-Loop em Sistemas Inteligentes

A abordagem *HITL* propõe a inserção estratégica da supervisão humana em sistemas automatizados, especialmente aqueles baseados em inteligência artificial. O objetivo é aumentar a confiabilidade dos resultados, mitigar erros e alucinações, e permitir o refinamento contínuo de tarefas sensíveis (Amershi et al., 2014).

No contexto de agentes inteligentes, Akbar e Conlan (2024) exploram o uso de *HITL* em agentes personalizados proativos (*PIPAs*, do inglês *Proactive Intelligent Personalized Agents*), propondo mecanismos para ajustar a autonomia dos agentes com base em fatores como preferências dos usuários e níveis de incerteza do sistema. Embora o foco do estudo seja a atuação de agentes assistivos em ambientes pessoais, seus conceitos de supervisão humana são aplicáveis ao presente trabalho, no qual a intervenção humana é utilizada para validar e corrigir as ações dos agentes.

Outro exemplo relevante é o *HULA* (Takerngsaksiri et al., 2024), um *framework* que integra agentes baseados em *LLMs* para tarefas de planejamento e codificação, com e sem intervenção humana. Avaliado no ambiente real da Atlassian JIRA, o *HULA* demonstrou que a supervisão humana melhora significativamente a qualidade, clareza e funcionalidade do código gerado, reforçando os benefícios da abordagem *HITL* no ciclo de vida do software.

3.3 Relação com o trabalho proposto

O presente trabalho dialoga com as abordagens mencionadas ao propor uma arquitetura multi-agente baseada em *LLMs* com papéis especializados — validador de requisitos, desenvolvedor e testador — orquestrados por meio do *framework* CrewAI. A principal contribuição deste estudo está na comparação estruturada entre dois modos de operação: com e sem *HITL*.

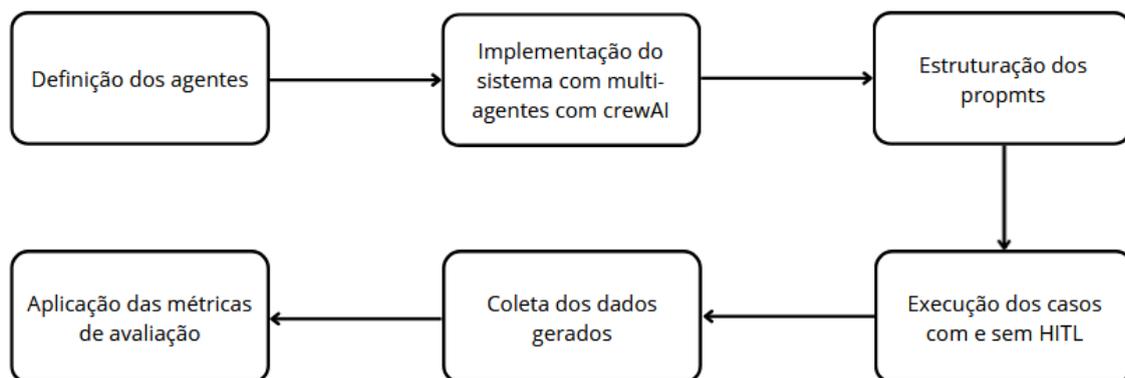
Enquanto o *HULA* explora *HITL* em um ambiente corporativo e os demais trabalhos simulam empresas ou priorizam estratégias de colaboração autônoma, este trabalho diferencia-se por avaliar comparativamente os dois modelos (i.e. autônomo e supervisionado) aplicados ao mesmo conjunto de tarefas, com base em métricas objetivas da Engenharia de Software, como complexidade ciclomática, métricas de Halstead, índice de manutenibilidade e cobertura de testes.

Dessa forma, este trabalho contribui para a análise experimental da autonomia de agentes em tarefas de codificação, destacando a relevância da supervisão humana estratégica em sistemas baseados em LLMs, com foco na construção de software confiável e sustentável.

4 METODOLOGIA

Este trabalho adota uma abordagem experimental, com base em estudos de casos. O objetivo foi avaliar a aplicação de um sistema multi-agente baseado em *LLMs* em dois cenários distintos: com e sem supervisão humana, segundo os princípios da abordagem *HITL*. O experimento envolveu a execução controlada de tarefas de desenvolvimento de software pelos agentes, utilizando um problema comum como referência para ambos os cenários. A metodologia foi estruturada em etapas sequenciais, desde a preparação do sistema até a coleta e avaliação dos dados gerados. As subseções a seguir detalham cada uma das etapas realizadas em cada estudo de caso, incluindo a implementação do sistema, os critérios de entrada, a execução dos cenários, o registro dos dados e a aplicação das métricas de avaliação. A Figura 1 apresenta uma visão geral do fluxo metodológico adotado.

Figura 1 – Fluxograma metodológico realizado neste trabalho



Fonte: Elaborada pelos autores (2025)

4.1 Definição dos Agentes

Nesta etapa, foram definidos os três agentes especializados que compõem o sistema multi-agente: Validador de Requisitos, Desenvolvedor e Testador, cujas funções refletem as três fases cruciais do ciclo clássico em cascata (Royce, 1970): especificação, implementação e validação. Optou-se por essa divisão mínima para manter o experimento controlável e focado nos pontos de maior impacto no processo de engenharia de software, permitindo comparar de forma objetiva a atuação autônoma dos agentes com a supervisão humana. A Tabela 2 mostra os agentes e seus respectivos objetivos dentro do sistema.

Tabela 2 – Objetivo dos Agentes

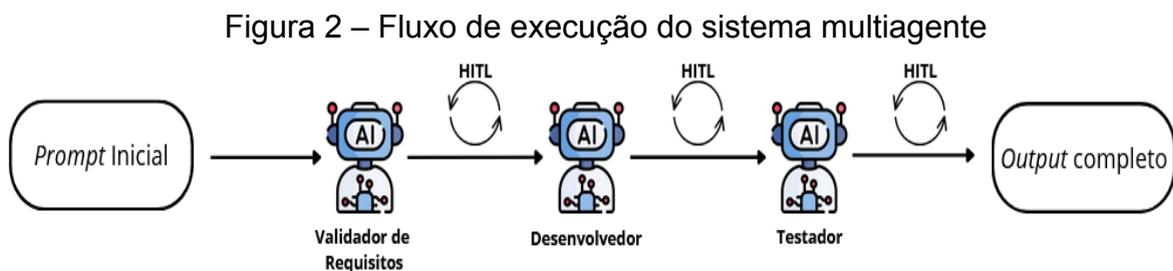
Agente	Objetivo
Validador de Requisitos	Garantir a completude, clareza e viabilidade dos requisitos de software.
Desenvolvedor	Implementar código de alta qualidade e bem estruturado com base nos requisitos validados.
Testador	Gerar Testes Unitários.

Fonte: Elaborada pelos autores(2025)

4.2 Implementação do Sistema com CrewAI

O sistema multi-agente foi implementado com o *framework* CrewAI, que permite a orquestração de agentes baseados em *LLMs*, com definição de papéis, controle de fluxo e troca de mensagens entre os componentes (CREWAI, 2024).

A arquitetura adotada segue o modelo de desenvolvimento em cascata (ROYCE, 1970), no qual a saída de cada agente serve como entrada para o próximo. A Figura 2 ilustra a execução do sistema, destacando a sequência entre os agentes e os pontos de possível intervenção humana no fluxo (*HITL*).



Fonte: Elaborada pelos autores(2025)

4.3 Estruturação dos Prompts

O sistema foi alimentado com *prompts* elaborados a partir de dois estudos de caso distintos, que serviram como base para a execução dos agentes nos dois cenários avaliados (com e sem *HITL*). Cada *prompt* apresenta um problema específico de desenvolvimento de software a ser solucionado pelos agentes. O primeiro estudo de caso refere-se ao desenvolvimento de uma calculadora com interface gráfica em Python. O segundo contempla a implementação de um serviço de carrinho de compras, com ênfase na manipulação de listas.

A seleção desses estudos buscou contemplar diferentes níveis de complexidade e aspectos característicos da programação, desde a integração com interfaces gráficas até a lógica de negócios e o tratamento de estruturas de dados. Importa salientar que os *prompts* empregados foram utilizados de forma idêntica em ambos os cenários experimentais. Os *prompts* utilizados encontram-se descritos no Quadro 1.

Quadro 1 - *Prompts* utilizados

Caso	Prompt
Calculadora	Desenvolva um código em Python para criar uma calculadora com interface gráfica usando Tkinter. A interface deve conter um display para exibir os números digitados e o resultado das operações. Deve haver botões para os números 0 a 9, além das operações soma (+), subtração (-), multiplicação (×) e divisão (÷). O botão "=" deve calcular o resultado e o botão "C" deve limpar o display.
Carrinho de Compras	Desenvolva uma implementação de um serviço de carrinho de compras. A função deve receber dois arrays de inteiros: itens, com os IDs inicialmente no carrinho, e query, com operações a serem aplicadas. Regras: Valores positivos em query: adicionar o item ao final do carrinho. Valores negativos: remover a primeira ocorrência do item correspondente (valor absoluto). Garantias: Nenhum valor em query será zero. O carrinho final não estará vazio. A função deve retornar o estado final do carrinho como lista de inteiros. Exemplo: itens = [2, 3, 4]. query = [1, -3, 4] Resultado final: [2, 4, 1, 4]

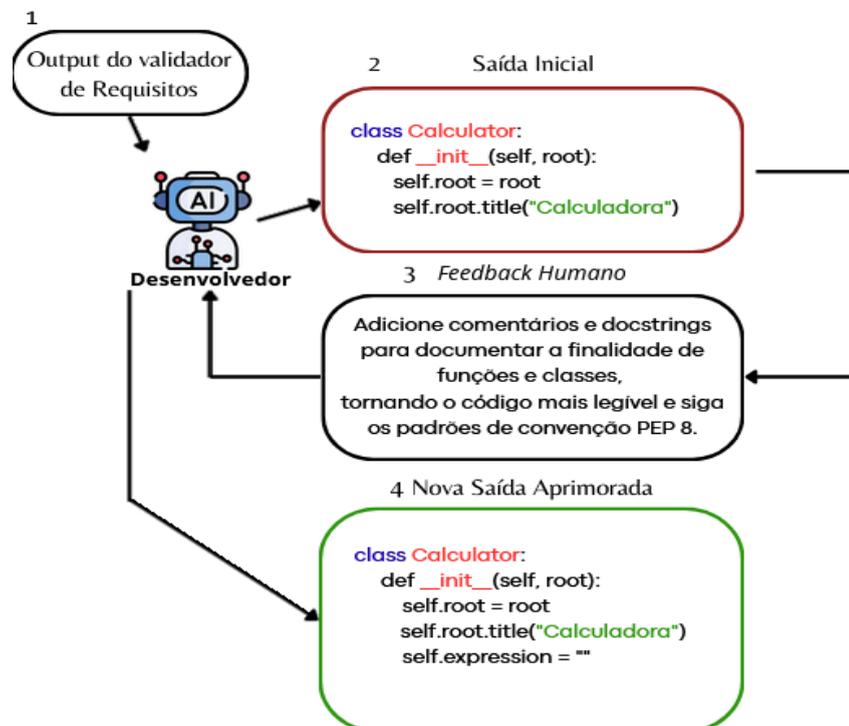
4.4 Execução dos Casos com e sem HITL

No cenário sem *HITL*, os agentes atuaram de forma totalmente autônoma, sem qualquer intervenção externa durante a execução. Cada agente recebeu como entrada a saída bruta do agente anterior, processou sua etapa conforme seu papel no fluxo e produziu sua própria saída. Todo o ciclo foi conduzido de forma linear, e os dados gerados em cada fase foram coletados exatamente como produzidos pelos agentes.

No cenário com *HITL*, a dinâmica de execução incorporou pontos de controle estratégicos ao longo do processo. Após a execução de cada agente, sua saída era apresentada ao operador humano, que atuava como supervisor técnico. Esse operador avaliava a resposta do agente e decidia se o conteúdo estava suficientemente adequado para prosseguir à etapa seguinte. Quando necessário, eram fornecidos feedbacks textuais ao agente, que reprocessava a tarefa com base nesse novo *input*, gerando uma saída ajustada e mais alinhada às expectativas humanas.

Esse ciclo de validação e refinamento podia ser repetido quantas vezes fossem necessárias, até que o operador considerasse a resposta satisfatória. A Figura 3 ilustra esse processo.

Figura 3 - Ciclo de execução com *HITL*



Fonte: Elaborada pelos autores(2025)

4.5 Aferição das Métricas de Avaliação

As métricas utilizadas neste trabalho foram aferidas com base nos artefatos de código produzidos pelos agentes. A coleta desses artefatos foi realizada manualmente, a partir das saídas geradas durante a execução dos estudos de caso.

As métricas de qualidade de código — incluindo complexidade ciclomática, índice de manutenibilidade (*MI*) e métricas de Halstead — foram calculadas a partir do código-fonte gerado pelo agente Desenvolvedor. Para isso, utilizou-se a biblioteca Python *Radon* (Radon, 2025), que realiza análise estática e fornece essas métricas de forma automatizada e padronizada.

A cobertura de testes, por sua vez, foi aferida sobre os testes unitários gerados pelo agente testador. Nesse caso, utilizou-se a biblioteca *Coverage* (Batchelder, 2025).

5 ESTUDOS DE CASO

Esta seção apresenta os resultados obtidos na execução dos estudos de caso, considerando dois cenários distintos: com e sem intervenção humana. O código-fonte e todos os registros com resultados e métricas utilizados neste trabalho encontram-se disponíveis no repositório mencionado no Apêndice A.

5.1 – Estudo de Caso 1: Calculadora com Interface Gráfica

O primeiro estudo de caso teve como objetivo o desenvolvimento de uma calculadora com interface gráfica utilizando a biblioteca Tkinter, em Python. A seguir, apresentamos os principais resultados obtidos com e sem a presença da supervisão humana durante o processo.

5.1.1 Complexidade Ciclométrica

Ambas as abordagens apresentaram baixa complexidade, classificadas como A na Tabela 3. Essa classificação foi atribuída automaticamente pela biblioteca *Radon* (RADON, 2025), que utiliza uma escala de risco. Valores de 1 a 5 indicam blocos simples e de baixo risco, enquanto valores entre 11 e 20 indicam risco moderado. Entre 30 e 40, o risco é alto, com maior complexidade e propensão a erros. Acima de 41, o risco é muito alto, com blocos complexos e instáveis.

Tabela 3 - Resultado da métrica de complexidade ciclométrica.

Execução	Resultado
Sem intervenção	A (2,25)
Com <i>HITL</i>	A (2,14)

Fonte: Elaborada pelos autores (2025)

5.1.2 Índice de Manutenibilidade

O índice de manutenibilidade melhorou com a introdução do *HITL*, conforme indicado na tabela 4. Segundo a classificação adotada por ferramentas como o Visual Studio, valores entre 20 e 100 são considerados indicativos de boa manutenibilidade (Microsoft, 2025). Embora ambos os resultados estejam dentro dessa faixa, a elevação do índice sugere uma melhora na estrutura do código, com maior clareza e facilidade de manutenção.

Tabela 4 - Resultado da métrica do índice de Manutenibilidade

Execução	Resultado
Sem <i>HITL</i>	61,38
Com <i>HITL</i>	73,98

Fonte: Elaborada pelos autores (2025)

5.1.3 Métricas de Halstead

A versão com *HITL* apresentou melhorias em diversas métricas, conforme indicado na Tabela 5. Houve redução no volume e no esforço cognitivo, sugerindo um código mais conciso. Também foi registrada uma queda no número estimado de erros (*bugs*), o que reforça o papel da supervisão humana no refinamento das soluções.

Tabela 5 - Resultado das métricas de halstead

Métrica	Sem <i>HITL</i>	Com <i>HITL</i>
Vocabulário	20	14
Comprimento (Length)	30	21
Comprimento Calculado	74,24	42,81
Volume	129,66	79,95
Dificuldade	1,76	1,91
Esforço (Effort)	228,81	152,64
Tempo Estimado (s)	12,71	8,48
Erros Estimados	0,0432	0,0267

Fonte: Elaborada pelos autores (2025)

5.1.4 Cobertura e Taxa de Sucesso dos testes unitários

Tabela 6 - Resultado da Cobertura e Taxa de Sucesso dos testes unitários

Métrica	Sem HITL	Com HITL
Cobertura de Testes	94%	90%
Total de Testes Executados	6	8
Testes Aprovados	2	8
Testes Reprovados	4	0
Taxa de Sucesso	33,3%	100%

Fonte: Elaborada pelos autores (2025)

Os agentes de forma autônoma apresentaram falhas críticas em operações matemáticas básicas, comprometendo a funcionalidade geral da calculadora. Apenas 2 dos 6 testes foram aprovados, revelando problemas recorrentes de alucinação — um fenômeno comum em modelos de IA, no qual o sistema gera respostas incorretas, mas aparentemente plausíveis (JI et al., 2022). A Figura 4 ilustra o trecho de código do agente desenvolvedor responsável pelas falhas durante a execução dos testes. Esse trecho evidencia uma lógica inadequada na construção da expressão final, resultando em concatenações incorretas dos valores inseridos. Tal comportamento compromete diretamente a precisão dos cálculos e a confiabilidade da aplicação.

Figura 4 - Trecho de código incorreto gerado pelo agente desenvolvedor

```
def calculate(self):
    if self.current_input and self.result:
        if self.current_input == '÷' and self.display.get() == '0':
            messagebox.showerror("Erro", "Divisão por zero não é permitida.")
            self.clear()
            return
        expression = self.result + self.current_input + self.display.get()
    try:
        expression = expression.replace('x', '*').replace('÷', '/')
        calculation_result = eval(expression)
        self.clear()
        self.display.insert(tk.END, calculation_result)
    except Exception as e:
        messagebox.showerror("Erro", "Entrada inválida.")
        self.clear()
```

Fonte: Elaborada pelos autores (2025)

A Tabela 7 a seguir ilustra a evolução dos valores armazenados nos atributos destacados na figura 4 durante a simulação de uma operação básica de soma na calculadora.

Tabela 7 - Simulação de operação de soma na calculadora sem HITL

Ação	<i>self.result</i>	<i>self.current_input</i>	<i>dispaly.get()</i>
Digita 5	none	"5"	"5"
Aperta +	"5"	"+"	"5+"
Digita 6	"5"	"6"	"5+6"
Aperta =	"5"	"6"	"5+6"

Fonte: Elaborada pelos autores (2025)

Observa-se que, ao final do processo, a expressão construída no método `calculate` apresenta uma concatenação redundante, resultando na expressão `5+6+6` em vez da esperada `5+6`. Isso indica uma falha na lógica de atualização dos valores internos. Em vez de montar corretamente a operação, o código do agente desenvolvedor duplica o último operando, comprometendo a precisão do cálculo.

Apesar de simples na aparência, esse erro revela uma falha no controle de estado da calculadora, que não diferencia corretamente a entrada do usuário da expressão em construção. Situações como essa acabam exigindo uma análise mais detalhada do fluxo interno da aplicação.

Com a introdução da intervenção humana, houve uma melhora significativa na confiabilidade e na abrangência dos testes da aplicação: todos os 8 testes executados foram aprovados, evidenciando que a supervisão foi essencial para identificar possíveis inconsistências e garantir o funcionamento correto do sistema.

5.1.5 Feedbacks Realizados

O Quadro 2 apresenta os feedbacks fornecidos a cada agente durante a execução no cenário com HITL. Cada agente recebeu um único feedback ao longo do processo, com o objetivo de ajustar sua atuação e aprimorar os resultados obtidos.

Quadro 2 - *Feedbacks* fornecidos aos agentes

Agente	Feedback
validador	Especifique que será respeitada a ordem matemática no caso de operações encadeadas (exemplo: $2 + 2 * 3$). No tratamento de erros, o sistema deve exibir mensagens claras e amigáveis sempre que ocorrerem erros durante a operação, como uma divisão zero. Além disso, o código deve seguir boas práticas de programação e as convenções estabelecidas para Python.
desenvolvedor	O código parece bem estruturado para a finalidade proposta de acordo com os requisitos solicitados, apenas Adicione comentários e docstrings para documentar a finalidade de funções e classes, tornando o código mais legível e siga os padrões de convenção PEP 8
Testador	Especifique que será aceito tanto o valor inteiro quanto o valor flutuante no teste da divisão (exemplo: aceitar 2 e 2.0). No teste de divisão por zero, o código espera uma exceção, mas isso não vai funcionar; ajuste para não esperar uma exceção, já que o método trata o erro internamente. Em vez disso, utilize o <code>unittest.mock.patch</code> para interceptar a chamada ao <code>messagebox.showerror</code> e verifique se a mensagem é exibida corretamente

O feedback direcionado ao agente testador teve papel fundamental, ao buscar corrigir imprecisões técnicas que poderiam comprometer a etapa de testes. Da mesma forma, as intervenções realizadas junto ao Desenvolvedor e ao Testador contribuíram para a geração de um artefato final mais conciso e menos verboso. Tais ajustes impactaram positivamente as métricas analisadas, evidenciando que a supervisão humana, ainda que pontual como no caso do desenvolvedor e validador, exerceu influência na qualidade dos artefatos gerados.

5.2 Estudo de Caso 2: Carrinho de compras

O segundo estudo de caso propôs a implementação de um serviço de carrinho de compras, focado na manipulação de listas conforme regras específicas de adição e remoção de elementos. A seguir, estão os principais resultados obtidos com e sem a presença da supervisão humana durante o processo.

5.2.1 Complexidade Ciclométrica

Em ambos os cenários, as abordagens apresentaram níveis baixos de complexidade como observado na tabela 8, classificados como A pela biblioteca Radon, conforme detalhado na Seção 5.1.1. No entanto, a intervenção humana resultou em um código mais enxuto e direto.

Tabela 8 - Complexidade Ciclométrica. Estudo de caso 2

Execução	Resultado
Sem <i>HITL</i>	A (3,0)
Com <i>HITL</i>	A (2,67)

Fonte: Elaborada pelos autores (2025)

5.2.2 Índice de Manutenibilidade

A intervenção humana contribuiu para um incremento expressivo na manutenibilidade do código como evidenciado na tabela 9, mostrando melhorias na sua clareza estrutural, modularidade e facilidade de modificação.

Tabela 9 - Índice de Manutenibilidade. Estudo de caso 2

Execução	Resultado
Sem <i>HITL</i>	49,28
Com <i>HITL</i>	86,14

Fonte: Elaborada pelos autores (2025)

5.2.3 Métricas de Halstead

A intervenção humana trouxe impactos positivos expressivos nas métricas de Halstead como mostrado na tabela 10. A redução no vocabulário e no comprimento do código gerado indica uma solução mais enxuta. Conseqüentemente, o volume de informação necessário para compreender o sistema também foi significativamente reduzido. Além disso, o esforço estimado para a compreensão do código caiu de 484,06 para 97,63, representando uma diminuição de aproximadamente 80%. O tempo necessário para entendimento e análise do programa também foi drasticamente reduzido, de 26,89 segundos para apenas 5,42 segundos. Outro ponto de destaque é a redução no número estimado de erros (*bugs*), que caiu de 0,0375 para 0,0116. Essa melhoria reforça a maior confiabilidade e qualidade do código resultante após a intervenção humana.

Tabela 10 - Métricas de Halstead. Estudo de Caso 2

Métrica	Sem <i>HITL</i>	Com <i>HITL</i>
Vocabulário	20	9
Comprimento (Length)	26	11
Comprimento Calculado	67,76	19,61
Volume	112,37	34,87
Dificuldade	4,31	2,8
Esforço (Effort)	484,06	97,63
Tempo Estimado (s)	26,89	5,42
Erros Estimados (Bugs)	0,0375	0,0116

Fonte: Elaborada pelos autores (2025)

5.2.4 Cobertura e Taxa de Sucesso dos testes unitários

No cenário sem *HITL*, o agente autônomo apresentou falhas em 3 dos 8 testes executados, com cobertura de 82% e taxa de sucesso de 62,5%. Já com a intervenção humana, observou-se uma melhora significativa: todos os testes foram aprovados e a cobertura atingiu 100%, demonstrando maior eficácia na validação das funcionalidades do serviço de carrinho de compras.

Esses resultados indicam que a presença humana contribuiu positivamente para a precisão e abrangência dos testes, reduzindo falhas e assegurando que o comportamento do agente estivesse de acordo com os requisitos definidos.

Tabela 11 - Resultados da cobertura e taxa de sucesso dos testes unitários

Métrica	Sem <i>HITL</i>	Com <i>HITL</i>
Cobertura de Testes	82%	100%
Total de Testes Executados	8	7
Testes Aprovados	5	7
Testes Reprovados	3	0
Taxa de Sucesso	62,5%	100%

Fonte: Elaborada pelos autores (2025)

5.2.5 Feedbacks Realizados

O Quadro 3 apresenta os feedbacks fornecidos a cada agente durante a execução no cenário com *HITL*. Assim como no Caso 1 (descrito no item 5.1.5), cada agente recebeu apenas um feedback ao longo do processo, com o objetivo de ajustar sua atuação e aprimorar os resultados obtidos.

Agente	Feedback
validador	percebo que você adicionou pontos extras como requisitos não funcionais detalhados (tratamento de exceções, desempenho com grandes listas e documentação), que, apesar de úteis, não foram explicitamente pedidos. Para tornar sua resposta ainda mais direta ao ponto e alinhada ao prompt original, recomendo focar estritamente no que foi solicitado
desenvolvedor	Use nomes mais descritivos, extraia a aplicação de cada operação para funções auxiliares
Testador	Você cometeu um equívoco no teste da função: <code>test_add_and_remove_items</code> , <code>expected_cart</code> deveria ser <code>[2, 4, 1, 4]</code> de acordo com os itens iniciais e as operações que você definiu e eu também gostaria de ver um teste para remoção única de elementos ex: <code>initial_items = [1, 1, 2]</code> e <code>operations = [-1]</code> deve resultar em <code>[1, 2]</code> , removendo só a primeira ocorrência.

A intervenção do testador foi fundamental, assim como no Caso 1, para corrigir inconsistências que poderiam comprometer a execução dos testes. O feedback do validador também teve papel relevante ao contribuir para que o levantamento de requisitos permanecesse claro e conciso, evitando a inclusão de elementos não solicitados. Essa objetividade alcançada a partir da intervenção impactou diretamente nas fases seguintes, refletindo em um código mais enxuto e focado. A Figura 5 e 6 ilustra essa evolução do artefato gerado pelo validador, comparando o artefato gerado antes e depois da intervenção. Já o feedback ao desenvolvedor teve um foco mais específico na melhoria da legibilidade do código.

Figura 5 – Requisitos do agente validador antes do feedback humano

<p>1. Descrição Geral do Sistema O sistema implementa a lógica de um serviço de gerenciamento de carrinho de compras em Python, permitindo manipulações baseadas em um conjunto de operações definido no array query.</p> <p>2. Requisitos Funcionais</p> <ul style="list-style-type: none"> • RF1. Entrada: array de inteiros items (IDs dos itens atualmente no carrinho). • RF2. Entrada: array de inteiros query (operações a serem aplicadas). • RF3. Se $query[i] > 0$, adicionar $query[i]$ ao final de items. • RF4. Se $query[i] < 0$, remover a primeira ocorrência de $query[i]$ de items, caso exista. • RF5. Impedir que o carrinho fique vazio após todas as operações; neste caso, lançar erro. • RF6. Garantir que nenhum valor em query seja igual a zero; caso contrário, lançar erro. • RF7. Retornar, ao final, o array items atualizado com todas as operações aplicadas. <p>3. Requisitos Não Funcionais</p> <ul style="list-style-type: none"> • RNF1. Implementar em Python, seguindo boas práticas de codificação (legibilidade, modularidade e testes). • RNF2. Complexidade de tempo $O(n)$ em relação ao número de operações em query. • RNF3. Fornecer mensagens de erro claras e específicas sempre que uma operação inválida for detectada. 	<p>4. Requisitos de Desempenho</p> <ul style="list-style-type: none"> • RDP1. O sistema deve processar e retornar resultados para um array items de até 10 000 itens e um array query de até 10 000 operações em tempo razoável (≤ 2 s de latência). <p>5. Requisitos de Teste</p> <ul style="list-style-type: none"> • RT1. Testar casos de borda, incluindo: <ul style="list-style-type: none"> • Adição de elementos em um carrinho já cheio. • Remoção de elementos que não estão no carrinho. • Manipulação de um carrinho nos limites mínimos e máximos de itens válidos. • RT2. Verificar todos os fluxos de operação para garantir que o carrinho nunca fique vazio após a aplicação completa de query. <p>6. Considerações de Implementação</p> <ul style="list-style-type: none"> • Incluir tratamento de exceções para operações inválidas de forma controlada, fornecendo feedback claro ao usuário. • Documentar todas as funções principais, com comentários explicativos do fluxo de lógica no código.
--	--

fonte elaborada pelos autores (2025)

Figura 6 – Requisitos do agente validador depois do feedback humano

<p>1. Objetivo do Sistema Desenvolver em Python uma função que gerencie um carrinho de compras, permitindo adicionar e remover itens conforme uma lista de operações.</p> <p>2. Requisitos Funcionais</p> <p>2.1 Parâmetros de Entrada</p> <ul style="list-style-type: none"> • items (list[int]): IDs dos itens inicialmente presentes no carrinho. • query (list[int]): Operações a serem aplicadas ao carrinho. <p>2.2 Regras de Operação</p> <ul style="list-style-type: none"> • Adição ($x > 0$): adiciona o item de ID x ao final do carrinho. • Remoção ($x < 0$): remove a primeira ocorrência do item de ID x (valor absoluto), se presente. <p>2.3 Restrições</p> <ul style="list-style-type: none"> • Não pode haver valores iguais a zero em query. • O carrinho não pode ficar vazio após aplicar todas as operações. <p>2.4 Saída</p> <ul style="list-style-type: none"> • Retornar uma lista de inteiros contendo o estado final de items depois de processadas todas as operações. 	<p>3. Casos de Uso Exemplo - Entrada: items = [2, 3, 4] query = [1, -3, 4] Saída esperada: [2, 4, 1, 4]</p> <p>4. Validação de Requisitos</p> <ul style="list-style-type: none"> • Todos os itens acima foram revisados para clareza e viabilidade técnica. • Não existem ambiguidades: as regras e restrições estão bem definidas para garantir o comportamento esperado.
--	--

fonte elaborada pelos autores (2025)

5.3 Análise Comparativa entre os Estudos de Caso

No primeiro estudo de caso, a atuação autônoma dos agentes levou a falhas lógicas críticas nas operações matemáticas da calculadora, com apenas 33,3% dos testes unitários aprovados. Já no segundo estudo de caso, mesmo sem a

intervenção, o sistema ainda conseguiu executar parte das funções corretamente, atingindo 62,5% de sucesso nos testes. Ainda assim, em ambos os casos, a introdução do *HITL* elevou a taxa de sucesso nos testes para 100%, eliminando falhas e inconsistências.

Em relação às métricas de qualidade de código, o *HITL* promoveu avanços consistentes nos dois estudos de caso. Houve aumento do índice de manutenibilidade e diminuição da complexidade ciclomática média em ambas as aplicações. No segundo estudo, o ganho em manutenibilidade foi ainda mais expressivo, passando de 49,28 para 86,14, evidenciando uma melhoria mais significativa na organização e na clareza do código.

As métricas de Halstead também reforçam essa tendência de melhoria. No primeiro caso, o esforço cognitivo estimado caiu cerca de 33% com a intervenção humana, enquanto no segundo caso a redução foi de aproximadamente 80%. Esses resultados indicam que, à medida que a carga estrutural e as possibilidades de erro lógico aumentam, o impacto do *HITL* na qualidade final do código se torna ainda mais relevante.

A análise dos dois estudos de caso evidencia o impacto positivo da intervenção humana na qualidade, estabilidade e manutenibilidade dos sistemas desenvolvidos. Em ambos os cenários, a calculadora com interface gráfica e o carrinho de compras, a supervisão ao longo do processo resultou em códigos mais eficientes, claros e confiáveis.

De forma geral, a comparação entre os dois estudos permite afirmar que a necessidade e o impacto do *HITL* aumentam conforme cresce a complexidade dos requisitos e a liberdade de interpretação na tarefa. Em cenários onde os agentes têm mais liberdade de decisão e precisam lidar com o uso ou a manipulação de bibliotecas específicas — como no caso da calculadora com interface gráfica — a supervisão humana se mostrou fundamental para garantir a confiabilidade dos artefatos gerados.

6. CONCLUSÃO

Este trabalho propôs a avaliação experimental de um sistema multi-agente baseado em LLMs aplicado ao desenvolvimento de software, considerando dois cenários: atuação autônoma dos agentes e atuação com intervenção humana. Foram implementados dois estudos de caso distintos, uma calculadora com interface gráfica e um serviço de carrinho de compras, analisando-se métricas objetivas de qualidade de código, como complexidade ciclomática, índice de manutenibilidade, métricas de Halstead e cobertura de testes.

Os resultados obtidos evidenciaram que o *HITL* exerce um impacto positivo consistente na qualidade dos artefatos gerados. Em ambos os estudos de caso, a supervisão resultou em melhorias significativas na organização, clareza e confiabilidade do código.

Embora os agentes autônomos tenham demonstrado capacidade de gerar soluções funcionais, os experimentos reforçam que a presença humana no ciclo de desenvolvimento foi fundamental para corrigir inconsistências, reduzir erros e assegurar a qualidade do produto final.

Este trabalho contribui para o entendimento prático das potencialidades e limitações dos agentes inteligentes no desenvolvimento de software, além de reforçar a importância da integração entre IA e supervisão humana em sistemas de apoio à Engenharia de Software.

Como limitação deste estudo, destaca-se a quantidade reduzida de execuções experimentais, o que restringe a generalização dos resultados. Trabalhos futuros poderão explorar uma avaliação mais ampla, envolvendo diferentes tipos de tarefas de programação, variações nos perfis dos agentes e metodologias de intervenção humana mais sofisticadas, assim como explorar o uso de outros LLMs.

REFERÊNCIAS

AKBAR, A.; CONLAN, O. Towards integrating Human-in-the-loop control in proactive intelligent personalised agents. In: ACM CONFERENCE ON USER MODELING, ADAPTATION AND PERSONALIZATION, 32., 2024, [S. l.]. Adjunct Proceedings.... New York: ACM, 2024. Disponível em: <https://doi.org/10.1145/3631700.3664903>. Acesso em: 9 abr. 2025.

AMERSHI, S. et al. Power to the people: The role of humans in interactive machine learning. **AI Magazine**, v. 35, n. 4, p. 105–120, 2014.

AMMANN, Paul; OFFUTT, Jeff. Introduction to software testing. 2. ed. Cambridge: Cambridge University Press, 2017.

BATCHELDER, Ned. Coverage.py: code coverage measurement for Python. Disponível em: <https://coverage.readthedocs.io/>. Acesso em: 25 maio 2025.

CHEN, Mark *et al.* Evaluating large language models trained on code. 2021. Disponível em: <https://arxiv.org/abs/2107.03374>. Acesso em: 15 maio 2025.

CREWAI. Framework for building multi-agent LLM systems. Disponível em: <https://www.crewai.com>. Acesso em: 12 mar. 2025.

COLEMAN, D.; ASH, D.; LOWTHER, B.; OMAN, P. Using metrics to evaluate software system maintainability. **IEEE Computer**, v. 27, n. 8, p. 44–49, ago. 1994.

HALSTEAD, M. H. Elements of software science. New York: Elsevier, 1977.

Jl, Z. et al. Survey of hallucination in natural language generation. **ACM Computing Surveys**, v. 55, n. 12, p. 1–38, 2023. Publicado anteriormente como pré-impressão em: arXiv:2202.03629, 2022.

MCCABE, T. J. A complexity measure. **IEEE Transactions on Software Engineering**, v. SE-2, n. 4, p. 308–320, dez. 1976.

MICROSOFT. Code metrics – maintainability index range and meaning. 10 jan. 2025. Disponível em: <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022>. Acesso em: 6 jul. 2025.

OMAN, P. W.; HAGEMEISTER, J. R. Construction and testing of polynomials predicting software maintainability. **Journal of Systems and Software**, v. 24, n. 3, p. 251–266, 1994. DOI: 10.1016/0164-1212(94)90067-1.

OPENAI. GPT-4 technical report. San Francisco: OpenAI, 2024. Disponível em: <https://openai.com/research/gpt-4>. Acesso em: 12 mar. 2025.

PADGHAM, L.; WINIKOFF, M. Prometheus: a methodology for developing intelligent agents. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 3., 2002, [S. l.]. Proceedings.... New York: ACM, 2002. p. 37–38.

PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de software: uma abordagem profissional. 9. ed. São Paulo: McGraw-Hill, 2020.

QIN, C. et al. ChatDev: revolutionizing software development with AI agents. arXiv preprint, arXiv:2307.07924, 2023. Disponível em: <https://arxiv.org/abs/2307.07924>. Acesso em: 9 abr. 2025.

RADON. Radon: Python tool for code metrics. Disponível em: <https://radon.readthedocs.io/>. Acesso em: 25 maio 2025.

RASHEED, Zeeshan et al. CodePori: large scale model for autonomous software development by using multi-agents. arXiv preprint, arXiv:2402.01411v1, 2024. Disponível em: <https://arxiv.org/abs/2402.01411v1>. Acesso em: 9 abr. 2025.

ROYCE, Winston W. Managing the development of large software systems. In: IEEE WESTERN ELECTRONIC SHOW AND CONVENTION – WESCON, 1970, Los Angeles. Proceedings.... Los Angeles: IEEE, 1970.

RUSSELL, Stuart J.; NORVIG, Peter. Inteligência artificial. Tradução de Regina Célia Simille. Rio de Janeiro: Elsevier, 2013. Tradução de: Artificial intelligence: a modern approach. p. 77–88.

TAKERNGSAKSIRI, W. *et al.* Human-in-the-loop software development agents. *arXiv preprint*, arXiv:2411.12924, 2024. Disponível em: <https://arxiv.org/abs/2411.12924>. Acesso em: 9 abr. 2025.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: theory and practice. **The Knowledge Engineering Review**, v. 10, n. 2, p. 115–152, 1995.

WOOLDRIDGE, M.; JENNINGS, N. R.; KINNY, D. The Gaia methodology for agent-oriented analysis and design. **Autonomous Agents and Multi-Agent Systems**, v. 3, n. 3, p. 285–312, 2000.

WOOLDRIDGE, M. An introduction to multiagent systems. 2. ed. Chichester: John Wiley & Sons, 2009.

APÊNDICES

APÊNDICE A – CÓDIGO-FONTE E HISTÓRICO DE INTERVENÇÕES

Todo o código-fonte gerado e utilizado, além do histórico de intervenções, script das métricas utilizadas e estudos de caso apresentados neste trabalho, está disponível no repositório:

<https://github.com/matheusfmb/hmtech>