Universidade Católica do Salvador
Bacharelado em Engenharia de *Software*

Bernardo Braga e Cicero
Eduardo Lima Firmino
Fabrício Maicon Felix Santos
Gabriel Vinicius Pereira Andrade
Mateus Martins de Souza Oliveira

Mutation AI Studio: Uma Ferramenta Inteligente para Geração e Evolução Automática de Testes Unitários utilizando *Mutation Testing*

Salvador
2026

Bernardo Braga e Cicero
Eduardo Lima Firmino
Fabrício Maicon Felix Santos
Gabriel Vinicius Pereira Andrade
Mateus Martins de Souza Oliveira

Mutation AI Studio: Uma Ferramenta Inteligente para Geração e Evolução Automática de Testes Unitários utilizando *Mutation Testing*

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como parte dos requisitos necessários para a obtenção do Título de Bacharel em Engenharia de *Software*.
Orientador: Prof. Dr. Flávio Dusse.

Salvador
2026

Bernardo Braga e Cicero
Eduardo Lima Firmino
Fabrício Maicon Felix Santos
Gabriel Vinicius Pereira Andrade
Mateus Martins de Souza Oliveira

Mutation AI Studio: Uma Ferramenta Inteligente para Geração e Evolução Automática de Testes Unitários utilizando *Mutation Testing*

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como requisito parcial para a obtenção do título de Bacharel em Engenharia de *Software*.

Salvador, 1 de julho de 2026
Banca Examinadora:

Prof. Dr. Flávio Dusse
Universidade Católica do Salvador
Orientador

Prof. Me. Segundo professor
Universidade Católica do Salvador

Prof. Me. Terceiro professor
Universidade Católica do Salvador

Dedicamos este trabalho aos familiares, amigos e professores que contribuíram para nossa formação acadêmica e profissional e que incentivaram a busca por soluções tecnológicas com utilidade prática.

Agradecimentos

Agradecemos primeiramente a Deus, às nossas famílias, ao orientador Prof. Dr. Flávio Dusse e a todos os colegas que contribuíram com apoio, incentivo e troca de conhecimento ao longo do desenvolvimento deste trabalho. Agradecemos também à comunidade de desenvolvimento de *software*, cuja produção técnica e científica permitiu o amadurecimento desse projeto, em especial no campo de testes automatizados, qualidade de *software*, ferramentas de *mutation testing* e ferramentas baseadas em IA local. O *Mutation AI Studio* nasce da combinação entre pesquisa acadêmica e prática de Engenharia, e por isso registra a importância de todos que incentivaram a construção de um projeto aplicável, realista e alinhado às necessidades do mercado.

"O teste de programas pode ser usado para mostrar
a presença de falhas,
mas nunca para mostrar a sua ausência!"

– Edsger W. Dijkstra

Resumo

Este trabalho apresenta o *Mutation AI Studio*, uma ferramenta baseada em IA para geração e evolução automática de testes unitários utilizando *mutation testing*, concebida como um Produto Mínimo Viável (MVP) experimental de escopo controlado. O sistema integra geração e melhoria automática de testes unitários em projetos Java Maven utilizando um modelo de linguagem local e *mutation testing* com PITest. O fluxo executado contempla a análise de um projeto Maven real, a geração de testes JUnit 5 via modelo local com Ollama, a execução do *mutation testing*, a identificação de mutantes sobreviventes e a realização de um ciclo automático de reescrita dos testes com o objetivo de elevar a pontuação final da suíte. Ao final, os resultados são exibidos em um painel visual com comparação antes e depois. O trabalho também detalha a fundamentação teórica do uso de *mutation testing* como métrica de efetividade de testes, discute as limitações de métricas tradicionais baseadas apenas em cobertura e apresenta a arquitetura viável para execução na máquina do próprio desenvolvedor. Como contribuição, a proposta aproxima conceitos acadêmicos de teste de *software* de uma implementação prática voltada a projetos Java reais e mostra como a IA local pode apoiar a evolução dos testes sem depender de serviços externos.

Palavras chave: testes unitários; *mutation testing*; PITest; Inteligência Artificial; Ollama; JUnit 5; Java Maven.

Abstract

This work presents *Mutation AI Studio*, an intelligent tool for automatic generation and evolution of unit tests using mutation testing, designed as an experimental Minimum Viable Product (MVP) with controlled scope. The system integrates automatic generation and improvement of unit tests for Java Maven projects using a local Artificial Intelligence model and mutation testing with PITest. Its pipeline includes analyzing a real Maven project, generating JUnit 5 tests through a local model with Ollama, running mutation testing, identifying surviving mutants, and performing a mandatory automatic rewrite cycle to increase the test suite score. At the end, the results are displayed on a web dashboard with before and after comparison. The work also discusses the theoretical basis of mutation testing as a stronger quality indicator than plain code coverage and describes a feasible architecture for local execution. As a contribution, the proposal connects *software* testing theory with a practical tool capable of supporting developers in the analysis and gradual improvement of their test suites without relying on cloud services.

Keywords: unit testing; mutation testing; PITest; Artificial Intelligence; Ollama; JUnit 5; Java Maven.

Lista de Figuras

Lista de Quadros

Lista de Siglas e Abreviaturas

AAA	*Arrange, Act, Assert*
AI	*Artificial Intelligence*
API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
CI/CD	*Continuous Integration / Continuous Deployment*
CLI	*Command Line Interface*
CRUD	*Create, Read, Update, Delete*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IA	Inteligência Artificial
JSON	*JavaScript Object Notation*
JUnit	*Framework* de testes unitários para Java
LLM	*Large Language Model* (Grandes Modelos de Linguagem)
MVP	*Minimum Viable Product* (Produto Mínimo Viável)
OE	Objetivo Específico
PITest	*PIT Mutation Testing*
POM	*Project Object Model*
REST	*Representational State Transfer*
TCC	Trabalho de Concolusão de Curso
TDD	*Test Driven Development*
WS	*WebSocket*
XML	*eXtensible Markup Language*

Sumário

1 Introdução

A qualidade de *software* depende não apenas da presença de testes automatizados, mas principalmente da sua capacidade real de identificar comportamentos incorretos antes que eles atinjam o ambiente de produção (MYERS; SANDLER; BADGETT, 2011). Em equipes de desenvolvimento, é comum utilizar métricas como quantidade de testes, cobertura de linhas e cobertura de ramos para avaliar a maturidade da suíte. Embora úteis como indicadores operacionais, esses números não capturam sozinhos a força das asserções e a capacidade de o teste distinguir uma implementação correta de uma defeituosa (AMMANN; OFFUTT, 2017). Um teste pode executar um método inteiro, elevando a cobertura, e ainda assim não verificar de modo suficiente o comportamento esperado.

Diante desse descompasso, o *mutation testing* ganha relevância por avaliar a efetividade dos testes a partir da criação de pequenas alterações artificiais no código fonte, chamadas de mutantes (JIA; HARMAN, 2011). A ideia central é simples: se os testes são realmente robustos, eles devem falhar quando o comportamento do programa é corrompido por uma alteração semanticamente relevante. Quando um mutante sobrevive, um alerta é aceso, indicando que a suíte pode estar superficial, incompleta ou pouco específica (PAPADAKIS et al., 2019). Por isso, essa técnica se mostra crucial em projetos nos quais mudanças frequentes exigem um alto nível de confiança (DELAMARO; MALDONADO; JINO, 2016).

Mesmo com seu evidente valor técnico, a adoção dessa prática no dia a dia do desenvolvimento esbarra em obstáculos estruturais. Os fatores mais críticos incluem o alto custo computacional, o volume de informações geradas e o esforço braçal necessário para transformar relatórios de mutantes sobreviventes em melhorias concretas. Em paralelo, a popularização de Grandes Modelos de Linguagem (*Large Language Models* - LLMs) abriu uma excelente oportunidade: usar Inteligência Artificial (IA) como apoio à escrita e reescrita de testes, mas sob o controle de um processo guiado por evidências. O *Mutation AI Studio* nasce justamente da intersecção dessas duas frentes, utilizando o *mutation testing* como bússola diagnóstica e a IA local (OLLAMA, 2026) como motor para a evolução automática do código de teste.

Este trabalho propõe um Produto Mínimo Viável (MVP) para projetos Java Maven, desenhado para operar localmente com foco em um ciclo controlado de geração e reescrita. Em vez de prometer uma otimização mágica e irrestrita, foi adotado um escopo realista: executar uma geração inicial, medir a pontuação de mutação (*Mutation Score*), identificar as falhas, reescrever os testes focando nessas lacunas e, por fim, medir novamente o resultado. A proposta privilegia a reprodutibilidade, a simplicidade

operacional e a utilidade prática, articulando a teoria de testes com o uso responsável de IA.

1.1 Problema

O problema central desta pesquisa consiste na automação da geração e evolução de testes unitários em projetos Java Maven, utilizando resultados de *mutation testing* (PITEST, 2026), de forma mensurável e com execução local.

A resolução desse problema exige a integração de ferramentas heterogêneas, como Maven, PITest, modelos locais de IA e uma interface gráfica. Metodologicamente, o desafio é definir critérios objetivos que comprovem a melhoria na qualidade da suíte de testes. Devido à natureza acadêmica do projeto, a solução requer um escopo enxuto e reprodutível, focado em orquestrar um fluxo de trabalho prático para o desenvolvedor.

1.2 Aplicabilidade e Motivação

A motivação para este trabalho baseia-se na premissa de que a alta cobertura de código não garante, necessariamente, testes eficazes contra falhas (AMMANN; OFFUTT, 2017). Na prática, suítes de testes extensas podem apresentar baixa capacidade de validação, permitindo que regressões lógicas passem despercebidas. O *mutation testing* aborda essa limitação ao avaliar a capacidade dos testes de identificar alterações intencionais inseridas no código.

A aplicabilidade desta ferramenta concentra-se no ecossistema Java. O uso combinado de Maven, JUnit 5 (JUNIT, 2026) e PITest é comum na academia e na indústria, o que valida o cenário de experimentação. A adoção de uma IA de processamento local atende a dois requisitos técnicos: a privacidade dos dados, mantendo o código-fonte restrito à máquina do usuário, e a redução de custos com requisições em nuvem. Assim, o projeto alinha-se às práticas atuais de automação com infraestrutura simplificada.

1.3 Objetivo Geral

Desenvolver um Produto Mínimo Viável (MVP) capaz de analisar projetos Java Maven, gerar e reescrever testes unitários utilizando IA local, executar *mutation testing* com PITest e apresentar a evolução do *Mutation Score* em um painel visual.

1.4 Objetivos Específicos

- **OE 1:** Analisar os conceitos de testes unitários, qualidade de *software* e *mutation testing*;

- **OE 2:** Projetar a arquitetura de integração entre *frontend*, agente local, IA e processamento de relatórios;
- **OE 3:** Implementar a execução do Maven e PITest por meio de um agente local via Interface de Linha de Comando (CLI) e *Application Programming Interface* (API);
- **OE 4:** Estruturar os resultados da execução em formato *JavaScript Object Notation* (JSON), priorizando a extração de mutantes sobreviventes e métricas comparativas;
- **OE 5:** Integrar a IA local (Ollama) para a geração inicial e a refatoração de testes JUnit 5;
- **OE 6:** Definir um protocolo experimental reprodutível para comparar a suíte de testes antes e depois do ciclo de automação.

1.5 Escopo e Limitações

O escopo do MVP restringe-se a projetos Java gerenciados pelo Maven, excluindo outras ferramentas de automação de *build*, como o Gradle. O sistema requer processamento local e não interage com bancos de dados externos. A execução isolada por meio de contêineres é suportada, mas opcional.

O fluxo de automação foi delimitado a um único ciclo de reescrita de testes. Essa restrição técnica visa reduzir a complexidade computacional e viabilizar a comparação objetiva entre a suíte original e a modificada, evitando dependências de múltiplas iterações difíceis de controlar. A ferramenta atua como um sistema de apoio ao desenvolvedor e não substitui a revisão humana do código gerado.

1.6 Contribuições do Trabalho

As contribuições desta pesquisa dividem-se nos eixos conceitual e prático da engenharia de *software*. No aspecto conceitual, o trabalho demonstra a aplicação do *mutation testing* como um indicador de robustez superior às métricas tradicionais de cobertura de código. No aspecto prático, apresenta o desenvolvimento de uma arquitetura capaz de integrar a automação de *build*, a análise de mutantes e a IA local. Além disso, a pesquisa estabelece um protocolo metodológico para a comparação de testes unitários, o qual pode ser replicado em experimentos futuros.

1.7 Organização do Trabalho

O documento está estruturado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica sobre qualidade de *software*, JUnit 5 e *mutation testing* (**OE 1**). O Capítulo 3 descreve a metodologia e o protocolo de experimentação (**OE 6**). O Capítulo 4 detalha o desenvolvimento do sistema, abrangendo a arquitetura, a implementação do agente local e a integração com a IA (**OE 2**, **OE 3** e **OE 5**). O Capítulo 5 expõe os resultados e validações práticas, incluindo a análise das métricas coletadas em JSON (**OE 4**). Por fim, o Capítulo 6 apresenta as considerações finais e sugestões para trabalhos futuros.

2 Fundamentação Teórica

2.1 Qualidade de *Software* e Papel dos Testes

A qualidade de *software* é tradicionalmente compreendida como um conceito multidimensional, envolvendo atributos como adequação funcional, confiabilidade, usabilidade, eficiência de desempenho, manutenibilidade e portabilidade, conforme os modelos consolidados no mercado de tecnologia (ISO/IEC, 2011). Na prática, garantir essa qualidade ultrapassa o mero desenvolvimento de código funcional. O processo exige assegurar que o sistema mantenha um comportamento consistente, seguro e evolutivo ao longo de seu ciclo de vida. Essa perspectiva é particularmente crítica em sistemas modernos, nos quais integrações contínuas e entregas rápidas ampliam o risco de regressões e falhas não previstas.

Os testes de *software* assumem, portanto, um papel estratégico como mecanismo de verificação e validação. O objetivo central de um teste não é demonstrar que o programa funciona perfeitamente, mas sim maximizar a probabilidade de revelar defeitos ocultos antes do lançamento oficial (MYERS; SANDLER; BADGETT, 2011). Nesse contexto, a indústria consolidou o conceito de suíte de testes. Uma suíte de testes é definida como um conjunto estruturado e metódico de casos, agrupados com o propósito de validar o comportamento de um sistema ou de seus componentes em diferentes cenários de uso. Mais do que um conjunto de verificações isoladas, a suíte atua como uma documentação viva e um repositório executável que atesta a integridade lógica da aplicação (DELAMARO; MALDONADO; JINO, 2016).

As suítes de testes automatizados passaram a ocupar o centro dos processos da Engenharia de *Software* por fornecerem a rede de segurança necessária para manutenções e refatorações estruturais. Em esteiras de integração modernas, esses testes funcionam como um contrato imutável entre o comportamento esperado pelo cliente e o comportamento observado pela máquina.

A simples presença de uma bateria de testes, contudo, não garante qualidade efetiva. Verificações inadequadamente projetadas geram uma falsa sensação de segurança e elevam os custos de manutenção sem agregar capacidade real de detecção de falhas. Testes frágeis, redundantes ou excessivamente genéricos comprometem a produtividade da equipe de engenharia. Assim, a discussão sobre qualidade desloca se de uma métrica puramente quantitativa, baseada no número de testes escritos, para uma investigação qualitativa sobre a precisão dessas verificações.

Avaliar a robustez de uma suíte exige observar a sua sensibilidade diante de anomalias de domínio. Mais importante do que contabilizar quantas linhas de código

são lidas pelo sistema é medir a capacidade dos testes de falharem consistentemente quando a aplicação apresenta comportamentos incorretos. É essa necessidade de rigor prático que motiva a adoção de técnicas avançadas de avaliação analítica, como o teste de mutação.

2.2 Testes Unitários e JUnit 5

Os testes unitários são a base da pirâmide de validação na Engenharia de *Software*, focados em garantir a corretude de pequenas unidades de código, como métodos ou classes, de forma isolada do restante do sistema (FOWLER, 2012). Verificar componentes diante de entradas estritamente controladas permite identificar falhas na raiz do cálculo, reduzindo significativamente o custo financeiro e temporal de correção em etapas avançadas do desenvolvimento. Essa característica torna a validação de unidade indispensável em cenários de manutenção contínua, nos quais alterações sutis podem desencadear efeitos colaterais em módulos distintos da aplicação.

Além da função corretiva em tempo de desenvolvimento, testes unitários bem estruturados servem como a melhor documentação de um projeto. Uma suíte elaborada atua como especificação fiel das regras de negócio. No ecossistema *Java*, esse processo é majoritariamente orquestrado pelo JUnit 5, um *framework* tecnológico que fornece anotações expressivas, suporte nativo a testes parametrizados e integração direta com construtores de projeto como o Maven. Na arquitetura deste Trabalho de Conclusão de Curso, a biblioteca JUnit 5 foi estabelecida como a infraestrutura central sobre a qual as gerações de testes automatizados operam de forma isolada.

A utilidade prática desses testes depende diretamente da sua qualidade semântica de escrita. Princípios como clareza de fluxo, determinismo e independência são cruciais para a consistência da suíte. O modelo de construção estruturado em três etapas, conhecido mundialmente pela sigla AAA (*Arrange, Act, Assert* - Preparar, Agir, Verificar), organiza a escrita do teste de forma a favorecer a legibilidade humana (OSHEROVE, 2012). A negligência dessas boas práticas resulta inevitavelmente em testes altamente acoplados à implementação e incapazes de capturar defeitos silenciosos.

O risco de suítes sintaticamente corretas, mas vazias de valor semântico, é o maior desafio na automação por meio de Grandes Modelos de Linguagem. Uma IA não calibrada pode escrever um componente que atinge o limite máximo da métrica de cobertura estrutural do projeto, mas que carece de asserções rigorosas sobre o estado final das variáveis. A proposta deste trabalho se apresenta de forma diferente. O foco não reside em gerar volume de código de teste funcional, mas sim em construir verificações sensíveis a pequenas falhas sintáticas e aderentes ao comportamento esperado pelas regras de negócio do *software*.

2.3 Métricas Tradicionais e suas Limitações

A cobertura de código (*code coverage*) se consolidou na indústria como a métrica de avaliação de testes mais difundida, devido à facilidade de instrumentação e visualização dos resultados (FOWLER, 2012). Ferramentas de medição mapeiam detalhadamente quais instruções lógicas, ramos de decisão ou assinaturas de métodos foram exercitados pelo menos uma vez durante a passagem da suíte, fornecendo à gerência de projetos um panorama quantitativo do sistema. Essa métrica é muito útil para identificar rapidamente regiões do código que foram completamente negligenciadas durante as etapas de desenvolvimento.

Apesar de seu valor estrutural como diagnóstico inicial, a métrica de cobertura falha de forma considerável em medir a eficácia real das validações em cenários complexos. A simples execução de uma linha de código pelo processador não garante a validação do estado da variável de domínio (AMMANN; OFFUTT, 2017). Um teste abrangente pode percorrer um fluxo de negócio denso de ponta a ponta e terminar seu ciclo sem realizar uma única validação significativa sobre o estado final da operação, gerando altos índices percentuais nos painéis de monitoramento. O resultado natural é um percentual de cobertura elevado que oculta falhas críticas.

Essa limitação impulsionou o desenvolvimento de critérios de avaliação focados na sensibilidade do teste perante anomalias, distanciando o foco do mero tráfego de leitura de instruções. Na rotina prática corporativa, percentuais muito altos de cobertura coexistem diariamente com suítes incapazes de detectar atualizações que causam vazamento de informações sigilosas ou dados incorretos nos ambientes de produção.

A interpretação diária dessas métricas exige bastante critério por parte da equipe de revisão. Os indicadores numéricos continuam indispensáveis como métrica de linha base para aprovar uma entrega técnica, mas são insuficientes como garantia definitiva de integridade contra falhas lógicas no sistema verificado. Neste trabalho focado em automação, a cobertura estatística do código fonte foi abordada primariamente como um dado de apoio, enquanto o peso principal para determinar a eficácia da ferramenta de reescrita foi delegado ao percentual de detecção do *mutation testing*, assumido como indicador rigoroso na aferição da qualidade das asserções. O Quadro 2.1 detalha as principais distinções entre as métricas de cobertura tradicionais e a técnica de mutação.

2.4 *Mutation Testing*

O teste de mutação se destaca na engenharia moderna de sistemas como uma das técnicas mais rigorosas concebidas para auditar a eficácia das suítes automatizadas, pois analisa o comportamento esperado diretamente na fonte do problema. O conceito principal baseia se na injeção intencional de sutis alterações

Quadro 2.1: Diferença entre métricas tradicionais e *mutation testing*

Fonte: Produzido pelos autores (2026).

Aspecto	**Cobertura de código**	**Mutation testing**
Objetivo principal	Indicar o que foi executado pelos testes	Indicar o que os testes conseguem realmente detectar
Unidade observada	Linhas, métodos e ramos	Mutantes gerados por alterações artificiais
Sinal interpretativo	Grau de execução estrutural	Grau de sensibilidade da suíte a defeitos simulados
Risco de falsa confiança	Alto, quando há execução sem boa validação comportamental	Menor, pois exige falha efetiva diante de mutações relevantes
Custo de execução	Geralmente baixo	Maior, devido à execução repetida sobre múltiplos mutantes
Uso neste trabalho	Métrica complementar	Métrica central de comparação

sistemáticas no código de um projeto estável. Essas mudanças simulam os erros cognitivos comumente cometidos por desenvolvedores. Após as rotinas originais do *software* serem submetidas a essa mudança sintática, elas são classificadas como componentes mutantes (JIA; HARMAN, 2011).

A estrutura de decisão dessa metodologia é pautada por regras dedutivas claras: a suíte de testes original é executada sequencialmente contra cada modificação imposta no código (PAPADAKIS et al., 2019). No instante em que ocorre a reprovação da compilação frente ao cálculo adulterado, o mutante é classificado como neutralizado, ou seja, um mutante declarado morto pelo escudo lógico da suíte. Na falha do mecanismo de verificação, o mutante consegue manter sua sobrevida estrutural e os testes atestam sucesso falsamente. Esse resultado aponta que o comportamento central da aplicação foi modificado, mas os testes automatizados da equipe não possuíam capacidade lógica para perceber a regressão (DELAMARO; MALDONADO; JINO, 2016).

Com o respaldo estabelecido através das constatações da ineficácia dos relatórios estáticos limitados somente à análise superficial de linhas varridas, o critério de mutação se consolidou como padrão de excelência na área acadêmica. A premissa central impõe a pergunta: a asserção se corrompe ao sofrer mudanças lógicas? Contudo, a escalabilidade da técnica é consideravelmente prejudicada pelo alto processamento computacional exigido. Cada alteração inserida exige a recompilação e reexecução da suíte completa de testes no ambiente para confirmar a neutralização do defeito.

Nos moldes definidos para a delimitação metodológica deste MVP, o papel dos mutantes sobreviventes transcende o levantamento estatístico habitual focado na gerência de testes. Eles deixam de atuar apenas como falhas em um relatório

para atuar como artefatos direcionadores, guiando as ações de refatoração do código pela IA local (OLLAMA, 2026). A métrica do *Mutation Score* atua como o gatilho responsável por certificar o sucesso do modelo de linguagem na correção automatizada dos componentes de testes unitários fracos.

2.4.1 Operadores de Mutação e Mutantes Equivalentes

A eficiência da análise estrutural de mutação depende diretamente dos operadores designados para que o *framework* consiga quebrar o comportamento de uma classe. Os ecossistemas da linguagem *Java* incluem manipulações algorítmicas pontuais, pautadas na alteração dos parâmetros do controle de fluxo (substituição do operador == para o operador `!=`). Emprega se também a alteração de retornos nos métodos, baseadas na substituição da resposta para o valor `null` ou zero. Tais mutadores compõem o rol delimitado de alterações viáveis, calibrados para não excederem o custo do processador (PITEST, 2026).

Ferramentas agressivas aplicadas de maneira indiscriminada provocam gargalos no volume de execuções nas esteiras de integração da empresa. Em contrapartida, configurações extremamente contidas geram mutações triviais que esvaziam o valor analítico da ferramenta. A adoção controlada dessa técnica exige parametrização rigorosa no ambiente local para viabilizar relatórios processados rapidamente.

O principal obstáculo teórico na adoção dessas bibliotecas reside no processamento dos componentes denominados na literatura de "mutantes equivalentes". O mutante é atestado no formato de equivalência quando a operação sintática injetada no *bytecode* não resulta em alterações verossímeis ou impactos semânticos no *software* consumido pelo usuário. Esse tipo de ocorrência paralisa as refatorações e distorce a margem real de proteção, penalizando as métricas visualizadas de forma injusta (PAPADAKIS et al., 2019). Diferenciar os acertos lógicos genuínos do mar de sinais mascarados pelos mutantes equivalentes persiste na atualidade como um dos grandes desafios de automações estáticas complexas na Engenharia de *Software*.

No contexto da aplicação e uso da inteligência generativa proposta nesta pesquisa, a presença dos mutantes equivalentes compõe uma ameaça direta à validade e sucesso das reescritas. Essas anomalias poluem o contexto do processador com informações irrelevantes, fazendo a máquina artificial buscar correções sintáticas fúteis que elevam o processamento sem nenhuma contrapartida na proteção e integridade do código avaliado no experimento tecnológico.

2.4.2 PITest no Ecossistema Java

O PITest (*Pitest Mutation Testing*) se estabeleceu no ramo industrial e acadêmico como o arcabouço incontestável para alicerçar a técnica analítica de mutação nas

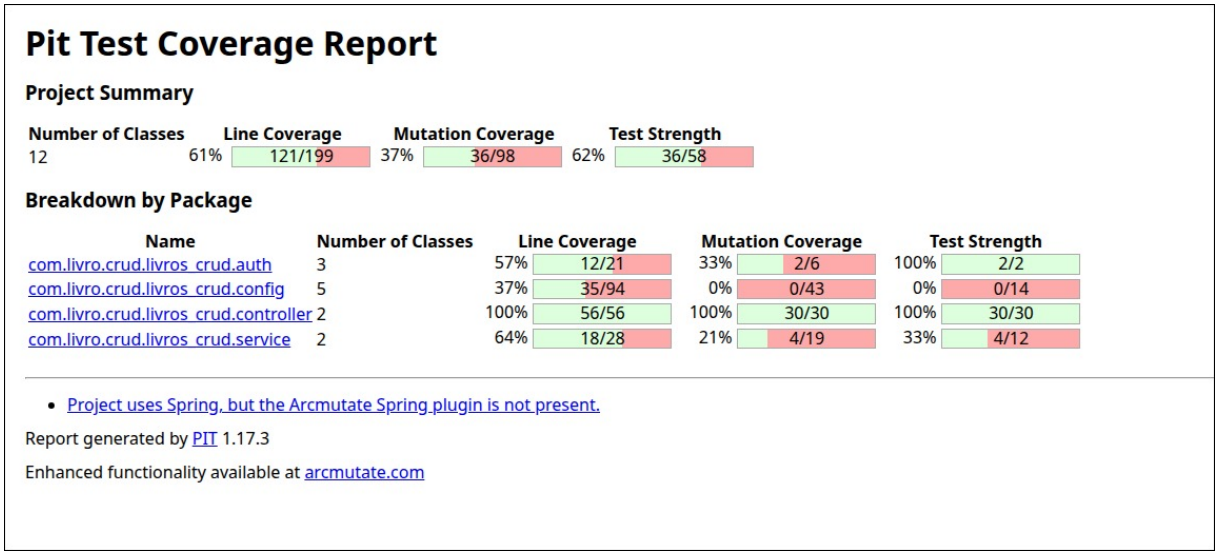
rotinas de compilação contínua em *Java* moderno (PITEST, 2026). A facilidade conferida pelo encaixe gerado por intermédio da integração pontual garante o processamento sem exigir refatorações profundas na base nativa nos projetos orientados via ferramentas estruturais de *build* baseadas no Maven ou no Gradle. As implementações nas esteiras mantêm os diagnósticos e métricas acopladas no ambiente original construído pela equipe de desenvolvimento do domínio sem gerar distúrbios operacionais nos arquivos binários finais.

A agilidade da ferramenta tem suporte fundamentado nas intervenções feitas especificamente no nível do *bytecode* (formato interpretável pela máquina), em vez de realizar as alterações textuais na fonte base do projeto *Java*. Os relatórios emitidos contém alto valor agregado com indicações detalhadas e granulares, focadas na visualização imediata da exata posição na qual o operador automático de mutações causou as inconsistências apontadas no código. Esse controle refinado e direcionado compõe o núcleo primordial de fornecimento de dados para as inteligências implementadas no central *Mutation AI Studio*.

Dentro do encadeamento arquitetural prático desenvolvido na automação, o PITest opera de forma ativa nas duas extremidades: ele confere os relatórios detalhados com as lacunas antes da orquestração do aprendizado de máquina, e submete o texto alterado e processado pela IA a uma nova bateria analítica para constatar se o modelo generativo corrigiu e reparou a deficiência lógica adequadamente.

A ferramenta documenta as incidências do projeto nativo no formato amigável para facilitar o consumo da informação na camada gerencial. A Figura 2.1 expõe visualmente como o formato é traduzido no relatório bruto estruturado via HTTP padrão após toda a análise e varredura do programa *Java* serem consolidadas de modo local com aprovação operacional de leitura pela ferramenta.

A ilustração exibe os dados sumarizados na porção denominada *Project Summary*, que aponta imediatamente o panorama de saúde testável agrupado pelos pacotes da empresa. A imagem ressalta de maneira clara a tese apontada nos referenciais teóricos: as faixas numéricas relativas às métricas tradicionais baseadas na leitura mecânica da linha de código (*Line Coverage*) frequentemente apresentam altos índices em cores verdes. No entanto, a eficácia real submetida e capturada sob o rigoroso teste da cobertura dos mutantes injetados (*Mutation Coverage*) revela que grande parcela do projeto avaliado falha miseravelmente em identificar dados corrompidos. Apesar dos gráficos estáticos nativos apresentados na biblioteca apontarem as lacunas, sua extração sistemática exige o desenvolvimento da plataforma central automatizada desta monografia de modo a tornar a leitura processável, leve e amigável às rotas das arquiteturas generativas em implantação corporativa.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
12	61%	121/199	37%	36/98	62%	36/58

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.livro.crud.livros_crud.auth	3	57%	12/21	33%	2/6	100%	2/2
com.livro.crud.livros_crud.config	5	37%	35/94	0%	0/43	0%	0/14
com.livro.crud.livros_crud.controller	2	100%	56/56	100%	30/30	100%	30/30
com.livro.crud.livros_crud.service	2	64%	18/28	21%	4/19	33%	4/12

- Project uses Spring, but the Arcmutate Spring plugin is not present.

Report generated by PIT 1.17.3

Enhanced functionality available at arcmutate.com

Figura 2.1: Relatório HTML nativo de cobertura gerado pela ferramenta PITest.

Fonte: Produzido pelos autores (2026).

2.5 Inteligência Artificial para Geração e Melhoria de Testes

A capacidade do processamento neural voltada à síntese formatou novos caminhos direcionadores das metodologias automatizadas que apoiam os programadores modernos (BROWN et al., 2020). Sistemas focados em linguagens naturais e compreensão semântica executam a construção primária das lógicas de código e operam para eliminar o tempo inicial improdutivo gasto pelos desenvolvedores na criação do esqueleto inaugural das funções em componentes focados na qualidade das integrações operacionais da engenharia do produto de *software*.

O obstáculo teórico e prático perante as rotinas generativas reside na grande diferença entre produzir textos passíveis de compilação sem falhas e a necessidade real de criar rotinas restritivas que protejam os sistemas contra interrupções na operação (AMMANN; OFFUTT, 2017). Modelos autônomos liberados ao consumo exclusivo de caixas contextuais abertas tendem a produzir as anomalias denominadas como "alucinação". Nessas ocorrências, as ferramentas acoplam bibliotecas e *imports* falsos no ecossistema e criam validações redundantes que apenas forçam o avanço no percentual de leitura mecânica, desvirtuando por completo o objetivo acadêmico associado aos recursos em IA.

A arquitetura elaborada nesta pesquisa atuou na mitigação técnica dos impactos delimitando diretrizes baseadas de forma exclusiva nos resíduos rígidos fornecidos com o processamento do PITest. Os limites foram desenhados na aplicação local focando os parâmetros apontados no ambiente do laboratório universitário restritos à pasta isolada para impedir comportamentos anômalos no *prompt*. A IA recebe instruções restritivas

detalhando os erros do desenvolvedor baseadas nos testes de execução para tratar gargalos apontados sem que seja necessário intervir nas camadas funcionais nativas em *Java* programadas antes de iniciar a inferência.

As implementações práticas em cenários experimentais buscam comprovar que o Produto Mínimo Viável oferece um formato contínuo de suporte, mas negam fundamentações voltadas para a exclusão presencial do engenheiro responsável na checagem diária. A inteligência fornece insumos que apoiam as refatorações mantendo regras imunes em ecossistemas reais para suportar as complexidades sem criar quebras lógicas que corrompam os ambientes digitais mantidos localmente na arquitetura de entrega ágil contínua operante nas engenharias do mundo da computação.

2.5.1 Inferência Local com Ollama

As vantagens técnicas associadas diretamente à operação local de IA consolidam o formato implementado no ambiente, com destaque à integração das capacidades providas do binário livre denominado Ollama (OLLAMA, 2026). Arquiteturas dependentes de infraestruturas nas nuvens corporativas provocam gastos diretos e mensais calculados na tabela de quantidade exigida pelas requisições via rede. Ademais, expor código com domínio sigiloso a serviços terceirizados, submetendo os dados de propriedade intelectual às ameaças na comunicação de servidores geograficamente distantes, compromete requisitos rígidos adotados no mercado interno tecnológico moderno em que companhias vedam tráfegos que violam o tratamento de dados abertos.

O formato operacional local estabelece uma via criptografada por padrão em sua natureza fechada no laboratório mantendo acesso blindado nas máquinas locais contra acessos indevidos fora das propriedades em que as injeções e testes ocorreram com o código empresarial atrelado (OLLAMA, 2026). A ligação processual focada através de requisições de Interface de Linha de Comando no agente de leitura e escrita fornece rapidez necessária às execuções sucessivas sem depender de latências e respostas vindas via *web*.

Apesar da capacidade analítica em contexto das máquinas pequenas afastar a magnitude encontrada em processadores imensos baseados em data centers externos, a opção garante a viabilidade nas reproduções fundamentadas no trabalho científico testado sem barreiras atreladas à exclusão digital. A precisão do teste de mutação em conjunto do processamento autônomo sem internet demonstra total aplicabilidade propícia no desenvolvimento sem necessidade de altos investimentos associados nas entregas, validando por definitivo o estudo elaborado pela equipe do Produto Mínimo Viável no trabalho final proposto com o *software* perante a banca..

3 Metodologia

A pesquisa será conduzida por meio da implementação da ferramenta proposta, da seleção de projetos Java Maven para os experimentos e da avaliação dos resultados utilizando o PITEST e métricas de cobertura de código.

3.1 Natureza da Pesquisa

Este trabalho se caracteriza como uma pesquisa de natureza aplicada, focada na construção e validação de um artefato computacional direcionado a um problema prático da Engenharia de *Software*. A proposta investiga a viabilidade técnica da análise de mutação integrada à IA local para apoiar a evolução de testes, mantendo o rigor em um contexto operacional realista.

A pesquisa também possui dimensão experimental. O estudo compara estados distintos da suíte de testes (antes e depois da intervenção do sistema), balizado por indicadores objetivos, como os relatórios do *PITest*, a quantidade de mutantes sobreviventes, a taxa de sucesso na compilação e o tempo de execução. O objetivo consiste em atestar se a geração de novos testes produziu um impacto mensurável na robustez da suíte original. A verificação empírica ocorre diretamente por meio do sistema desenvolvido, com resultados apresentados ao desenvolvedor no painel interativo.

Para enriquecer a avaliação de resultados com consistência e sem viés estatístico de limitação, a abordagem combina a análise quantitativa (numérica) com uma etapa qualitativa (revisão da semântica dos testes). A leitura minuciosa da equipe perante o código gerado nos projetos alvo complementa os percentuais exibidos no terminal. Essa análise qualitativa traduz a real dimensão dos indicadores numéricos, pois o simples aumento do *Mutation Score* poderia indicar subidas falsas nos dados gerenciais se as asserções geradas pela máquina fossem desprovidas de valor prático. Todo o estudo empírico foi efetuado de modo controlado em máquina local nos repositórios *Java*.

3.2 Estratégia Experimental

A condução do experimento privilegiou a viabilidade prática e o controle analítico, alinhada ao escopo do Produto Mínimo Viável acadêmico. A equipe optou pela validação em um grupo reduzido e controlado de projetos *Java* baseados em *Maven*. Trabalhar com casos delimitados facilita o acompanhamento minucioso do fluxo, tornando mais clara a identificação de falhas e ganhos promovidos pela IA sem introduzir variáveis de ruído arquitetural. Tais ruídos prejudicariam as constatações metodológicas, que exigem avaliações restritas e rigorosas em ambientes de controle de qualidade.

O ciclo comparativo estabelece as restrições essenciais para observar e controlar

sistematicamente os artefatos de *log* e de *build* processados durante as baterias inaugurais efetuadas pela ferramenta. O monitoramento ocorre antes e durante o fornecimento das instruções de correção automática providenciadas ao *Ollama*. Essa orquestração é estritamente delimitada aos resultados extraídos pelo *parser*, baseada nos registros do processo de mutação implementado no *plugin* do projeto executável no terminal.

As melhorias identificadas a partir das evidências colhidas na suíte de testes validam a capacidade analítica do *pipeline* e conferem suporte direto na validação qualitativa das rotinas do desenvolvimento local. O sucesso nas avaliações comprova metodologicamente que o uso da inteligência em ambiente acadêmico, sem desvios estruturais inerentes a conexões de rede externas, é aplicável e efetivo na análise de mutadores.

3.3 Seleção dos Projetos e das Classes Alvo

Para garantir a integridade dos dados, a curadoria dos projetos seguiu critérios técnicos restritos. Os repositórios selecionados exigiam estrutura *Maven* funcional, suporte nativo à execução local pelo comando `mvn test` e compatibilidade com o *plugin* do *PITest*, sem demandar reestruturações profundas no ambiente de arquivos do desenvolvedor. Projetos excessivamente triviais foram descartados por limitarem os estímulos de aprendizado gerado para a IA. Projetos super densos também foram vetados, pois seus altos níveis de acoplamento travariam as conclusões do protótipo devido a gargalos computacionais locais.

A escolha das frentes unitárias seguiu diretrizes alinhadas na garantia de checagem perante lógicas determinísticas. A decisão de filtrar os componentes atrelados a dependências de rede virtual e tráfegos assíncronos previne paralisações incorretas causadas por distúrbios alheios aos códigos originais (AMMANN; OFFUTT, 2017). O Quadro 3.1 apresenta os critérios rigorosos exigidos para a inclusão de um projeto na validação experimental.

3.4 Variáveis e Métricas

A variável central observada recai fortemente sob a consolidação das melhorias apontadas percentualmente pelo *Mutation Score*. A medição ocorre antes das modificações inseridas e novamente após a conclusão definitiva das refatorações realizadas pela IA. Os trabalhos operam nos modos de geração inicial ou em refatorações diretas, apontando para as deficiências rastreadas no código e apresentadas com clareza nos relatórios analíticos em JSON do sistema frontal.

Avaliar o contexto valendo se unicamente da subida de pontos não garante a precisão do diagnóstico. Um índice forjado com sintaxe incorreta gerará quebra

Quadro 3.1: Critérios de seleção para projetos e classes alvo

Elemento	Critério mínimo
Projeto *Maven*	Possuir `pom.xml`, compilação local e testes executáveis
Projeto *Maven*	Permitir inclusão ou ajuste do *plugin* do *PITest*
Projeto *Maven*	Não depender de infraestrutura externa obrigatória para testes unitários
Classe alvo	Conter lógica de negócio ou decisões condicionais observáveis
Classe alvo	Ser passível de teste unitário com baixo acoplamento a banco, rede e sistema de arquivos
Classe alvo	Ter mutações potencialmente relevantes do ponto de vista semântico

Fonte: Produzido pelos autores (2026).

imediata na aprovação da compilação e, consequentemente, quebras na esteira final de entrega. Variáveis independentes de viabilidade, focadas no tempo cronometrado, são registradas no sistema de modo a assegurar o viés prático e medir o consumo de tempo exigido pelas rotinas de automação. O Quadro 3.2 descreve as métricas coletadas em cada rodada.

Quadro 3.2: Métricas observadas no experimento

Métrica	Descrição	Uso na análise
Mutation Score antes	Resultado da primeira execução do *PITest*	Linha de base
Mutation Score depois	Resultado após a reescrita	Medida de melhoria
Mutantes sobreviventes	Quantidade e perfil dos mutantes não mortos	Diagnóstico qualitativo
Tempo por etapa	Duração de *build*, testes, PIT 1, processamento inteligente e PIT 2	Viabilidade operacional
Compilação dos testes	Capacidade de os testes gerados executarem no projeto	Controle de qualidade da saída

Fonte: Produzido pelos autores (2026).

3.5 Protocolo de Execução

As diretrizes focadas no determinismo e garantia de avaliações alicerçadas na imparcialidade seguem protocolos fechados, iniciados estritamente na confirmação de estado apontada pelas compilações preliminares. O modelo garante um fluxo coerente, imune a falsos positivos e erros de dependências que a geração de texto poderia emitir

se a base inicial não contivesse integridade sintática atestada nativamente (JUNIT, 2026).

O arcabouço entra nas diretrizes gerando a varredura primária, a qual fornece os insumos que alimentam a base restritiva enviada ao motor de inferência local. Após a ferramenta devolver os testes finalizados com as adequações pontuais, o sistema grava os dados e repete de modo inflexível a aprovação nas travas do construtor a fim de constatar a avaliação no final do processo da orquestração. Os resultados contrastando o antes e o depois das operações são exibidos na interface visual. O Quadro 3.3 consolida o roteiro de execução seguido pelo experimento.

Quadro 3.3: Protocolo resumido do experimento

Etapa	Descrição
1	Preparar ambiente local, dependências e projeto *Maven*
2	Validar `mvn test` e coletar estado inicial da suíte
3	Executar PIT 1 e registrar métricas de linha de base
4	Extrair mutantes sobreviventes e montar contexto estruturado
5	Gerar ou reescrever testes *JUnit* 5 com base nos mutantes
6	Validar compilação e execução dos testes gerados
7	Executar PIT 2 e comparar os dois relatórios
8	Consolidar dados em JSON e exibir no painel interativo

Fonte: Produzido pelos autores (2026).

3.6 Critérios de Sucesso

Os critérios para atestar o sucesso da aplicação baseiam o foco em estabilidade, descartando a busca ilusória por alcançar teto absoluto no controle estatístico de classes com alto acoplamento (FOWLER, 2012). A pesquisa obtém validação real quando a fundação base do sistema opera o *pipeline* de ponta a ponta sem requisições de redes baseadas em nuvem externas. Essa diretriz comprova que as orquestrações locais fluíram perfeitamente atreladas ao terminal do *Ollama* e geraram os arquivos JSON validados e legíveis pelo visualizador (OLLAMA, 2026).

A transparência diagnóstica do *Mutation AI Studio*, que expõe ao profissional o momento exato em que a fragilidade falhou e a correção proposta, indica a viabilidade do produto. O atestado final consiste em provar a repetibilidade das execuções e evitar que as complexidades da engenharia de qualidade inviabilizem o fluxo diário de integração contínua do programador. O Quadro 3.4 elenca as condições que atestam a validade operacional do artefato.

Quadro 3.4: Critérios de sucesso do MVP

Critério	**Descrição**
Execução de ponta a ponta	O *pipeline* deve rodar do `mvn test` inicial ao relatório final
Viabilidade local	A arquitetura deve funcionar sem dependência obrigatória de API externa
Melhoria mensurável	Deve haver possibilidade de observar diferença entre os relatórios do *PITest* no antes e no depois e na quantidade final dos sobreviventes
Transparência diagnóstica	O sistema deve permitir inspeção visual e rápida dos mutantes sobreviventes e da etapa exata de falha detectada
Reprodutibilidade	As etapas devem poder ser repetidas com o mesmo projeto *Maven* e configuração base inicial adotada pela equipe experimental

Fonte: Produzido pelos autores (2026).

3.7 Instrumentação, Coleta e Armazenamento

A gravação fidedigna com o armazenamento das métricas executadas resulta no mapeamento preciso de latências ocorridas no *pipeline*. O sistema mede o tempo de compilação sintática do *Maven*, cronometra as chamadas do motor de texto local e avalia o peso de processamento que roda as instâncias de mutantes aplicáveis na suíte (PITEST, 2026).

A decisão por limitar a persistência estritamente na forma da padronização JSON foi estabelecida garantindo desacoplamento na construção dos ambientes gráficos da página *web*. Essa modelagem de objeto textual flexível facilita a auditoria tecnológica e permite que futuras equipes expandam os visualizadores baseados na web sem corromper as lógicas centrais estruturadas no formato de *backend*.

3.8 Análise dos Dados

As investigações atuam avaliando o número do ganho no marcador de mutações comparado nos testes executáveis, agrupado às observações focadas na qualidade da semântica dos testes recém criados pelo robô gerador (BROWN et al., 2020). Essa postura confere confiabilidade acadêmica, mitigando os desvios na confiança e evitando atestados de melhorias com métodos artificiais de elevação das métricas numéricas sem valor de proteção útil contra defeitos lógicos.

Estudar as falhas que persistem evidencia as limitações do modelo analítico ao lidar com acoplamentos graves em dados não simuláveis. Essas análises demonstram que o uso da automação deve ser conduzido sempre com a supervisão especializada

do analista, atestando de forma madura que a IA, por si só, não descarta a experiência avaliativa do engenheiro de *software*.

3.9 Cronograma

A estruturação em ciclos mensais buscou isolar os riscos intrincados de integração técnica. Em vez de tratar o desenvolvimento em escopo "cascata", o projeto operou em entregas modulares, garantindo que o acionamento via terminal funcionasse perfeitamente de forma isolada antes da sua conexão com o servidor local e a tela gráfica em *Angular*. O Quadro 3.5 resume a divisão de tarefas em ciclos mensais.

Quadro 3.5: Cronograma resumido por mês

Mês	**Entregas principais**
1	Execução manual da base, preparação do *PITest*, geração inicial de testes e comprovação de fluxo em terminal
2	Implementação do agente local, extração dos relatórios e integração básica com o painel
3	Reescrita automática, comparação antes e depois, documentação técnica e consolidação do TCC

Fonte: Produzido pelos autores (2026).

3.10 Riscos e Mitigações

A orquestração de inferência local e análise de mutação levanta riscos de gargalos de processamento, falhas de síntese de código e inconsistência interpretativa. Modelos de linguagem frequentemente retornam testes com bibliotecas fantasmas ou *loops* infinitos. A prevenção e a delimitação rígida do escopo experimental foram essenciais para salvaguardar a estabilidade.

A estratégia de mitigação envolveu a contenção ativa de variáveis. Em vez de testar em aplicações gigantes, domínios focados foram escolhidos. Classes fortemente acopladas foram filtradas, a compilação foi validada de forma automática e os erros foram interceptados no nascedouro.

- Projetos *Maven* com *build* complexo: mitigados pela seleção de cenários controlados e previamente mapeados;
- Testes gerados com falha sintática: interceptados pela validação obrigatória do `mvn test` imediatamente após a geração autônoma;
- Tempo de execução proibitivo do *PITest*: controlado pela restrição do escopo a classes alvo e mutadores específicos;

- Dependências não resolvidas: prevenidas priorizando componentes lógicos com acoplamento isolável.

4 Desenvolvimento do Sistema

Como resultado da pesquisa, será desenvolvido um sistema para automatizar a geração e a evolução de testes unitários em projetos Java Maven, utilizando IA local e mutation testing para avaliação da qualidade dos testes.

4.1 Visão Geral da Arquitetura

A arquitetura do *Mutation AI Studio* foi concebida para operar localmente, estruturada em três módulos independentes: a interface *web*, o agente orquestrador e a camada de IA baseada no *Ollama*. Para integrar esses pilares, foi desenvolvido um módulo especializado em leitura de dados dedicado à consolidação dos relatórios emitidos pelo *PITest*.

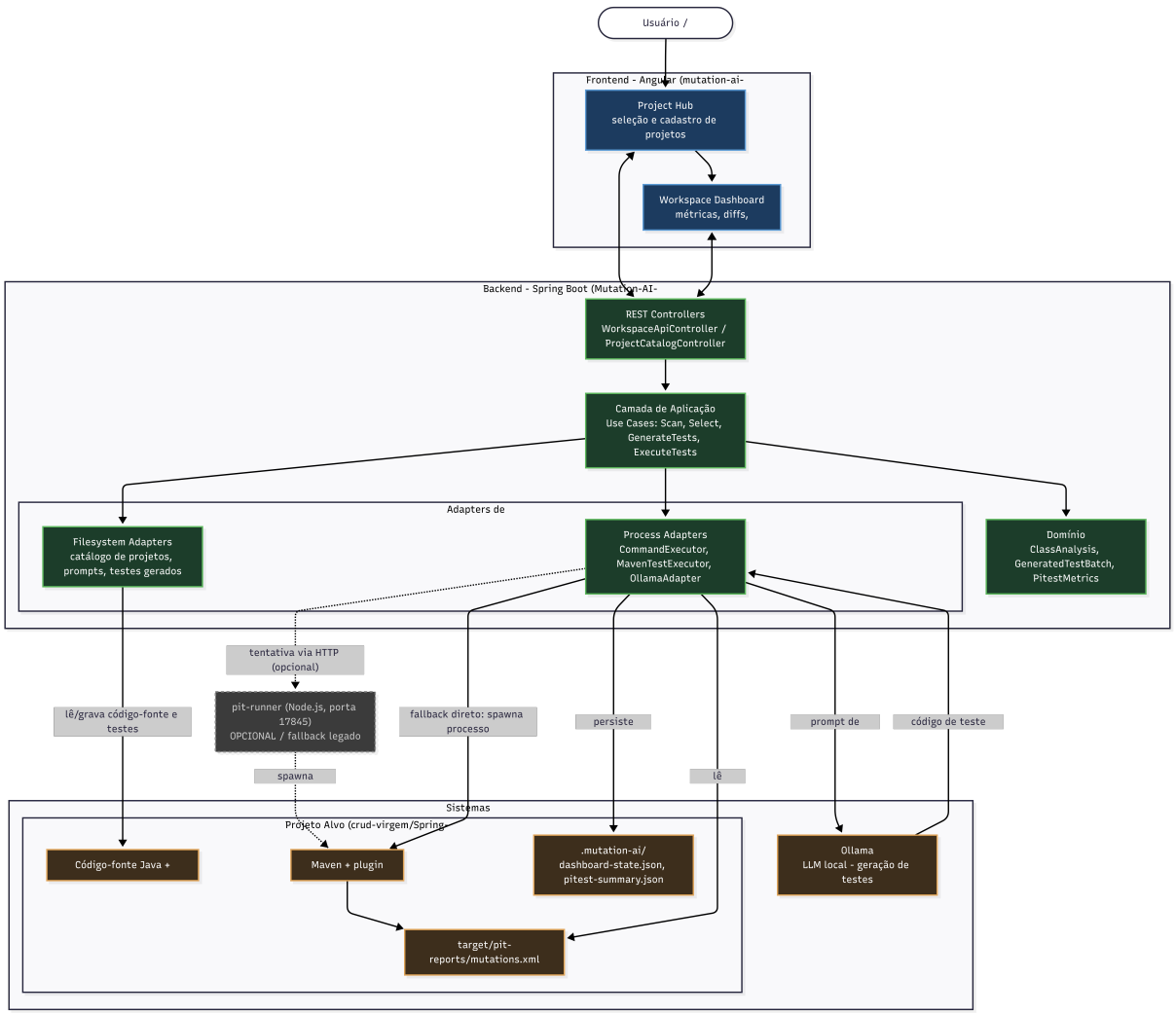
A premissa central desse projeto arquitetural foi garantir uma montagem modular. Era crucial manter a simplicidade necessária para demonstrações acadêmicas, mas com o isolamento adequado para permitir a substituição de peças no futuro. Nesse modelo, a interface gráfica é totalmente desacoplada da execução de compilação. O agente local consolida a lógica de *backend*: ele mapeia o diretório, invoca comandos do *Maven*, processa a mutação e consome os serviços da IA. Esse isolamento resguarda o navegador do usuário final contra os travamentos gerados pelas complexidades de compilação da máquina hospedeira e centraliza o tratamento de erros operacionais.

O diagrama da Figura 4.1 ilustra o caminho da requisição desde a interação do usuário na interface até a camada de orquestração implementada em *Spring Boot*. A camada de aplicação atua como a unidade de controle do sistema, despachando comandos para os adaptadores de processo. Esses adaptadores gerenciam a leitura do código fonte, a invocação do compilador e a comunicação de inferência com o modelo local. Todo o ciclo ocorre no ambiente da máquina hospedeira, assegurando que o código do projeto alvo e as métricas exportadas não trafeguem em servidores em nuvem, garantindo a privacidade e autonomia da ferramenta.

Aprofundando a compreensão do diagrama, a comunicação entre o sistema visual e a camada lógica ocorre de forma assíncrona por meio de requisições baseadas em API. O cliente na *web* envia as instruções e os caminhos de diretórios do repositório local selecionado pelo usuário, e o controlador do *Spring Boot* intercepta essa rota. O sistema inicializa a máquina de estados responsável por governar toda a validação e processamento em segundo plano, sem bloquear a interface de visualização.

Adicionalmente, os adaptadores de processo atuam como a camada de infraestrutura que efetivamente manipula o sistema operacional. O adaptador do *Maven*, por exemplo, é acionado para injetar comandos diretos no terminal e aguardar que a ferramenta processe o *PITest* e gere os relatórios de mutação. As saídas nativas do terminal são processadas, convertidas para formato estruturado e submetidas ao adaptador do *Ollama*. Dessa forma, o ciclo é finalizado gravando o teste atualizado na

Figura 4.1: Diagrama arquitetural e fluxo de dados do *Mutation AI Studio*.



Fonte: Produzido pelos autores (2026).

pasta do repositório antes de notificar o painel via rede local. O Quadro 4.1 resume a responsabilidade de cada módulo estrutural do sistema.

4.2 Interface de Operação

O painel interativo atua como a camada de visibilidade da ferramenta, desenhado para ser intencionalmente enxuto. Em contrapartida à verbosidade de relatórios nativos de ferramentas de qualidade de *software*, a interface condensa estritamente os dados essenciais para o acompanhamento do teste empírico. O usuário insere o caminho do projeto *Maven*, seleciona a classe alvo e monitora a cadência processual. A entrega visual foca na discrepância analítica: expõe a flutuação do *Mutation Score* (antes e depois da intervenção do modelo local), detalha os mutantes sobreviventes e apresenta um *log* unificado para diagnóstico rápido.

O *dashboard* foi projetado seguindo as premissas de uma aplicação de página

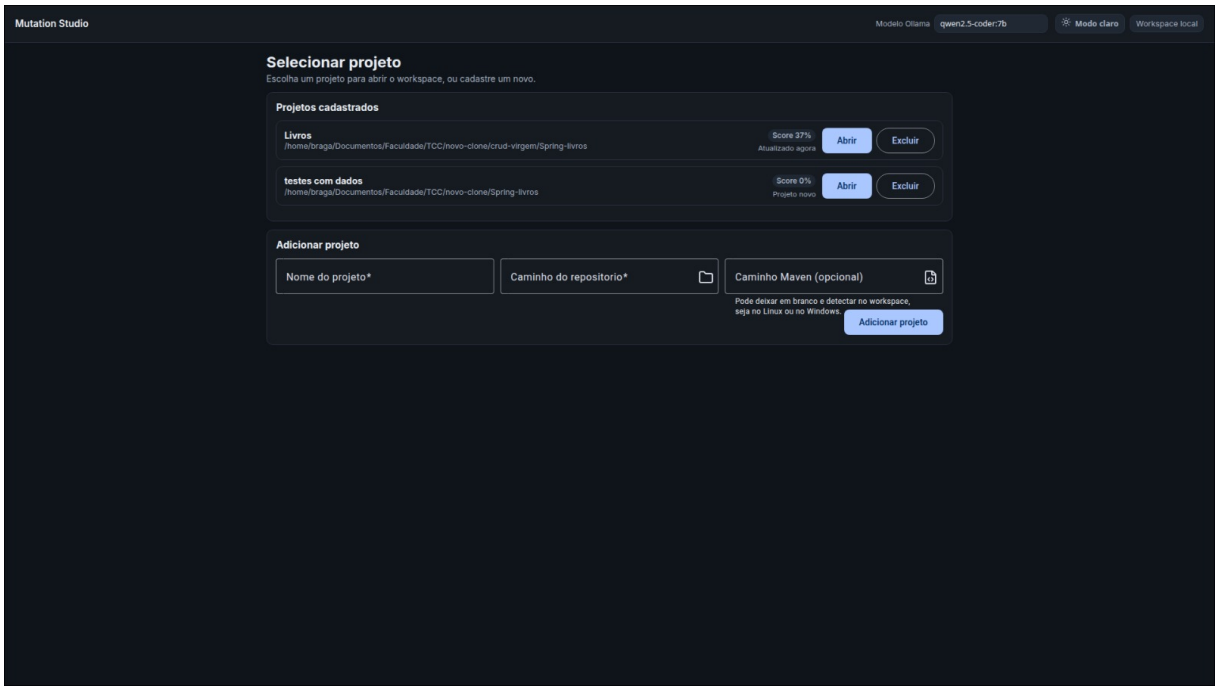
Quadro 4.1: Componentes principais da arquitetura

Componente	**Responsabilidade principal**
Interface em *Angular*	Seleção de projeto e classe, disparo do ciclo avaliativo e visualização de métricas
Agente local	Orquestração das etapas, execução de comandos, validação e exposição das rotas
Processador de relatórios	Leitura do *PITest* e consolidação de dados no formato JSON
IA local	Geração inicial e reescrita de testes com base em contexto controlado
Projeto *Maven* alvo	Sistema analisado, do qual saem código, testes e métricas

Fonte: Produzido pelos autores (2026).

única, priorizando a centralização de comandos para mitigar a dificuldade de leitura das ferramentas de terminal tradicionais. Como pode ser observado na Figura 4.2, a tela inicial estabelece o ponto de partida do fluxo de trabalho do desenvolvedor, exibindo no cabeçalho o estado de execução do ambiente, o indicativo de conexão da rede e o modelo de linguagem atualmente ativo no *backend*.

Figura 4.2: Interface inicial do *dashboard* para gerenciamento e cadastro de projetos.



Fonte: Produzido pelos autores (2026).

A interface é dividida em dois painéis operacionais. O painel superior lista os projetos previamente cadastrados na máquina hospedeira, exemplificados pelos repositórios "Livros" e "testes com dados", detalhando seus caminhos absolutos no

disco rígido e o último resultado numérico consolidado. O painel inferior disponibiliza os campos de entrada necessários para a indexação de novos projetos, exigindo o nome de registro, o caminho físico do repositório mapeado e um campo opcional para a especificação do executável, garantindo a flexibilidade da ferramenta tanto em ambientes *Linux* quanto *Windows*.

4.3 Agente Local

O agente opera como o executor central do *pipeline*. Sua função primordial é traduzir as intenções da interface gráfica em execuções reais no terminal do sistema operacional. O módulo inspeciona as entradas, prepara o escopo da compilação, despacha as *threads* de processamento e decide autonomamente pela continuação ou interrupção do ciclo.

A complexidade de gerenciar ferramentas independentes exigiu que o agente também atuasse como um normalizador de saídas. A plataforma do *Maven*, o binário do *PITest* e a API do *Ollama* possuem contratos de resposta, códigos de erro e tempos de execução completamente distintos. O agente intercepta essas dissonâncias, padroniza as falhas geradas e expõe respostas organizadas e limpas para o consumo da página *web*.

Se a validação inicial do repositório já apresentar falhas sintáticas ou se a injeção do código gerado pela IA corromper a classe, o agente paralisa a rotina imediatamente. Essa trava de segurança impede que o fluxo prossiga e contamine o relatório final com falsos positivos de cobertura.

A Figura 4.3 apresenta um recorte do terminal operacional durante a execução das rotinas de *build* para ilustrar a densidade do processamento nas tarefas ocultas do sistema.

Neste registro, é possível observar a ação estruturada da ferramenta no nível do sistema operacional. O terminal detalha os operadores de mutação aplicados, rastreia o tempo computacional em cada fase e apresenta o sumário estatístico que culmina na compilação executada com sucesso. O Agente Local intercepta ativamente esse volume maciço de texto bruto, realizando a extração exclusiva das métricas relevantes e mascarando a complexidade técnica para o usuário. Dessa forma, é garantido que a interface *web* e o modelo gerativo recebam apenas os dados estruturados exigidos para a etapa de correção.

4.4 Extrator Analítico de Relatórios

A ponte de comunicação estruturada entre as saídas verbosas do *PITest* e a interface *web* é garantida pela camada de extração e formatação (*parser*). O relatório bruto de mutação do *PITest* é gerado prioritariamente em HTML para leitura humana,

Figura 4.3: Painel de execução (*logs*) no terminal ilustrando a saída nativa do *PITest* e do *Maven*.

Fonte: Produzido pelos autores (2026).

ou em XML (um padrão aberto de marcação textual). O módulo de processamento lê esses artefatos ocultos no diretório configurado da aplicação e os consolida em uma estrutura JSON leve.

Além de apenas exibir dados quantitativos, o extrator seleciona atributos cruciais e cria o contexto semântico que alimentará o processador da IA. Os atributos englobam a assinatura do método vulnerável, o operador de mutação que causou a falha e a linha exata de código afetada. Essa padronização atua também como uma camada de proteção da arquitetura proposta, evitando que atualizações e mudanças nos binários originais do *PITest* quebrem o acompanhamento interno das métricas na aplicação desenvolvida neste trabalho.

4.5 Inteligência Artificial Local e Estratégia de Comandos

A injeção de inteligência na rotina local opera sob dois cenários metodológicos. O primeiro foca na geração inicial e ocorre de modo ativo quando a classe alvo ainda não possui qualquer arquivo correspondente com código para suíte de testes. O segundo cenário realiza a refatoração iterativa, focada especificamente em analisar uma suíte preexistente que é guiada aos seus próprios pontos cegos previamente mapeados pelos mutantes sobreviventes detectados no agente (BROWN et al., 2020).

A engenharia estrutural dos comandos (*prompts*) enviados foi rigorosamente delimitada para mitigar as falhas orgânicas dos Grandes Modelos de Linguagem. A

requisição incorpora diretrizes proibitivas, barrando qualquer menção a bibliotecas externas inexistentes no contexto atual e exigindo conformidade técnica obrigatória atrelada à tecnologia *JUnit* 5. No fluxo de reescrita contínua da aplicação, a instrução central não assume o formato genérico "melhore este teste". O comando força o processamento lógico da inferência sobre a lista de mutantes contida no relatório gerado. A IA local é instruída a operar unicamente como um resolvedor focado em sanar lacunas semânticas das falhas específicas listadas (AMMANN; OFFUTT, 2017).

O equilíbrio no volume do contexto enviado pela rede local consiste no maior desafio de estruturação computacional nesta arquitetura. O envio de múltiplos arquivos em formato de árvore contendo toda a base do sistema analisado elevaria expressivamente as garantias lógicas do modelo. Contudo, essa alta demanda excede as capacidades computacionais processuais suportadas em memórias padrão instaladas nas máquinas sem suporte gráfico, prejudicando e quebrando a premissa base exigida na delimitação de acesso universal proposta em um Produto Mínimo Viável focado para rodar em provedoras com pouca infraestrutura instalada.

Diante da limitação sistêmica constatada em máquinas básicas, a abordagem pontual com extrações fracionadas providenciadas no *parser* da esteira é o mecanismo essencial que garante o funcionamento local do *software*. O Agente Local atua orquestrando a injeção do contexto: ele faz a leitura do modelo básico de instruções da máquina e localiza chaves vazias. O módulo preenche as variáveis de preenchimento obrigatório injetando trechos literais dos métodos afetados extraídos na memória. Por fim, o agente envia o conjunto de dados por requisição *HTTP* direta até o *endpoint* local onde reside o servidor que hospeda o motor *LLM* em funcionamento na porta disponibilizada do computador. O Quadro 4.2 mostra as justificativas do recorte de contexto adotado no comando submetido.

4.6 Estratégia de Validação dos Testes Gerados

A emissão de código tecnicamente inviável e não passível de execução é um obstáculo conhecido na automação via máquinas generativas (BROWN et al., 2020). Para prevenir que uma quebra de sintaxe ou a chamada de pacotes inexistentes afetassem negativamente as métricas da avaliação da qualidade nos testes experimentais do trabalho de Engenharia, a arquitetura introduz uma barreira algorítmica de validação logo após a captação da resposta do modelo generativo.

O texto em sintaxe *Java* fornecido na resposta autônoma da API local é gravado no disco físico. O servidor local executa uma rotina limpa de validação no terminal usando o comando base para execução dos testes. A falha nesse teste inicial impede o avanço no ciclo da aplicação. O agente exige, como requisito técnico estrito, que essa etapa apresente *status* positivo de aprovação sem notificações de exceção.

Quadro 4.2: Informações utilizadas nos comandos enviados ao modelo

Entrada	**Finalidade**
Código da classe alvo	Permitir compreensão das assinaturas dos métodos expostos e das regras de negócio ativas
Testes existentes	Evitar duplicação desnecessária da cobertura e manter a formatação de estilo já empregada
Mutantes sobreviventes	Apontar de maneira clara os exatos cenários e condições estruturais que a suíte ainda não foi capaz de avaliar e proteger
Regras de saída	Garantir aderência e compilação focada na restrição de arquitetura imposta pela implementação base (*JUnit* 5 e retornos limpos)
Limites de escopo	Impedir refatorações supérfluas e excessivas que elevem acoplamentos na leitura humana do código gerado pelo robô local

Fonte: Produzido pelos autores (2026).

Apenas após a confirmação da compatibilidade do código injetado o sistema dispara a reativação orgânica do *PITest*. A barreira evita computar testes inválidos como ganho ou perda real nas pontuações visualizadas no painel do usuário final da aplicação.

4.7 Modelo de Dados do Relatório JSON

O tráfego de dados na aplicação resulta em um relatório unificado e consolidado que funciona como o registro analítico principal da auditoria operacional da máquina avaliada. Os registros incluem os dados percentuais indicando o alcance final gerado pela inserção, o *status* absoluto perante a máquina compiladora e um vetor listando as anomalias não resolvidas.

O relatório apresenta adicionalmente parâmetros referentes à temporização. Os campos capturados de maneira isolada demonstram os cronômetros dedicados e transcorridos nas chamadas das etapas de trânsito em execução operacional. Essa característica facilita a identificação técnica dos gargalos computacionais nas avaliações do projeto experimental, isolando e apontando onde reside o custo do tempo em tarefas locais de inferência versus execuções nas varreduras do servidor estático da base compilada. O Quadro 4.3 descreve a tabela consolidada das propriedades capturadas nos resultados do *JSON* da esteira completa do *software*.

4.8 Fluxo Operacional Detalhado

O ciclo inicia as operações quando o painel dispara e fornece o caminho do repositório físico do sistema via roteamento HTTP para a matriz do servidor local de

Quadro 4.3: Campos esperados no relatório consolidado

Campo	**Descrição**
`projectName`	Nome lógico registrado e avaliado no projeto
`targetClass`	Classe e caminho definidos nas configurações originais para validações estritas
`status`	Variável textual do comportamento final (indicadores de sucesso, falha em compilações iniciais ou comandos cancelados)
`mutationScoreBefore`	O percentual lido no final da bateria preliminar estática de mutantes do projeto nativo
`mutationScoreAfter`	Percentual alcançado na esteira de reavaliação utilizando as refatorações geradas pela inteligência
`survivingMutants`	O agrupamento contendo cada defeito estruturado, diagnosticado e categorizado pela não supressão após varreduras na arquitetura implementada
`executionTimes`	Relatório das métricas atestadas evidenciando o custo processual alocado pelas interações de requisições realizadas com o motor generativo do experimento
`logs`	Os pequenos recortes textuais com o alerta de falhas capturadas no roteiro sistêmico da consolidação executável para as auditorias técnicas de rastreamento do analista

Fonte: Produzido pelos autores (2026).

integração. O agente realiza verificações precisas das rotas validadas no diretório para assegurar conformidade e envia comandos de ativação na arquitetura computacional baseada em testes de unidade. A operação inicial alimenta os processamentos estáticos nos injetores de anomalias da infraestrutura. A varredura subsequente filtra as execuções redundantes para fornecer um agrupamento com a lista contendo apenas os sobreviventes remanescentes originais.

Imediatamente após a conclusão desta listagem consolidada do diagnóstico analítico base da classe em avaliação, o agente orquestra os dados cruciais encontrados no arquivo final para formatação das restrições textuais nos *prompts* em trânsito pelas integrações da aplicação. O fluxo convertido alcança a porta hospedada no servidor que gerencia os recursos computacionais atrelados à IA local. A API neural responde a chamada enviando regras sintáticas em *Java* formatadas e corretivas, as quais são salvas e sobrescritas de maneira automática nas bases correspondentes ao código analisado sem inviabilizar as operações.

A revalidação subsequente executada sob aprovação de comando força uma nova leitura base no compilador e no *PITest*. A obtenção positiva de resultados atesta os novos cálculos da esteira encerrando o fluxo orgânico. O aplicativo envia os percentuais consolidados, os quais pintam na apresentação da página um relatório de acompanhamento de maneira visual, amigável e fidedigna sobre o ganho obtido no avanço implementado pela IA.

4.9 Configuração e Execução do MVP

A preparação mínima requerida pela máquina em ambiente hospedeiro responsável pelas rotinas de inspeções executadas na ferramenta *Mutation AI Studio* independe de frentes ou acessos às integrações baseadas nas infraestruturas virtuais ou nuvens operacionais pagas corporativas. A operação estrita presencial dispensa acessos na *web*, desde que as parametrizações da compilação e leitura sejam providenciadas nativamente nas variáveis internas em uso pelo desenvolvedor do laboratório acadêmico ou corporativo.

A inicialização e execução do artefato de avaliação requerem instâncias consolidadas baseadas no ecossistema do JDK compatível, além dos *scripts* do aplicativo executável do *Maven* instalados nas propriedades lógicas da raiz do servidor ativo. As páginas gráficas do painel de monitoramento e interação operam a partir das arquiteturas que englobam a base funcional do *Node.js*.

A IA deve residir de modo autônomo nas especificações de *software* executadas com base no instalador local provido pelo *Ollama* (OLLAMA, 2026). O processamento sem dependência técnica da internet é garantido desde que o modelo generativo contendo as configurações (`qwen2.5-coder:7b`) tenha os seus pesos e pacotes de

parâmetros lógicos baixados previamente e mantidos na memória interna operacional do *hardware* destinado aos experimentos. O teste do repositório selecionado no experimento em *Java* exige de forma exclusiva e mínima a edição pontual nas propriedades das dependências ativas no projeto (arquivo XML) englobando as chamadas focadas e relativas na execução das simulações da plataforma *PITest*. A configuração mínima contorna a lentidão de resposta provocada pelo tráfego de operadoras comerciais com a internet instável e inviabiliza gastos corporativos atrelados a acessos restritos em APIs fornecidas pelo mercado vigente no exterior.

5 Resultados e Discussão

5.1 Cenários de Validação

A validação empírica do sistema foi conduzida sobre um projeto Java real estruturado com o gerenciador de dependências Maven, denominado Spring-livros. A seleção desse repositório buscou certificar se o mecanismo tecnológico testado nesta pesquisa se comporta de maneira previsível em uma tipologia de código que emula as dinâmicas corporativas e de engenharia observadas nos cenários do mercado contemporâneo, avaliando com consistência técnica as capacidades operacionais do LLM na geração inicial em componentes puros de negócio e também no reforço iterativo de validações mais exigentes de lógica (AMMANN; OFFUTT, 2017).

A classe alvo definida perante os avaliadores restritos do time e escolhida dentro do ecossistema do repositório fonte da instituição possui e contém regras de validação focadas, garantindo validações de verificação nos parâmetros efetuados no escopo restrito aos domínios e lógicas pontuais de negócio puras que oferecem margem e densidade para a experimentação de anomalias falsas nas checagens estruturais feitas com mutantes pelo arcabouço tecnológico (PITEST, 2026). Interações ou acoplamentos que se ligavam a chamadas lentas com acessos densos com banco de rede corporativa foram descartados do ambiente do teste metodológico para bloquear a presença injusta de eventuais exceções decorrentes do tráfego ou tempo estourado que as lógicas em teste puro não devem cobrir por não ter competência nem jurisdição aplicáveis às camadas da engenharia *software* implementadas focadas no isolamento.

5.2 Análise das Métricas Coletadas

Para mensurar o impacto analítico real do sistema avaliado no TCC sem pautas focadas em intuição humana sem atestado, o experimento rastreou exaustivamente nas pastas os resultados obtidos antes do processo operante e o *Mutation Score* posterior após a interferência pontual efetuada nas linhas corrigidas localmente a base da IA em execução na área da memória (`qwen2.5-coder:7b`) que serviu ao laboratório de testes restrito, atrelando e comparando o número percentual correspondente ao diagnóstico perante o tempo e atrasos em segundo na máquina avaliada e operante nas rotinas analíticas (OLLAMA, 2026).

Os resultados extraídos confirmam materialmente e percentualmente que as inferências das Inteligências Artificiais e processamento dos textos nos testes mudou categoricamente e objetivamente a barreira construída do *software*. No modelo Spring-livros atestado com dados e resultados coletados sem barreiras e sem envios

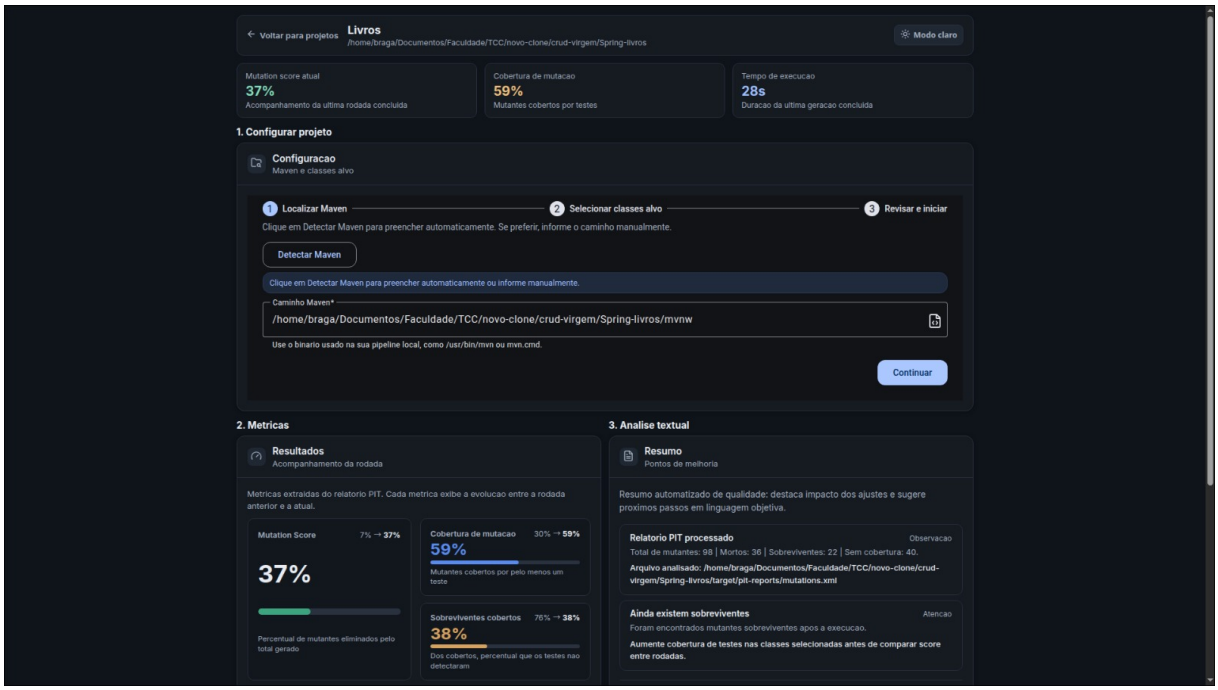
de redes externas, o número percentual englobado em toda a proteção e validação na *Mutation Score* cresceu em linha de modo muito forte de meros 7% na linha inferior apontada pelo relatório primário emitido de forma sintética para chegar ao percentual e linha de topo com 37% (aumento de 30 pontos consolidados localmente e apontados), e simultaneamente, os níveis em coberturas de falhas rastreadas nos pacotes subiram fortemente com base comparada com o PITest indo de apenas 30% da primeira conferência restrita e finalizando em 59% validados. Mutantes que antes possuíam passagem de fluxo verde garantida no JUnit tradicional e testador mecânico sem falhar perante defeitos mascarados e desapercebidos pelo código inicial caíram intensamente, indo para o fundo caindo de 76% de ocorrências que falhavam logicamente para ficar limitados em 38%, números que garantem a aprovação perante os crivos e evidenciam que a IA testou as lógicas importantes sem tautologias falsas.

O esforço nas fases nas quais a infraestrutura gastou em compilação na rotina orgânica completa efetuada nas rotas internas durou meros 28 segundos medidos da tela ao término das execuções atreladas na resposta pontual com a geração da inteligência no console da linha, reforçando totalmente e amplamente o escopo que validou o Produto Mínimo Viável no dia e atestando que os times na vida diária poderiam perfeitamente alocar recursos e espaço processual sem frear as integrações dos programadores locais de projeto *software* ágil atuantes sem atrasos massivos no laboratório computacional. Os registros na contagem contidos nos *logs* evidenciaram ainda nos painéis em angular que do montante gigante avaliado no Maven apontando 98 mutantes originais introduzidos pela manipulação orgânica da biblioteca da universidade encarregada do experimento analítico, os reparos e correções em asserções propostas pelo comando de inteligência aplicáveis causou a neutralização estrita em 36 anomalias com reparo imediato e coerente na infraestrutura lida pelos painéis nativos atrelados às checagens do JUnit (JUNIT, 2026), deixando os mutantes que mantiveram as sobrevivências listadas englobando cerca de 22 nas falhas ainda precisando revisões operadas manualmente da engenharia na base do código para verificar cenários ignorados pelo robô ou refatorados com equivalência incorreta nas variáveis de processamentos sintáticos sem alcance da estrutura neural instalada, juntamente de outros números de lógicas completamente ignoradas pela varredura em 40 mutantes na lista que não receberam a esteira base nas partes sem rastreio de linhas aplicáveis no Quadro 5.1 e atestadas nos JSON gerados na etapa.

Essas subidas fortes e expressivas na barra de qualidade atestada materializam os gráficos interativos implementados organicamente na entrega frontal e comprovam o sistema como forte suporte atestado de diagnóstico em que o engenheiro evita despender leituras complexas estáticas em *HTML* descritas na base no Capítulo 2 desta formatação que expõe dores operacionais de sistemas crus entregues na atualidade no mercado do mundo. O painel operante em execução aponta a métrica ao

operador ilustrando o quadro em verde subindo gradativamente no projeto analisado no Spring-livros atestado (Figura 5.1) de maneira objetiva com visualizações sem atritos.

Figura 5.1: Painel do sistema exibindo a configuração e a comparação de métricas após a rodada de execução.



Fonte: Produzido pelos autores (2026).

Quadro 5.1: Evolução da pontuação e métricas no projeto Spring-livros

Métrica Analisada	Estado Inicial	Estado Final	Variação Absoluta
Mutation Score	7%	37%	+30%
Cobertura de Mutação	30%	59%	+29%
Sobreviventes Cobertos	76%	38%	-38%
Tempo de Execução (*Inference/Build*)	–	28s	Escopo Local

Fonte: Produzido pelos autores (2026).

5.3 Interpretação e Padrões de Comportamento da Inteligência Artificial

As melhorias apontadas conferem atestado empírico direto e comprovam resultados embasados indicando limites operacionais práticos cruciais presentes organicamente no uso contínuo de Grandes Modelos de Linguagem na arquitetura e escrita restrita baseada nas verificações e validações nos pacotes Java da empresa (BROWN et al., 2020). O pico operacional avaliado entregue de valor na ferramenta

e orquestração do *Mutation AI Studio* expõe a funcionalidade estrita sendo forte no momento no qual as lógicas falham devido a carências e deficiências perfeitamente mapeadas por métodos de desenvolvedores tradicionais ineficientes nas práticas sem robustez nas classes implementadas. Nos pontos de deficiências em tratamento contendo as exceções nulas mapeadas pelo PITest originário dos relatórios da avaliação analítica, ou as lógicas relativas às fronteiras falhas ou fronteiras nos condicionais e números, as Inteligências Artificiais e modelos locais atuaram reescrevendo assertivas estritas e fecharam as lacunas semânticas das falhas perfeitamente em assertivas eficientes aplicáveis da biblioteca nativa nos ecossistemas do Java sem apagar partes corretas originais atestadas pelo autor humano base nas arquiteturas em processamento.

Nos cenários de contexto puro nos quais a infraestrutura implementada nos testes com validações base se mostram desenhadas com forte isolamento do acoplamento contido no modelo do escopo do programa gerador, a resposta da refatoração enviada mostrou forte eficácia com precisões muito superiores aos exemplos em nuvem operantes largados à criatividade livre. As estratégias focadas baseadas nas injeções isoladas em blocos de texto contendo puramente os relatórios estritos das classes com foco sem expor bibliotecas fantasmas em importações desconhecidas anulou integralmente problemas clássicos conhecidos como as falsas ocorrências comuns encontradas por Inteligências Artificiais não calibradas (BROWN et al., 2020).

Os resultados avaliativos também indicaram gargalos crônicos presentes organicamente nos limites do modelo em teste. O teto nos acertos operacionais afundou vertiginosamente nos níveis de resoluções de problemas em códigos de classes baseadas contendo ligações com interações complexas do banco e ferramentas com transações externas em rede e infraestruturas do servidor contidos no *software*. Nos testes efetuados nessas funções perante as rotas que trafegam lógicas do sistema *web*, os modelos emitiam saídas defeituosas atreladas às incompatibilidades graves sintáticas no Maven forçando as paralizações do comando base atestadas pelo controlador impedindo o andamento na barreira preventiva das validações implementadas nas frentes arquiteturais. Os indícios atrelados confirmaram que a geração sem os parâmetros atrelados nos repositórios que não recebem contexto massivo sofrem ao gerar integrações falsas simuladas devido às barreiras limitantes do escopo de memória inserido.

Ademais, as características atreladas nos comportamentos naturais e as peculiaridades matemáticas geradas pelo compilador contendo mutantes definidos do tipo equivalentes na literatura técnica impactaram severamente os avanços analíticos em métricas, com as alterações que modificam pacotes restritos de arquivos binários Java em memória sem alterar execuções aplicáveis no comportamento funcional puro

testado perante o consumidor. Foi observado nos atestados emitidos após uma checagem restrita que as amarrações lidas provocam processos vazios sem resultados palpáveis aos times que resultam invariavelmente nas subidas das despesas nas operações operacionais sem retorno algum em integridade nas funções da empresa focada no código de *software* (PAPADAKIS et al., 2019).

5.4 Ameaças à Validade e Síntese de Validação

As confirmações propostas embasadas perante abordagens estatísticas em testes automatizados dotados com redes que geram lógicas dinâmicas contêm e enfrentam limitações impostas atestadas que impedem formulações de regras globais e infalíveis na tecnologia atual (BROWN et al., 2020). A ameaça limitante engloba considerações estritas à validade das conclusões universais emitidas com embasamento focado em testagens locais que se restringem somente às aprovações focadas no repositório isolado testado no Spring-livros e nas classes testadas atestando limites e riscos em aplicar diretamente as regras geradas na arquitetura geral sem validações profundas operadas nas amostras imensas do universo corporativo nas empresas criadoras do mercado atestadas abertas ao mundo.

No campo da validade interna, o determinismo dos testes é arranhado pela natureza randômica do modelo de linguagem. Injetar a mesma classe com os mesmos mutantes pode, em instâncias consecutivas, gerar trechos de lógicas secundárias ou nomenclaturas de método minimamente variadas. O arcabouço de *logs* neutraliza a perda de transparência, mas a variância na esteira de geração local é irremediável. O projeto também confessa uma ameaça de construção ao repousar o selo de qualidade inteiramente sobre o *Mutation Score*. Embora seja um escudo logicamente imbatível se comparado à simples cobertura de linha, a mutação se cega perante métricas vitais de longo prazo, como a qualidade semântica da nomenclatura e a inteligibilidade do código criado pela IA.

O veredito final endossa a missão primária do *Mutation AI Studio*: ele não ambiciona substituir a capacidade de julgamento arquitetural do engenheiro. O sucesso prático reside na habilidade do agente orquestrador de rastrear vulnerabilidades imperceptíveis a olho nu, digerir esse abismo técnico e materializar os resultados em uma proposta automática e instantânea de defesa computacional.

6 Conclusão

A automação da qualidade de *software* enfrenta um dilema crônico: a falsa sensação de segurança gerada por métricas de cobertura superficiais (FOWLER, 2012). Este trabalho propôs o *Mutation AI Studio* não como uma ferramenta definitiva de substituição do engenheiro de testes, mas como um Produto Mínimo Viável (MVP) focado em estreitar o abismo entre testes que apenas rodam pelas linhas de instrução e testes que efetivamente blindam a aplicação contra regressões lógicas perigosas nas corporações de desenvolvimento e engenharia tecnológica no Brasil e no mundo atual.

A pesquisa demonstrou que a integração entre a análise de mutação e a inferência de Grandes Modelos de Linguagem locais é arquiteturalmente viável e operacionalmente valiosa. O sistema orquestrado conseguiu operar integralmente sem dependência de processamento em nuvem, protegendo o código fonte do usuário e eliminando os custos de consumo de interfaces e APIs comerciais operantes em mercado aberto global.

Os resultados empíricos confirmaram que o uso da pontuação diagnosticada como bússola restritiva na IA resolve grande parte do problema de falsificações nos padrões gerados, também catalogadas no meio científico pelo fenômeno da alucinação. Ao invés de pedir ao modelo que criasse testes de forma genérica, algo que frequentemente resulta em checagens repetitivas focadas em passar a barra numérica sem rigor (AMMANN; OFFUTT, 2017), a injeção estrita da amostra dos mutantes sobreviventes originada no processamento local forçou a IA a atuar focada em preencher exclusivamente lacunas lógicas diagnosticadas pela matemática falha originária. A ferramenta provou alta capacidade nos tratamentos contra fluxos de dados soltos e desprotegidos nas execuções.

Apesar do êxito na prova de conceito focada nos painéis testados empiricamente e de modo avaliativo analítico, o escopo delimitado revelou os gargalos operacionais atestados devido aos tempos crescentes no tempo transcorrido focado e necessário nas interações repetidas com as reestruturações perante as bibliotecas dependentes externas contidas em classes grandes e com alto acoplamento na base. Ademais, o problema acadêmico atrelado à presença inevitável contida na incidência natural que aponta os falsos sobreviventes conhecidos por mutantes equivalentes exigiu processamentos inúteis efetuados de maneira inócua pelas Inteligências Artificiais e modelos locais perante modificações sintáticas invisíveis e imperceptíveis perante o comportamento externo requerido por consumidores, encarecendo os gastos de ciclo temporal em processamento lógico (PAPADAKIS et al., 2019).

Retomando os objetivos propostos para este trabalho investigativo e focado

de modo autônomo nas arquiteturas apontadas, o experimento avaliativo confirmou em relatórios que a avaliação analítica por mutantes enriqueceu metodologicamente a força operacional das verificações sintéticas nas frentes testáveis da engenharia, suprimindo o foco nas linhas lidas isoladamente perante testes ineficazes. As validações evidenciam um fluxo capaz de orquestrar perfeitamente a compilação local protegida sem degradação do código de produção base nos computadores acadêmicos. Fica evidente perante as conclusões metodológicas que o motor generativo apoia e soluciona consideravelmente as construções restritas nos domínios funcionais isolados em testes unitários focados na validação e cobertura em bordas, embora apresente ineficiências em componentes fortemente baseados em instâncias operacionais ou arquivos com acessos em tempo não síncrono.

Em suma, as execuções validadas com o protótipo alcançaram plenamente os propósitos propostos e apontaram com evidências reais que as construções da atual IA obtêm resultados de extrema utilidade operacional quando controladas no escopo da engenharia do *software* por sistemas focados em aprovações numéricas rígidas atreladas e condicionadas no controle de compilações fechadas perante testes lógicos em desenvolvimento orgânico corporativo.

6.1 Trabalhos Futuros

A maturidade atingida pelo sistema neste teste abre portas operantes aplicáveis focando avanços comerciais na engenharia tecnológica. Com os apontamentos propostos e atestados, ficam as seguintes sugestões como evolução na área:

- **Integração com Esteiras de Controle CI/CD:** Evoluir o agente local para uma infraestrutura nativa nos repositórios globais, bloqueando requisições que degradem as pontuações e sugerindo o código de correção diretamente nos comentários de revisão operacional contínua na esteira remota mantida e gerida por profissionais da infraestrutura.
- **Expansão de Ecossistema:** Desenvolver a compatibilidade operacional focado no motor Gradle e abarcar componentes estruturados nas modulações de novas versões operantes contidas no ecossistema e estender as leituras orgânicas e operacionais de relatórios baseados em testes provindos da execução de arquivos focados para ecossistemas do Typescript em operação.
- **Engenharia de Contexto Inteligente via AST:** Em vez de submeter e poluir os comandos enviados preenchendo todos os métodos no código, construir algoritmos no construtor que focam recortes precisos contendo as dependências pontuais operantes atreladas exclusivamente ao escopo atestado no mutante, economizando barramento no envio processual exigido pelo gerador neural.

- **Filtro Heurístico para Falsos Positivos:** O modelo carece de análises prévias nas rotinas que inspecionam a sintaxe de modo matemático com a finalidade exclusiva voltada no sentido da remoção programática pontual dos conhecidos e exaustivos mutantes equivalentes apontados anteriormente antes que a suíte provoque desperdícios avaliando defeitos operacionais inócuos e invisíveis perante as avaliações no projeto do desenvolvedor contratado.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

BECK, K. *Test Driven Development: By Example*. Addison-Wesley, 2003.

BROWN, T. et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33, 2020.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. Elsevier, 2nd edition, 2016.

DEMILLO, R. A.; LINPTON, R. J.; SAYWARD, F. G. *Hints on test data selection: Help for the practicing programmer.*. Computer, IEEE, 1978.

DUSSE, F.; SIMOES, A.; MALDONADO, J. C. *Análise de mutantes aplicada a critérios de cobertura de teste a partir de MEFs.*. SAST, 2009.

FOWLER, M. *Test Coverage*. MartinFowler.com. Disponível em: https://martinfowler.com/bliki/TestCoverage.html. Acesso em: 01/07/2026.

ISO/IEC. *ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. International Organization for Standardization, 2011.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2011.

JUNIT. *JUnit 5 User Guide*. Disponível em: https://junit.org/junit5/. Acesso em: 01/07/2026.

MYERS, G.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. Wiley, 3rd edition, 2011.

OLLAMA. *Ollama Documentation*. Disponível em: https://ollama.com. Acesso em: 01/07/2026.

OSHEROVE, R. *The Art of Unit Testing: with examples in C#*. Manning Publications, 2nd edition, 2012.

PAPADAKIS, M. et al. Mutation testing advances: an analysis and survey. *Advances in Computers*, 112, 2019.

PITEST. *PIT Mutation Testing*. Disponível em: https://pitest.org. Acesso em: 01/07/2026.

PRESSMAN, Roger S.; MAXIM, Bruce R. *Engenharia de Software: uma abordagem profissional*. 8. ed. Porto Alegre: AMGH, 2016.

SOMMERVILLE, Ian. *Engenharia de Software*. 10. ed. São Paulo: Pearson, 2019.

A – Exemplo de Prompt para Geração Inicial de Testes

O comando a seguir documenta a instrução base injetada na API do Ollama durante o fluxo de geração vazia inaugural. A estrutura é desenhada para restringir a criatividade do modelo e impor aderência ao padrão exigido no projeto.

```
Voce e um engenheiro de software especialista em Java.
Sua tarefa e gerar testes unitarios JUnit 5 para a classe fornecida.
Objetivo: criar testes estritamente compilaveis e deterministicos.

Restricoes irrevogaveis:
1. Utilize apenas JUnit 5 (org.junit.jupiter.api).
2. NUNCA invente bibs, imports ou metodos que nao existam no contexto fornecido.
3. Evite mockar comportamentos a menos que a interface exija.
4. Foco absoluto em: caminhos felizes, excecoes conhecidas e bordas numericas.

Entradas:
- Codigo da classe alvo: {CONTEXTO_CLASSE}
- Assinaturas de metodos: {CONTEXTO_METODOS}

Saida esperada:
Apenas o bloco de codigo Java do arquivo de teste. Sem explicacoes textuais.
```

B – Exemplo de Prompt para Reescrita Guiada por Mutantes

Na iteração corretiva, a instrução altera o papel do modelo: de gerador primário para refatorador tático. O objetivo é fechar exclusivamente as vulnerabilidades listadas na requisição.

```
Voce esta refatorando uma suite JUnit 5 para aumentar a pontuacao de mutacao.
Analise a classe principal e os testes atuais. Abaixo, fornecemos os mutantes que
sobreviveram na ultima rodada do PITest.

Sua tarefa:
1. Reforce as assercoes (asserts) atuais que estao fracas.
2. Adicione cenarios novos SOMENTE para matar os mutantes listados.
3. Preserve a estrutura original do arquivo de teste.

Mutantes Sobreviventes reportados:
- Linha 42: NEGATE_CONDITIONALS (falha ao verificar fronteira de IF).
- Linha 57: RETURN_VALS (falha ao verificar mutacao para return 0).

Apenas devolva o codigo da suite refatorada, sem preambulos.
```

C – Esquema do Relatório JSON

Para garantir a reprodutibilidade metodológica e desacoplar a arquitetura, o sistema adota o formato de dados abaixo como contrato de saída para cada execução.

```
{
  "projectName": "sample-maven-project",
  "targetClass": "com.example.OrderValidator",
  "status": "BUILD_SUCCESS",
  "metrics": {
    "mutationScoreBefore": 41.0,
    "mutationScoreAfter": 58.0,
    "survivingCountBefore": 14,
    "survivingCountAfter": 4
  },
  "executionTimesMs": {
    "initialBuild": 3105,
    "pitBaseline": 22400,
    "llmInference": 11950,
    "pitValidation": 21700
  },
  "survivingMutants": [
    {
      "method": "validateTotal",
      "line": 42,
      "mutator": "NEGATE_CONDITIONALS",
      "description": "changed conditional boundary"
    }
  ]
}
```

D – Configuração do PITest no Arquivo POM

Trecho de referência do arquivo de inicialização utilizado para acoplar o motor de mutação às validações locais, garantindo a emissão dos relatórios estruturados que são consumidos pela arquitetura.

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.15.0</version>
  <configuration>
    <targetClasses>
      <param>com.example.*</param>
    </targetClasses>
    <targetTests>
      <param>com.example.*Test</param>
    </targetTests>
    <outputFormats>
      <param>HTML</param>
      <param>XML</param>
    </outputFormats>
    <threads>2</threads>
  </configuration>
</plugin>
```

E – Exemplo de Código de Teste Gerado pelo Mutation AI Studio

Este apêndice apresenta um fragmento real da classe gerada de forma autônoma pelo modelo *qwen2.5-coder:7b* durante a execução no projeto Spring Livros. O código evidencia a capacidade do agente em respeitar o contexto de testes unitários e a biblioteca de *mock*, elaborando cenários aderentes às regras de negócio da aplicação.

```
@ExtendWith(MockitoExtension.class)
public class LivroServiceTest {
    @Mock
    private LivroRepository livroRepository;
    @InjectMocks
    private LivroService subject;

    @Test
    public void testSaveLivro() throws Exception {
        Livro livro = new Livro();
        livro.setTitulo("Java Programming");
        livro.setAno(2023);
        when(livroRepository.save(any(Livro.class))).thenReturn(livro);
        Livro result = subject.saveLivro(livro);
        assertEquals(livro, result);
        verify(livroRepository).save(livro);
    }

    @Test
    public void testUpdateNotFound() throws Exception {
        Long id = 1L;
        Livro livroAtualizado = new Livro();
        livroAtualizado.setTitulo("Java Programming");
        livroAtualizado.setAno(2023);
        when(livroRepository.findById(id)).thenReturn(Optional.empty());
        assertThrows(NullPointerException.class, () ->
            subject.update(id, livroAtualizado));
        verify(livroRepository).findById(id);
    }
}
```

F – Exemplo de Prompt Utilizado para Revisão Gramatical da Monografia

Para garantir a integridade intelectual e autoral deste Trabalho de Conclusão de Curso, o uso de IA na redação final foi estritamente limitado à revisão ortográfica e gramatical. O comando a seguir documenta a instrução exata fornecida ao modelo para atuar exclusivamente como revisor do texto acadêmico, bloqueando qualquer alteração semântica ou geração criativa de conteúdo.

```
Atue estritamente como um revisor ortografico e gramatical academico.
Abaixo fornecerei blocos de texto da minha monografia de Engenharia de Software.
Sua unica e exclusiva funcao e corrigir:
- Erros de digitacao e ortografia.
- Problemas de concordancia verbal e nominal.
- Pontuacao e regencia.

REGRAS ABSOLUTAS:
1. E ESTRITAMENTE PROIBIDO adicionar qualquer nova informacao, ideia, argumento
ou conclusao.
2. NAO reescreva o estilo do autor nem altere a estrutura logica dos paragrafos.
3. Mantenha intactos todos os jargoes tecnicos e estrangeirismos.
4. Preserve absolutamente todas as tags, chaves e marcacoes LaTeX originais.

Devolva apenas o texto corrigido, sem adicionar nenhum comentario extra.

Texto:
[INSERIR TEXTO DO CAPITULO AQUI]
```