

Avaliação de LLMs na Automação da Migração de Código

Eduardo Pazzini Zancanaro¹, Giancarlo D. Salton¹

¹Universidade Federal da Fronteira Sul
Campus Chapecó

eduardo.zancanaro@estudante.uffs.edu.br, gian@uffs.edu.br

Resumo. Este trabalho avalia LLMs, no formato de agentes, para apoiar a modernização de software legado por meio da geração automática de testes unitários. Como estudo de caso, utiliza-se a CICS Banking Sample Application (CBSA), originalmente em COBOL, tendo como alvo a versão modernizada em Go. Comparam-se Claude Code, OpenCode e Kilo Code em um fluxo padronizado, medindo custo (tempo, tokens e interações), abrangência (módulos cobertos e cobertura do código de produção) e qualidade/manutenibilidade via SonarQube. O Claude Code foi o mais autônomo e abrangente (6/6 módulos; 93,4% de cobertura média em uma interação). O OpenCode cobriu 3/6 módulos (84,0%) e exigiu intervenções; o Kilo Code gerou testes mais idiomáticos, porém apenas para 1/6 módulo. Conclui-se que a abordagem é viável, mas ainda requer revisão humana e depende do modelo, sobretudo em agentes gratuitos.

Abstract. This paper evaluates LLM-based agents for supporting legacy software modernization through automatic unit-test generation. The case study uses the CICS Banking Sample Application (CBSA), originally in COBOL, targeting its modernized Go version. We compare Claude Code, OpenCode, and Kilo Code under a standardized workflow, measuring cost (time, tokens, interactions), scope (covered modules and production-code coverage), and test quality/maintainability via SonarQube. Claude Code achieved the best autonomy and scope (6/6 modules; 93.4% average coverage in one interaction). OpenCode covered 3/6 modules (84.0%) and required interventions, while Kilo Code produced more idiomatic tests but only for 1/6 module. Results indicate feasibility, but end-to-end autonomy and semantic fidelity remain model-dependent and still require human review, especially with free/open agents.

1. Introdução

Atualmente, estima-se que existam mais de 220 bilhões de linhas de código escritas em COBOL, responsáveis por processar cerca de 3 trilhões de dólares diariamente em transações ao redor do mundo [Taulli 2020]. Apesar de sua relevância histórica, esses sistemas legados têm se tornado um gargalo para o avanço tecnológico, conforme destacado por [Florêncio et al. 2024].

Além dos riscos associados à migração desses códigos, outro desafio tem vindo à tona: a escassez de profissionais qualificados para trabalhar com código legado [Florêncio et al. 2024]. Com a dificuldade de atrair novos desenvolvedores para trabalhar

com linguagens consideradas obsoletas e o envelhecimento da força de trabalho especializada nesses sistemas, empresas e organizações enfrentam dificuldades crescentes para manter ou até mesmo substituir esses sistemas essenciais.

À medida que a demanda por mão de obra continua a crescer, torna-se necessária a busca por novas alternativas que permitam manter e evoluir esses sistemas, muitas vezes considerados obsoletos. Nesse contexto, este trabalho visa verificar o uso de Modelos de Linguagem de Grande Escala (do inglês, *Large Language Models* - LLMs) como ferramenta principal para processos de migração e modernização de código legado. O objetivo não é apenas realizar uma conversão sintática, como faria um *parser* tradicional, mas sim preservar integralmente as regras de negócio enquanto se realiza a modernização da arquitetura, a adoção de padrões de desenvolvimento e a reestruturação do código, tornando-o sustentável novamente.

Por outro lado, a Inteligência Artificial (IA) é uma das áreas da ciência da computação que vem crescendo de forma exponencial nos últimos anos, dedicada ao desenvolvimento de sistemas capazes de executar tarefas que, anteriormente, necessitavam de inteligência humana, desde o raciocínio e aprendizado até o reconhecimento de padrões para, então, realizar a melhor tomada de decisão [Russell and Norvig 2021].

Em contraponto ao avanço acelerado das tecnologias emergentes, os códigos legados permanecem amplamente presentes nas organizações, muitas vezes sem qualquer evolução estruturada. Em diversos cenários corporativos, esses sistemas antigos acabam sendo mantidos por inércia, marcados pelo receio de ajustes ou atualizações devido às consequências imprevisíveis que podem causar na operação. O autor deste trabalho vivenciou diretamente esse desafio ao participar da migração de um código legado com mais de 20 anos para uma plataforma atualizada, etapa necessária para possibilitar a modernização de outros sistemas dependentes.

Diante dessa experiência, surgiu o interesse em investigar a eficiência e a viabilidade de realizar migrações de código legado utilizando uma das tecnologias que mais tem avançado nos últimos anos, a Inteligência Artificial, com foco especial nos LLMs. Assim, o objetivo geral desta pesquisa é avaliar o potencial dos LLMs na automação da migração de código de sistemas legados, considerando a capacidade de interpretar um código existente, aplicar regras de modernização e produzir uma versão funcional e atualizada a partir de instruções textuais.

Este trabalho está organizado da seguinte forma. A Seção 2 apresenta os conceitos fundamentais de código legado e discute desafios e estratégias de migração. Em seguida, a Seção 3 introduz modelos de linguagem e os principais elementos necessários para compreender sua aplicação em tarefas de engenharia de software. A Seção 4 sintetiza os trabalhos relacionados, destacando abordagens recentes para modernização e migração com suporte de IA. Na Seção 5 é descrita a metodologia adotada nesta pesquisa, incluindo o delineamento, o ambiente de avaliação e os critérios de análise. A Seção 6 detalha o planejamento e a execução dos experimentos conduzidos. Posteriormente, a Seção 7 apresenta os resultados obtidos e a discussão à luz das métricas definidas. Por fim, a Seção 8 reúne as conclusões, limitações e indicações de trabalhos futuros.

2. Migração de Código Legado

Com o avanço da tecnologia e o surgimento de novas arquiteturas e linguagens de programação, muitas empresas se deparam com o desafio de manter os sistemas legados que, embora ainda funcionem, tornam-se cada vez mais difíceis de manter ou até mesmo de integrar novas funcionalidades. Nesta seção, serão abordados os principais conceitos de código legado, suas implicações para a manutenibilidade do sistema e, em seguida, um estudo sobre um dos principais exemplos de código legado: a linguagem COBOL.

2.1. Código Legado

O termo código legado refere-se a um conjunto de trechos ou bases de código que continuam em uso em sistemas de produção, mas que foram desenvolvidos em versões antigas, tanto de linguagens quanto de arquiteturas ou requisitos, e que, diversas vezes, não recebem manutenção regular ou atualizações adequadas. Por exemplo, segundo a IBM, código legado engloba “código que ainda serve ao seu propósito, mas foi escrito usando tecnologias ou ambientes desatualizados, com pouca ou nenhuma cobertura de teste automática e documentação deficiente” (tradução nossa) [IBM 2023]. Essa definição indica que legado, no contexto de engenharia de software, não significa necessariamente algo obsoleto ou inoperante, mas sim que o código carrega dependências que dificultam sua evolução.

As principais características que tendem a converter um código em legado são a utilização de tecnologias descontinuadas ou não mais suportadas, a falta de testes automatizados para garantir o funcionamento em próximas atualizações, a baixa modularização ou arquitetura monolítica e a escassez ou total ausência de documentações, mesmo que desatualizadas. Essas características aumentam o risco das manutenções necessárias para manter o sistema operacional. Conforme indicado por [SonarSource 2024], código legado opera sobre tecnologias ou plataformas desatualizadas, mas permanece crítico para as operações da organização, o que torna sua manutenção e evolução mais desafiadoras. Essa condição torna evidente que o legado é tanto um ativo quanto um obstáculo essencial para a continuidade operacional, mas funciona também como um limitante para a inovação tecnológica.

Neste trabalho, entende-se por modernização de código o processo de transformar um sistema existente para reduzir o custo de manutenção e permitir sua evolução, sem alterar o comportamento observado pelo negócio. As estratégias variam em grau de intervenção: desde a troca de infraestrutura sem alteração do código, passando pela reestruturação interna com preservação do comportamento externo, até a reescrita completa em outra linguagem ou arquitetura. Este trabalho situa-se no caso mais invasivo, pois a versão em Go do estudo de caso é uma reescrita do sistema COBOL original. Nesse cenário, os testes unitários são o instrumento que permite verificar se o comportamento do código reescrito corresponde ao do sistema legado.

2.2. COBOL

A linguagem COBOL (*Common Business Oriented Language*) foi desenvolvida no final da década de 1950, com o principal objetivo de atender a grande necessidade de processamento de dados em grandes corporações e aplicações administrativas. Projetada para ser próxima a uma linguagem natural, sua sintaxe legível e estruturada para a época favoreceu a adoção em grande escala por instituições financeiras, órgãos governamentais e grandes

corporações, consolidando-se como uma das linguagens mais utilizadas para sistemas críticos [Woodie 2022].

Desde a sua criação até os dias atuais, o COBOL permanece amplamente utilizado nos mesmos ambientes corporativos e governamentais, sendo estimado que bilhões de linhas de código ainda estão sendo utilizadas [Technology 2022]. Essa vida longa e duradoura se deve à sua estabilidade, eficiência e, principalmente, à sua confiabilidade, características essenciais em aplicações bancárias, de seguros e gestão pública. Além disso, muitas dessas aplicações foram criadas em plataformas mainframe, que continuam sendo a base operacional de diversas instituições de missões críticas.

Outro fator que contribui para a permanência do COBOL é a dificuldade e o alto custo associados à substituição desses sistemas [Mikac 2024]. Por sua integração a processos institucionais e por possuírem décadas de evolução contínua, a migração completa se tornou de custo operacional imensurável e apresenta grande risco para os sistemas existentes. Desta maneira, a linguagem COBOL se consolidou não apenas como uma linguagem histórica, mas como um pilar ainda ativo para manter a infraestrutura tecnológica de larga escala, demonstrando-se como um ponto de equilíbrio singular entre o legado e a indispensabilidade técnica [Technology 2022].

3. Modelos de Linguagem

Modelos de linguagem são sistemas desenvolvidos para estimar a probabilidade de uma sequência de *tokens*, unidades mínimas de texto utilizadas pelo modelo, que podem corresponder a palavras inteiras, subpalavras ou caracteres representados matematicamente, e prever qual será o próximo elemento com base no contexto fornecido. Inicialmente, esses modelos eram baseados em abordagens estatísticas, como no caso dos n-gramas, que limitavam seu contexto a poucas palavras anteriores para calcular probabilidades. Esta abordagem é incapaz de obter dependências distantes e sofre de esparsidade de dados, motivando assim a busca por modelos capazes de aprender representações contínuas e expressivas da linguagem natural.

Um dos primeiros avanços ocorreu com o surgimento das Redes Neurais Recorrentes (RNNs), que implementaram um mecanismo de estado interno, permitindo a propagação de informações ao longo da sequência. No entanto, as RNNs simples apresentavam dificuldades para aprender relações distantes devido a problemas como *vanishing gradients* e *exploding gradients*. O *vanishing* ocorre quando, ao retropropagar o erro por muitos passos temporais, os gradientes se tornam minúsculos, tornando impossível o aprendizado de dependências de longo prazo. Já o *exploding* surge quando esses valores crescem exponencialmente, causando instabilidade numérica e atualizações abruptas dos pesos.

Para diminuir essas limitações, surgiram variações como as *Long Short-Term Memory* (LSTM) e *Gated Recurrent Unit* (GRU), que incorporam mecanismos de portas para controlar o fluxo de informação. Esses modelos tornaram-se, por muitos anos, predominantes em tarefas de processamento de linguagem natural, sustentando aplicações como tradução, sumarização e modelagem de sequência a sequência. Mesmo assim, sua natureza impunha limitações de paralelização, o que dificultava o treinamento em larga escala.

O Transformer é um modelo de transdução de sequência proposto por [Vaswani et al. 2017], com o objetivo de lidar com sequências de texto sem precisar per-

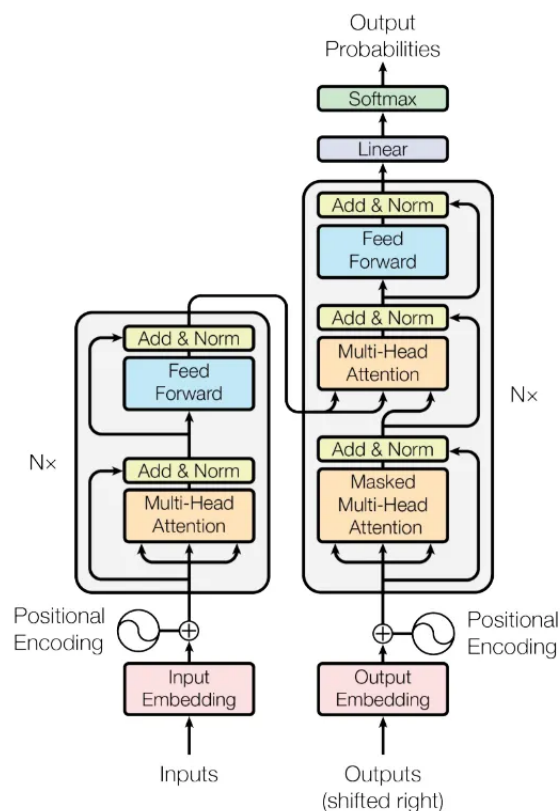


Figure 1. O Transformer — Arquitetura do modelo

Fonte: [Vaswani et al. 2017]

correr palavra por palavra para o entendimento. Diferentemente das arquiteturas anteriores, que se baseavam em redes recorrentes (RNNs) ou convolucionais (CNNs), o Transformer utiliza exclusivamente um mecanismo de auto-atenção para modelar dependências entre os elementos de uma sequência [Uszkoreit 2017]. Esse mecanismo aprimora o processamento dos *tokens*, de forma que todos sejam processados simultaneamente, capturando relações de longo alcance de maneira mais eficiente, quebrando a tarefa principal em tarefas menores que podem ser executadas de forma paralela [Vaswani et al. 2017].

A Figura 1 demonstra a arquitetura geral do Transformer. O modelo é composto por duas partes principais: o codificador (*encoder*) e o decodificador (*decoder*), ambos formados por múltiplas camadas empilhadas de submódulos de atenção e redes *feed-forward*. Essa estrutura é responsável por todo o fluxo de processamento da entrada em sequência e pela geração do conteúdo que será apresentado na sequência de saída, utilizando o mecanismo de auto-atenção para capturar as dependências contextuais entre os *tokens*.

O *encoder* processa a entrada em camadas empilhadas de auto-atenção e redes *feed-forward*. O *decoder* gera a saída de forma autorregressiva: cada *token* novo é produzido a partir das posições já geradas e da representação construída pelo *encoder* [Vaswani et al. 2017].

3.1. Large Language Models (LLMs)

Os LLMs são baseados na arquitetura Transformer [Vaswani et al. 2017] e seu princípio básico gira em torno da utilização do algoritmo de autoatenção para compreender e gerar textos em linguagem natural. Um LLM é treinado em enormes quantidades de dados, permitindo que o modelo aprenda padrões complexos, relações semânticas e estruturas sintáticas [Brown et al. 2020] presentes na língua.

No caso do código-fonte, esses padrões complexos correspondem a estruturas que multiplicam os caminhos de execução e as dependências implícitas de um programa, sendo condicionais aninhadas, desvios incondicionais e variáveis globais compartilhadas entre módulos. Programas COBOL antigos acumulam essas estruturas por serem procedurais e operarem sobre áreas de dados globais. Já linguagens modernas, como Go, favorecem funções curtas, com entrada e saída explícitas, que podem ser testadas de forma isolada.

O objetivo desses modelos é prever uma sequência de *tokens* a partir de probabilidades, de forma que possam prever o próximo *token* com alta precisão, formulado matematicamente como [Jurafsky and Martin 2023]:

$$P(t_1, t_2, \dots, t_n) = \prod_{i=1}^n P(t_i | t_1, t_2, \dots, t_{i-1}) \quad (1)$$

Com o avanço dos modelos e, conseqüentemente, o aumento nos conjuntos de dados, os LLMs passaram a apresentar capacidades que não eram observadas em modelos menores, como a melhora no raciocínio contextual, a compreensão de tarefas mais complexas e a geração de respostas coerentes em textos longos. Esse aprimoramento no desempenho se dá ao aumento do número de parâmetros treináveis, que podem chegar a trilhões em modelos modernos.

Outro aspecto essencial dos LLMs é a capacidade de operar como sistemas generalistas, onde basta fornecer instruções textuais adequadas para que o modelo execute funções diversas, característica conhecida como *in-context learning* [Brown et al. 2020]. Esse comportamento só se torna possível devido à forma como os Transformers processam *tokens* em paralelo e mantêm representações contextuais globais da entrada. Com isso, os LLMs tornaram-se fundamentais em diversas áreas, como assistentes virtuais, tradução, análise semântica e, o principal para este trabalho, a geração de código [Chen et al. 2021], além de diversas outras opções possíveis, consolidando-se como uma das maiores inovações tecnológicas derivadas do Transformer.

4. Trabalhos Relacionados

Em [Bandarupalli 2025] os autores investigam a modernização de sistemas legados, utilizando a inteligência artificial como pilar de apoio nesse processo, com foco na migração de códigos em COBOL para uma versão atualizada do Java. Para isso, os autores catalogaram cerca de 50 mil arquivos legados em COBOL, provenientes de códigos públicos e privados disponibilizados por empresas. Apesar de esse *corpus* não estar disponível, o artigo apresenta uma estratégia interessante para o levantamento dos casos de uso para a validação. Para medir a eficiência da modernização, foram utilizadas métricas de complexidade (ou ciclomática), atribuídas a [McCabe 1976], baseadas na quantidade

de 'caminhos' possíveis dentro de um módulo do código, e métricas de acoplamento, atribuídas a [Chidamber and Kemerer 1994], que indicam que, quanto mais dependências existem em um módulo, maior será o custo de manutenibilidade do sistema. Mesmo sem detalhar quais ferramentas foram usadas para determinar a precisão, os autores informam ter alcançado 93% de precisão na modernização, com redução de 35% na complexidade e queda de 33% no acoplamento, superando esforços manuais e ferramentas baseadas em regras.

No contexto de migração de dados, o trabalho [Santos and Kuszera 2024] propõe o desenvolvimento de uma ferramenta completamente automatizada para a migração entre bancos de dados relacionais, tendo como abordagem fundamental os LLMs e o uso de ferramentas de código aberto para integrar os modelos de inteligência artificial em uma *pipeline* de migração e transformação. A aplicação funciona como uma interface em que o desenvolvedor descreve ações e requisitos, e o sistema utiliza os LLMs para transformar essa requisição em um código executável que realiza a migração, buscando validar a eficácia das IAs na interpretação das requisições e na geração da lógica. Contudo, a meta de uma aplicação totalmente autônoma não foi atingida, pois ainda é necessário um especialista acompanhando todo o processo; assim, a abordagem se mostra mais viável como um auxiliar (ou copiloto) que acelera a produtividade, mas não substitui um especialista.

Já o artigo de revisão [Surendra 2025] discute como tecnologias de inteligência artificial têm sido aplicadas na manutenção e modernização de *software* legado, abordando problemas crônicos como a complexidade acumulada e a falta de documentação. A revisão se estrutura em três pilares: geração automatizada de documentação a partir da análise do código (regras de negócio e dependências), modernização por refatoração inteligente e reestruturação para padrões modernos, e geração automatizada de testes (unitários, integração, etc.) para reduzir riscos durante as modernizações e integrações de novas funcionalidades. Por se tratar de uma revisão, não há estudo de caso com métricas quantitativas: são descritos benefícios e impactos, mas sem resultados práticos (redução de custo, manutenções, cobertura de testes). Ainda assim, o texto pode ser entendido como um guia estratégico, enquanto este trabalho busca demonstrar, com dados e métricas, a capacidade das inteligências artificiais—especificamente agentes de IA—na migração de sistemas legados.

5. Metodologia

Esta pesquisa adota diferentes metodologias, iniciando com uma pesquisa aplicada, voltada à resolução de um problema real. Busca-se resolver um problema prático relacionado à criação de testes unitários para códigos legados utilizando LLMs. Para os experimentos, são utilizados códigos existentes e catalogados no GitHub [IBM CICS Development 2025].

Utilizou-se a abordagem metodológica quali-quantitativa. Do ponto de vista quantitativo, são analisadas métricas de custo operacional (tempo de execução, consumo de *tokens* e número de interações), além dos indicadores de análise estática do SonarQube, como complexidade, duplicação e débito técnico. Quanto ao aspecto qualitativo, corresponde à inspeção manual do código gerado, em duas frentes. A leitura dos testes quanto às convenções da linguagem e a comparação com a especificação COBOL original, para verificar se as regras de negócio foram preservadas.

A primeira etapa da metodologia consiste em um levantamento bibliográfico, no qual foram catalogados trabalhos relacionados à migração de sistemas legados, com ou sem a utilização de técnicas de Inteligência Artificial. A maioria desses estudos concentra-se na aplicação de algoritmos de IA padronizados. Em contraste, a proposta deste trabalho é explorar o uso de LLMs, com o objetivo de alcançar maior autonomia no processo de geração de testes. Para isso, foi utilizado como estudo de caso um código legado escrito em COBOL, tomando como base sua versão já modernizada em Go disponível no repositório, sobre a qual os agentes deveriam gerar testes unitários.

Posteriormente ao levantamento bibliográfico e tendo como base o trabalho de [Bandarupalli 2025], realizou-se uma pesquisa voltada à identificação e catalogação de uma base de código legado escrita em COBOL. O objetivo desta etapa foi encontrar repositórios que pudessem ser utilizados para a análise de migração do código legado para uma versão totalmente modernizada.

A busca foi realizada utilizando os termos "*cobol real project*" e/ou "*legacy cobol codebase*", procurando em repositórios públicos e bases digitais de código. Contudo, grande parte desses códigos apresenta um licenciamento que impede a utilização para fins acadêmicos ou condicionais, o que impediu a utilização para esta pesquisa.

Dentre os poucos casos em que a licença aplicada não refletia em bloqueios institucionais, a maioria referia-se a códigos fictícios ou muito simplórios que não representavam um projeto real. Essa limitação evidenciou um dos principais desafios metodológicos encontrados: a dificuldade de acesso a bases de códigos legados realistas e, ao mesmo tempo, viáveis para análise e modernização em trabalhos acadêmicos.

Diante desse cenário, foi selecionado o repositório [IBM CICS Development 2025], que contém um projeto de demonstração de um sistema bancário escrito em COBOL e projetado para fins de demonstração. Esse projeto foi escolhido por apresentar características mais próximas de um sistema legado real, além de oferecer código aberto, tornando-o adequado para servir como base para esta pesquisa. A suíte de testes já presente no repositório, embora escrita em COBOL, é composta por testes funcionais e de cenário, e não por testes unitários no sentido estrito, lacuna que define a tarefa atribuída aos agentes.

A catalogação dos LLMs foi realizada como segunda parte da pesquisa. Esses modelos são redes neurais de grande escala, com uma base imensa, que têm a capacidade de entender, criar e adaptar-se a diferentes contextos. Sua principal característica é a capacidade de generalizar tarefas sem precisar de treinamento específico, independentemente do caso, tornando-se versátil para usos que variam desde auxílio à escrita até programação.

Dentro desse segmento, a busca tornou-se refinada para modelos configurados no formato de agentes. Diferente dos modelos convencionais, que respondem de forma isolada a comandos específicos, os agentes têm a capacidade de responder de forma sequencial a tarefas de maneira autônoma, coordenando o raciocínio e interagindo com ferramentas externas. Optou-se por essa abordagem visando automatizar, tanto quanto possível, a geração de testes unitários para sistemas legados, apresentando apenas um documento informando o que o agente deve realizar e o código ao qual ele terá acesso.

Como resultado, foram selecionados os seguintes modelos para análise mais refinada: Claude Code, Kilo Code e OpenCode. Cada um deles se destaca por oferecer

abordagens distintas para execução autônoma, representando, assim, diferentes perspectivas metodológicas no uso de agentes para apoiar a geração autônoma de testes unitários. Cabe uma ressalva sobre o Kilo Code: em sua configuração padrão, ele não utiliza um modelo fixo, e sim um roteador que distribui as requisições entre LLMs gratuitos do OpenRouter. A comparação, portanto, ocorre entre agentes em sua configuração padrão, e não entre modelos específicos, o que limita a comparação direta e é retomado nas limitações.

A seleção desses três agentes foi baseada em critérios de relevância e complementaridade. O Claude Code é um agente proprietário e serve como parâmetro de comparação. O Kilo Code se diferencia por seu enfoque na análise e reescrita de código, alinhando-se mais ao contexto da tarefa. Por outro lado, o OpenCode proporciona um ambiente mais flexível e experimental, permitindo a adaptação a variados contextos. Assim, a seleção procurou equilibrar a solidez técnica, a aplicabilidade prática e a flexibilidade metodológica, assegurando a diversidade na abordagem do estudo.

Para a avaliação das métricas de qualidade dos testes gerados pelos LLMs, é utilizado o SonarQube, uma ferramenta para análise estática do código-fonte. O SonarQube permite estipular os atributos relacionados à qualidade estrutural e à manutenibilidade do *software*, à identificação de possíveis falhas e/ou vulnerabilidades e à violação das boas práticas de desenvolvimento. Sua utilização tem como objetivo oferecer uma base padronizada e comparável entre os resultados produzidos por cada modelo.

A ferramenta SonarQube baseia-se na inspeção estática do código, sem a necessidade de compilá-lo ou executá-lo. Ela utiliza regras predefinidas que avaliam aspectos de confiabilidade, segurança, manutenibilidade e cobertura de testes. Cada análise realizada gera indicadores numéricos e qualitativos, permitindo a classificação do código em níveis de qualidade. Entre as principais métricas, destaca-se a Complexidade Ciclomática (CC), que mede a quantidade de caminhos lógicos independentes existentes no código.

$$CC = E - N + 2P \quad (2)$$

onde:

- E é o número de arestas do grafo de fluxo de controle;
- N é o número de nós;
- P é o número de componentes conexos.

Além disso, a Duplicação de Código (DUP) serve para indicar o percentual de linhas de código duplicadas no projeto.

$$DUP(\%) = \frac{L_{duplicadas}}{L_{total}} \times 100 \quad (3)$$

onde:

- $L_{duplicadas}$ é o número de linhas idênticas detectadas em blocos diferentes;
- L_{total} é o total de linhas analisadas.

A Cobertura de Testes (CT) avalia a proporção de código coberta por testes automatizados.

$$CT(\%) = \frac{L_{testadas}}{L_{total}} \times 100 \quad (4)$$

Um code smell é um trecho de código que funciona corretamente, mas está escrito de forma que dificulta a leitura ou a manutenção futura. A Manutenibilidade é avaliada pelo modelo *SQALE*, adotado pelo SonarQube. A ferramenta estima o tempo de correção de cada *code smell*, e a soma desses tempos forma o débito técnico. Para definir o *rating*, que varia de A a E, esse débito é comparado ao custo estimado de reescrever todo o código analisado, caso ele represente até 5% desse custo, o projeto recebe nota A. A medida é relativa ao tamanho do código, e por isso projetos com débitos distintos podem receber a mesma nota.

Por fim, com os dados obtidos por meio das métricas coletadas durante os experimentos, é realizada uma análise comparativa entre os resultados produzidos pelos diferentes LLMs, com o intuito de avaliar a eficácia do uso dessa tecnologia de forma autônoma na geração de testes unitários para sistemas legados. Essa etapa permite avaliar não apenas o desempenho técnico de cada modelo, mas também suas limitações práticas no contexto da geração autônoma de testes. A partir desses dados, é possível quantificar o desempenho de cada modelo, identificando os cenários em que os LLMs apresentam maior eficiência e os pontos em que ainda ocorrem falhas, como cobertura parcial dos módulos, testes redundantes ou divergências entre a lógica COBOL original e a versão reescrita em Golang. Essa análise técnica permite avaliar o nível de maturidade das abordagens utilizadas e fornecer subsídios concretos para a definição de estratégias futuras de aprimoramento dos modelos voltados à geração autônoma de testes para código legado.

6. Experimentos

Os três agentes foram avaliados sobre o repositório CICS Banking Sample Application (CBSA) [IBM CICS Development 2025]¹, com escopo restrito à geração dos testes para a versão em Go derivada do código COBOL original.

Para garantir a comparabilidade entre os modelos, montou-se um fluxo padronizado de execução. O repositório foi copiado para três pastas idênticas, uma para cada agente, e em cada cópia inseriu-se um arquivo com as mesmas instruções de geração de testes, de modo que todos partissem da mesma tarefa inicial. Essa padronização é importante porque pequenas variações no estado inicial do código podem levar agentes baseados em LLMs a caminhos de interpretação muito distintos. A Figura 2 resume o fluxo de execução do experimento.

A partir desse ponto, cada agente trabalhou de forma independente em sua cópia do projeto, interpretando o código e gerando os testes unitários e os artefatos auxiliares necessários. Todo o processo foi acompanhado por ferramentas de registro, que capturaram as métricas descritas a seguir e permitiram tanto a análise isolada de cada agente quanto a comparação direta entre os três.

A avaliação combinou três grupos de métricas, escolhidos para capturar dimensões complementares do desempenho. O primeiro reúne o custo de operação de cada agente,

¹O material completo do experimento, incluindo a configuração utilizada e os códigos gerados por cada agente, está disponível em: <https://github.com/oduardu/COBOL-Legacy-Benchmark-Suite>.

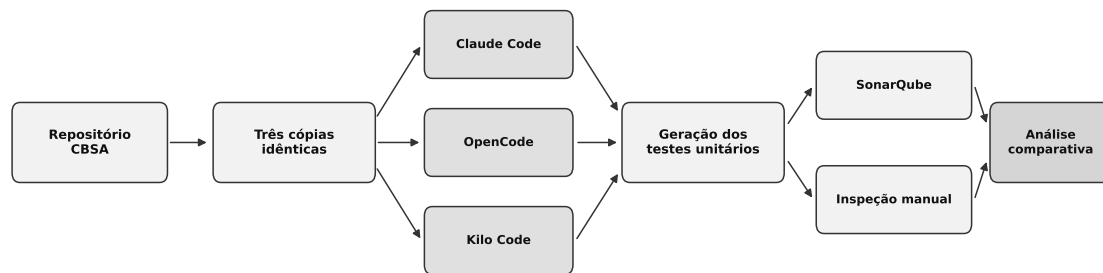


Figure 2. Fluxo de execução do experimento

Fonte: elaborado pelo autor.

registrado durante a execução, seja ele o tempo até a conclusão, o consumo de *tokens* e o número de interações necessárias. O segundo descreve a qualidade do código entregue, com complexidade ciclomática e cognitiva, duplicação, facilidade de manutenção e os chamados *code smells*. Essas métricas foram obtidas com o SonarQube, executado localmente sobre cada uma das três cópias do projeto, com o perfil de regras padrão para Go (*Sonar way*), e a análise abrangeu os arquivos de teste gerados e o código de produção. O terceiro mede o quanto do código é efetivamente exercitado pelos testes: a cobertura foi calculada com a própria ferramenta do Go e o relatório resultante foi importado no SonarQube, que a consolidou junto às demais métricas.

A métrica de cobertura prevista na metodologia foi ajustada quanto ao escopo. Medir a cobertura dos próprios testes é pouco informativo quando o objeto de análise são os testes em si, já que arquivos de teste raramente contêm caminhos alternativos e o resultado ficaria artificialmente próximo do total. Por isso, mediu-se quanto do código de produção é exercitado pelos testes gerados, abordagem mais coerente com o que se quer avaliar: a capacidade efetiva dos testes de proteger o sistema contra falhas introduzidas em mudanças futuras.

O código a seguir mostra a validação de tipo de investimento nas quatro versões. Os três agentes chegaram à mesma lógica do COBOL original, o que difere é apenas como cada um organizou o código em Go.

```

1  -- COBOL
2  3000-VALIDATE-TYPE.
3      IF LS-INPUT-VALUE NOT = 'STK'
4          AND NOT = 'BND'
5          AND NOT = 'MMF'
6          AND NOT = 'ETF'
7          MOVE VAL-INVALID-TYPE TO LS-RETURN-CODE
8          MOVE VAL-ERR-TYPE      TO LS-ERROR-MSG
9          EXIT PARAGRAPH
10     END-IF
11     MOVE VAL-SUCCESS TO LS-RETURN-CODE
12     MOVE SPACES      TO LS-ERROR-MSG
13
14  -- Claude Code
15  func ValidateInvestmentType(input string) ValidationResult {
16      switch input {
17      case "STK", "BND", "MMF", "ETF":
18          return ValidationResult{Success, ""}
19      default:
20          return ValidationResult{ErrInvalidType, ErrMsgInvalidType}
21      }
22  }
23
24  -- Kilo Code
25  var validTypes = map[string]bool{
26      "STK": true, "BND": true, "MMF": true, "ETF": true,
27  }
28  func ValidateType(input string) (int16, string) {
29      if !validTypes[input] {
30          return ValInvalidType, "Invalid Investment Type"
31      }
32      return ValSuccess, ""
33  }
34
35  -- OpenCode
36  func ValidateInvestmentType(input string) (int16, string) {
37      validTypes := map[string]bool{
38          "STK": true, "BND": true, "MMF": true, "ETF": true,
39      }
40      if !validTypes[input] {
41          return ValInvalidType, "Invalid Investment Type"
42      }
43      return ValSuccess, ""
44  }

```

Listing 1. Validação de tipo de investimento nas quatro versões

7. Resultado e Discussão

Os resultados são apresentados primeiro em uma análise comparativa, consolidando as métricas obtidas e, em seguida, detalhados para cada agente em separado. Fecham a seção as limitações que delimitam o alcance dos achados.

7.1. Análise comparativa

A Tabela 1 consolida as métricas obtidas, agrupadas nas três dimensões definidas no início desta seção: custo operacional, abrangência e qualidade estrutural. A diferença entre o modelo pago (Claude Code) e os modelos gratuitos (OpenCode e Kilo Code) manifesta-se, sobretudo, em autonomia de execução, abrangência da cobertura e número de interações exigidas. O Claude Code foi o único agente a validar autonomamente os testes gerados e o único a cobrir todos os módulos do domínio em uma única interação. Os modelos gratuitos exigiram intervenções adicionais: para acionar a execução, no caso do OpenCode, e para corrigir erros sintáticos e de execução, no caso do Kilo Code.

Table 1. Comparativo entre os três modelos avaliados.

Métrica	Claude Code	OpenCode	Kilo Code
<i>Custo operacional</i>			
Tempo de execução	~8 min	~7 min	~12 min
Tokens consumidos	52 mil	47 mil	31 mil
Interações	1	2	3
Validação autônoma	Sim	Não	Não
<i>Abrangência</i>			
Módulos cobertos	6/6	3/6	1/6
Funções de teste	76	45	9
Cobertura média	93,4%	84,0%	100,0%*
<i>Qualidade estrutural</i>			
Testes orientados a tabela	21	2	9
Subtestes	32	2	9
Manutenibilidade	A	A	A
Débito técnico	34 min	50 min	12 min
Code smells	11	5	2

* Cobertura calculada apenas sobre o único módulo contemplado pelo Kilo Code.

Fonte: elaborado pelo autor.

Quando a análise se desloca da abrangência para a profundidade, os resultados se reordenam. O Kilo Code, embora tenha coberto apenas um módulo, gerou para esse módulo os testes mais alinhados às convenções da linguagem, com baixa duplicação e zero *code smells* nos arquivos de teste, o melhor índice de manutenibilidade isolado do experimento. O desempenho do modelo padrão do Kilo Code, que opera como roteador entre LLMs gratuitos do OpenRouter, é sensível ao escopo: produz qualidade aceitável em tarefas circunscritas, mas falha em manter coerência ao tentar abranger um projeto inteiro. Em projetos extensos, fracionar a tarefa em escopos menores pode ser uma estratégia viável para extrair desempenho razoável de agentes baseados em modelos gratuitos.

O OpenCode ocupa uma posição intermediária em todas as métricas, com a particularidade de ter gerado o maior número de testes em valor absoluto (45), sem alcançar a abrangência do Claude Code nem o cuidado estrutural do Kilo Code. A duplicação de literais, com um único texto aparecendo 13 vezes em um arquivo, mostra o impacto da ausência de testes orientados a tabela sobre a manutenibilidade: o número bruto de funções de teste diz pouco sobre a qualidade do conjunto.

O custo total da operação, para este tipo de tarefa, não é uma função linear do preço por *token*. O Kilo Code apresentou o menor consumo, mas exigiu três interações e entregou a menor abrangência; o Claude Code apresentou o maior consumo, mas concluiu em uma única interação. O tempo do desenvolvedor envolvido nas interações intermediárias precisa ser considerado nessa conta.

[Santos and Kuszera 2024] concluiu, em contexto de migração de bancos de dados, que LLMs apresentam maior viabilidade como copilotos do que como agentes plenamente autônomos. O desempenho do OpenCode e do Kilo Code, ambos requerendo intervenção humana para concluir a tarefa, corrobora essa conclusão. O Claude Code, em contrapartida, aproxima-se do patamar reportado por [Bandarupalli 2025] na modernização de COBOL para Java (93% de precisão), o que sugere que a fronteira entre copiloto e agente autônomo, no estado atual da tecnologia, depende mais do modelo subjacente do que do paradigma de uso adotado.

Os desvios encontrados na inspeção manual do OpenCode delimitam o que a cobertura média de fato mede. A cobertura indica quanto do código gerado é exercitado pelos próprios testes gerados, mas nada diz sobre a correspondência entre esse código e a especificação original. Testes e código foram produzidos pelo mesmo agente, a partir da mesma interpretação. Um erro de leitura, como o da notação de *substring* (detalhado na Seção 7.3), gera ao mesmo tempo o código errado e o teste que o confirma. Por isso, a preservação das regras de negócio não pode ser inferida da cobertura e precisa ser verificada por revisão humana. O raciocínio se estende às etapas seguintes da modernização. Se a suíte tiver asserções fracas, caso da checagem binária do Kilo Code, o código validado contra ela herda essa tolerância. Na prática, a qualidade dos testes limita a qualidade de todo código que venha a ser construído sobre eles.

7.2. Claude Code

O Claude Code concluiu o processo em 8 minutos e 14 segundos, com consumo de 52 mil *tokens* em uma única interação. O agente gerou os arquivos de teste, junto com o código de apoio, ou seja, as funções auxiliares e as estruturas que representam os registros COBOL. Em seguida, executou o conjunto resultante e validou a aderência ao comportamento do programa original, sem intervenção. Esse modo de operação reduz o esforço do desenvolvedor ao custo de um consumo maior de *tokens*.

O conjunto produzido contém 6 arquivos de teste (1.329 linhas, sem contar comentários) que cobrem todos os 6 módulos do domínio, com 76 funções de teste e 32 *subtestes* nomeados. Predomina a abordagem de testes orientados a tabela, com 21 ocorrências: nela, os casos de teste são descritos em uma tabela de entradas e saídas esperadas, de modo que acrescentar um caso novo exige alterar apenas a tabela, sem replicar a estrutura de execução. É o formato recomendado pela comunidade Go. As verificações comparam tanto o código de retorno quanto a mensagem de erro produzida, o que torna os testes mais sensíveis a mudanças de comportamento do que uma checagem simples de erro deixaria passar.

A cobertura média sobre os módulos de produção foi de 93,4%. Três módulos atingiram cobertura total ou quase total: *portmstr* (100%), *portvald* (100%) e *porttran* (97,7%). A análise do SonarQube classificou a manutenibilidade como nível A, com débito técnico de 34 minutos e 11 *code smells*. Destes, 5 correspondem à duplicação

de literais de texto em arquivos de teste e 6 a marcadores de pendência não resolvidos. A presença dos marcadores é informativa: o agente sinalizou pontos do código que mereciam atenção posterior em vez de encobri-los, o que facilita a revisão humana.

Foi sinalizada uma vulnerabilidade bloqueante pela regra de detecção de credenciais, identificada posteriormente como falso positivo: o *token* de análise do SonarQube havia sido detectado no arquivo de configuração criado para o experimento, e não no código gerado pelo agente. O achado reforça a importância da inspeção manual mesmo quando o relatório automatizado aponta uma vulnerabilidade severa.

7.3. OpenCode

O OpenCode levou aproximadamente 7 minutos e consumiu 47 mil *tokens*, distribuídos em duas interações. A primeira gerou os arquivos de teste, junto com o código de apoio, mas sem executar os testes, o agente apenas indicou os passos para que o usuário o fizesse. Foi necessário um segundo *prompt* solicitando a correção de um erro identificado em tempo de execução, após o qual o conjunto passou integralmente. Essa necessidade de intervenção aproxima o agente do modo de operação de um copiloto, que trabalha em ciclos curtos e devolve o controle ao usuário a cada etapa.

O resultado final contém 3 arquivos de teste (472 linhas, sem contar comentários) que cobrem 3 dos 6 módulos do domínio, com 45 funções de teste. O uso de testes orientados a tabela é esporádico, com apenas 2 ocorrências e 2 *subtestes*. Prevaleceu a estratégia de uma função de teste para cada caso de entrada, abordagem verbosa e distante das convenções da linguagem, que encarece a manutenção: adicionar um novo caso exige replicar a estrutura completa da função, e cenários relacionados ficam espalhados em vez de agrupados.

A cobertura média foi de 84,0% sobre os módulos contemplados. A manutenibilidade foi classificada como nível A, com 50 minutos de débito técnico e 5 *code smells*, todos relativos à duplicação de literais de texto. Destaca-se o texto "GROWTH PORTFOLIO", repetido 13 vezes em um único arquivo de teste, consequência direta da ausência de testes orientados a tabela: qualquer mudança futura nesse valor precisa ser repetida em cada ocorrência. Também foi apontada a complexidade cognitiva do método `portvald`, que ultrapassou em 1 ponto o limite estabelecido para essa métrica. A violação por margem mínima sugere que o agente operou próximo do limite aceitável, sem otimizar a clareza além do necessário.

A inspeção manual do código revelou três padrões de divergência em relação à especificação COBOL que os testes gerados não detectaram.

O primeiro é um erro de leitura da notação de *substring* do COBOL. A linguagem define *substring* por posição inicial e comprimento, e o OpenCode interpretou o comprimento incorretamente, lendo um caractere a mais do que o esperado. O mesmo erro aparece em dois módulos distintos, o que indica uma interpretação sistemática incorreta, não um caso isolado. Na prática, identificadores mínimos válidos pela especificação passam a ser rejeitados.

O segundo envolve o tipo usado para campos de texto de um único caractere. No COBOL, esse tipo é sempre tratado como string, independente do tamanho. O OpenCode optou por um tipo diferente, comum em Go para esse caso, mas incompatível com a

especificação. O código funciona internamente, mas qualquer sistema externo que passe o valor no formato esperado pelo COBOL encontra um problema de compatibilidade.

O terceiro é a adição de uma verificação condicional onde a especificação não prevê nenhuma. No COBOL original, a checagem de existência de portfólio é sempre executada, sem exceção. O OpenCode introduziu uma condição que permite pular essa validação em determinadas circunstâncias, criando um caminho em que transações com portfólios inexistentes passariam como válidas.

Os três casos têm origem semelhante, o agente fez escolhas naturais para Go que não correspondiam ao comportamento do COBOL original. O código compila e passa nos próprios testes, mas diverge da especificação nos pontos que esses testes não alcançam.

7.4. Kilo Code

O Kilo Code levou aproximadamente 12 minutos e consumiu 31 mil *tokens*, em três interações: a geração inicial dos testes, uma correção de erros de sintaxe que impediram a compilação na primeira tentativa e um terceiro ajuste para resolver falhas de execução identificadas após a correção sintática. Foi o agente que mais dependeu de intervenção humana, ainda que o consumo de *tokens* tenha sido o menor do experimento.

O resultado final contém 1 arquivo de teste (200 linhas, sem contar comentários) que cobre apenas 1 dos 6 módulos do domínio (`portvald`), com 9 funções de teste e 9 *subtestes*, todos estruturados como orientados a tabela. Embora a cobertura sobre o módulo testado tenha atingido 100%, a abrangência é restrita: cinco módulos do projeto não receberam qualquer teste. O número isolado de cobertura é enganoso nesse caso, pois métricas locais podem mascarar lacunas globais.

As verificações utilizam estratégia binária, que apenas checa se houve erro, e não o código específico retornado. Na prática, uma mudança no código de produção que troque um erro de validação por outro erro qualquer passaria despercebida. A manutenibilidade foi classificada como nível A, com o menor débito técnico do experimento (12 minutos) e apenas 2 *code smells*, ambos no código de produção, nenhum nos arquivos de teste. O agente entrega qualidade alta em um escopo reduzido, característica retomada na análise comparativa.

7.5. Limitações

Três limitações devem ser consideradas na interpretação dos resultados. Primeiro, cada agente foi executado uma única vez, sem repetições estatísticas. LLMs são modelos estocásticos, ou seja, execuções com o mesmo *prompt* e o mesmo código de entrada podem produzir resultados diferentes. Os números apresentados são, portanto, uma amostra do comportamento de cada agente, não seu desempenho esperado. O caso do Kilo Code agrava essa incerteza, pois seu roteador interno alterna entre modelos gratuitos do OpenRouter e execuções subsequentes podem diferir até na arquitetura do código gerado.

Segundo, a cobertura de caminhos de decisão não foi avaliada, uma vez que a versão gratuita do SonarQube não consome esse formato de relatório quando a fonte é Go. A cobertura medida neste trabalho restringe-se à cobertura de linhas, métrica que pode superestimar a abrangência efetiva do conjunto de testes quando há condicionais cujos caminhos alternativos não são exercitados pelos testes. Além disso, a cobertura por si só não

avalia a qualidade das asserções: um teste pode executar uma linha sem necessariamente detectar regressões, sobretudo quando as verificações são pouco específicas.

Terceiro, os achados são condicionados ao contexto do experimento: um único código-base, um único conjunto de *prompts* e uma única configuração de ferramentas e ambiente. Diferenças de domínio, tamanho do projeto, estilo do código, convenções internas e versões dos agentes podem alterar substancialmente tanto a abrangência quanto a qualidade estrutural do conjunto gerado.

8. Conclusão

O uso de LLMs para gerar testes unitários a partir de sistemas legados é tecnicamente viável, ao menos no recorte avaliado neste trabalho: a geração de testes para a versão em Go derivada de um código COBOL. Os resultados indicam uma vantagem clara dos modelos pagos em autonomia e abrangência. Já os modelos gratuitos podem produzir testes de boa qualidade estrutural em escopos menores, mas tendem a exigir maior intervenção humana e/ou fracionamento da tarefa para manter a coerência.

No caso específico das LLMs livres (gratuitas e/ou *open-source*), o principal ganho potencial é a redução do custo direto e a flexibilidade de uso (por exemplo, alternar modelos, ajustar parâmetros e, quando aplicável, executar localmente). Em contrapartida, o experimento sugere que essa economia vem acompanhada de um maior custo operacional indireto, na forma de iterações adicionais, revisão humana e necessidade de restringir o escopo para manter a consistência. Assim, seu uso se mostra mais promissor como apoio incremental ao desenvolvimento — por exemplo, gerando testes por módulo, validando cenários específicos e refatorando gradualmente os testes para formatos mais idiomáticos — do que como um agente que conclui o ciclo fim a fim em uma única execução.

Os resultados do experimento dão concretude a essa conclusão. O Claude Code foi o único agente a fechar o ciclo de geração, execução e validação sem intervenção, cobrindo os seis módulos do domínio com cobertura média de 93,4%, um patamar próximo ao reportado por [Bandarupalli 2025] na modernização de COBOL para Java. O OpenCode e o Kilo Code dependeram de intervenção humana e não atingiram a mesma abrangência global, o que corrobora a observação de [Santos and Kuszera 2024] de que LLMs gratuitos operam melhor como copilotos do que como agentes plenos. A inspeção manual do OpenCode revelou divergências que a cobertura de testes não capturou: o código compila, passa nos próprios testes, mas se comporta de forma diferente do sistema legado. Nesse modelo de uso, a revisão humana precisa ir além da execução dos testes e alcançar a correspondência do código gerado com as regras de negócio originais.

Para trabalhos futuros, cabem três frentes. A primeira é repetir as execuções e variar os *prompts* para estimar a variabilidade inerente aos modelos. A segunda é acompanhar a evolução dos modelos *open-source*, observando se avançam na execução autônoma da tarefa em vez de operarem como copilotos. A terceira é ampliar o escopo da pesquisa, avaliando não só a geração de testes como apoio ao processo de modernização, mas também a modernização completa do código, incluindo projetos COBOL de maior porte. Nesse cenário mais abrangente, incluir métricas de fidelidade semântica à especificação original, além de cobertura e qualidade estrutural, permitiria verificar se estratégias de fracionamento e revisão humana são suficientes para aproximar o custo total dos agentes gratuitos ao dos pagos. Uma quarta frente é inverter a ordem do processo, ou seja,

gerar primeiro, com revisão humana, uma suíte de testes de alta qualidade a partir da especificação COBOL e usá-la como oráculo para a geração do código modernizado por outros agentes. Essa configuração atacaria diretamente o problema de fidelidade semântica observado no OpenCode.

References

- Bandarupalli, G. (2025). Code reborn ai-driven legacy systems modernization from cobol to java.
- Brown, T. B. et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*.
- Chen, M. et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493.
- Florêncio, J., Bentes, C., and Castro, M. (2024). Processo de migração de aplicação mainframe cobol para microsserviços java na nuvem. In *Anais da IX Escola Regional de Alto Desempenho do Rio de Janeiro*, pages 6–10, Porto Alegre, RS, Brasil. SBC.
- IBM (2023). What is legacy code? Acesso em: 05 out. 2025.
- IBM CICS Development (2025). Cics banking sample application (cbsa). <https://github.com/cicsdev/cics-banking-sample-application-cbsa>. Acesso em: 27 set. 2025.
- Jurafsky, D. and Martin, J. H. (2023). *Speech and Language Processing*. Pearson, 3 edition. Draft disponível online.
- McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.
- Mikac, E. (2024). What is cobol and who still uses it? Acessado em: 01 dez. 2025.
- Russell, S. and Norvig, P. (2021). *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition.
- Santos, J. V. and Kuszera, E. (2024). Abordagem de desenvolvimento de uma pipeline de migração de dados utilizando inteligência artificial. In *Anais do XXI Congresso Latino-Americano de Software Livre e Tecnologias Abertas*, pages 41–48, Porto Alegre, RS, Brasil. SBC.
- SonarSource (2024). What is legacy code and why it matters. Acesso em: 05 out. 2025.
- Surendra, P. K. M. (2025). Automating documentation and legacy code modernization: Revitalizing legacy systems with ai. *World Journal of Advanced Engineering Technology and Sciences*, 15(01).
- Taulli, T. (2020). Cobol language: Call it a comeback? Acesso em: 27 nov. 2025.
- Technology, T. S. (2022). Cobol code base in production hits 800+ billion lines. Acessado em: 20 nov. 2025.
- Uszkoreit, J. (2017). Transformer: A novel neural network architecture for language understanding. Google Research Blog. Acesso em: 01 dez. 2025.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Woodie, A. (2022). Global cobol growth spurs reevaluation of ‘legacy’ assets. *IT Jungle*.