v7.n11.2026

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

ANÁLISE HUMANA *VERSUS* ANÁLISE AUTOMATIZADA COM SONARQUBE: COMPARAÇÃO DA CAPACIDADE DE IDENTIFICAÇÃO DE PROBLEMAS DE QUALIDADE DE *SOFTWARE*

HUMAN VERSUS AUTOMATED ANALYSIS WITH SONARQUBE: COMPARING THE ABILITY TO IDENTIFY SOFTWARE QUALITY PROBLEMS

ANÁLISIS HUMANO VERSUS ANÁLISIS AUTOMATIZADO CON SONARQUBE: COMPARACIÓN DE LA CAPACIDAD DE IDENTIFICAR PROBLEMAS DE CALIDAD DE SOFTWARE

Julia dos Santos Daniel[1], José Eduardo Ribeiro[2]

e7119581
https://doi.org/10.47820/recima21.v7i11.9581

PUBLICADO: 11/2026

RESUMO

Este artigo compara a análise humana e a análise automatizada com o SonarQube quanto à capacidade de identificar problemas de qualidade de *software*, tratando a Inteligência Artificial como contexto e ferramenta auxiliar. Trata-se de uma pesquisa mista e exploratória, com revisão bibliográfica e prova de conceito. Um sistema acadêmico em Java, gerado com o auxílio do Claude e com falhas intencionais, foi analisado por um avaliador com conhecimento intermediário e pelo SonarQube Community Build 26.8.0. Os resultados indicam que cada abordagem encontrou problemas que a outra não encontrou e que parte dos defeitos conhecidos passou despercebida por ambas, o que sugere que se complementam. As limitações são o sistema pequeno, o único avaliador e o catálogo de defeitos elaborado após a geração do código.

PALAVRAS-CHAVE: Análise Automatizada. Análise Estática. Inteligência Artificial. Prova de Conceito. Qualidade de *Software*. SonarQube.

ABSTRACT

This article compares human analysis and automated analysis with SonarQube regarding the ability to identify software quality problems, treating Artificial Intelligence as context and as an auxiliary tool. It is a mixed, exploratory study, with a literature review and a proof of concept. A small academic Java system, generated with the help of Claude and containing intentional flaws, was analyzed by an evaluator with intermediate knowledge and by SonarQube Community Build 26.8.0. The results indicate that each approach found problems the other did not, and that some of the known defects went unnoticed by both, which suggests that they complement each other. The limitations are the small system, the single evaluator, and a defect catalog built after the code was generated.

KEYWORDS: *Automated Analysis. Static Analysis. Artificial Intelligence. Proof of Concept. Software Quality. SonarQube.*

RESUMEN

Este artículo compara el análisis humano y el análisis automatizado con SonarQube en cuanto a la capacidad de identificar problemas de calidad de software, tratando la Inteligencia Artificial como contexto y herramienta auxiliar.

[1] Graduanda em Sistemas de Informação pela Universidade de Araraquara (UNIARA), Araraquara-SP.
[2] Mestre em Ciência da Computação (UFSCar), graduado em Sistemas de Informação, coordenador e docente dos cursos de Engenharia de Computação e Sistemas de Informação da Universidade de Araraquara (UNIARA), Araraquara-SP.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Se trata de una investigación mixta y exploratoria, con revisión bibliográfica y prueba de concepto. Un sistema académico en Java, generado con la ayuda de Claude y con fallos intencionales, fue analizado por un evaluador con conocimiento intermedio y por SonarQube Community Build 26.8.0. Los resultados indican que cada enfoque encontró problemas que el otro no encontró y que parte de los defectos conocidos pasó inadvertida para ambos, lo que sugiere que se complementan. Las limitaciones son el sistema pequeño, el único evaluador y el catálogo de defectos elaborado después de la generación del código.

PALABRAS CLAVE: *Análisis Automatizado. Análisis Estático. Inteligencia Artificial. Prueba de Concepto. Calidad de Software. SonarQube.*

1. INTRODUÇÃO

1.1. Contextualização

Nos últimos anos, a Inteligência Artificial (IA) passou a ser usada em tarefas ligadas à qualidade de *software*, como encontrar falhas e automatizar verificações. Isso aconteceu por causa da evolução do aprendizado de máquina e do aumento do poder de processamento, que hoje permitem analisar um sistema inteiro em poucos segundos. Antes disso, a qualidade dependia quase só das pessoas: revisão de código feita por colegas, testes manuais e inspeções. Esses métodos funcionam, mas levam tempo e o resultado muda conforme a atenção e a experiência de quem revisa. A IA não tira o papel do ser humano; ela entra como um apoio a mais nas decisões técnicas durante o desenvolvimento.

A qualidade de *software* é a capacidade de um sistema cumprir os requisitos funcionais e não funcionais definidos para ele, com características como confiabilidade, manutenibilidade, eficiência, segurança, usabilidade e desempenho. A norma ISO/IEC 25010 (ISO/IEC, 2023) organiza esses atributos e serve para verificar se o produto atende ao usuário e aos objetivos da organização. Hoje a qualidade é uma exigência, e não apenas um diferencial, porque uma falha pode causar prejuízo financeiro, perda de dados e problemas para quem usa o sistema. O risco é maior em áreas como saúde, finanças e infraestrutura, onde um erro que ninguém percebeu pode afetar a segurança de quem depende do serviço. Por isso, a avaliação da qualidade não fica mais restrita ao fim do projeto e acompanha todo o ciclo de vida do *software*, do levantamento de requisitos até a manutenção.

Nesse contexto, existem ferramentas automatizadas que ajudam o desenvolvedor a analisar o código-fonte, achar vulnerabilidades e defeitos e acompanhar métricas de qualidade. Elas usam a análise estática, feita direto no código sem executar o programa, ou a análise dinâmica, feita com o sistema em funcionamento. Com isso, os problemas aparecem mais cedo, quando corrigir costuma custar bem menos do que no fim do projeto.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Surge então a pergunta de quais problemas cada tipo de análise encontra, a feita por uma pessoa e a feita por uma ferramenta, e se os resultados se complementam. A literatura já reconhece as vantagens da automação, mas são poucos os estudos práticos que comparam os dois lados sobre o mesmo código-fonte. É essa lacuna que este trabalho procura tratar.

A IA aparece neste trabalho de duas formas. Primeiro como contexto, já que sua expansão aumentou o uso de ferramentas automatizadas na Engenharia de *Software*. Depois como ferramenta de apoio, pois o Claude foi usado para gerar o código analisado e executar os comandos de análise. O SonarQube é um analisador estático que aplica regras já definidas. Assim, o que se compara aqui é a abordagem de análise (humana ou automatizada), e não o desempenho de modelos de IA na detecção de defeitos.

1.2. Justificativa

Este estudo se justifica pelo uso cada vez maior de ferramentas automatizadas no desenvolvimento de *software*, algumas delas com apoio de IA, e pela necessidade de entender o que elas mudam na qualidade dos sistemas. Análise estática, assistentes de código baseados em modelos de linguagem e recomendações automáticas já fazem parte do dia a dia de muitas equipes. Por isso, é útil saber, com evidências, o que essas ferramentas realmente encontram e como isso se compara ao que um avaliador humano encontra. Ainda existem desafios de confiabilidade, interpretabilidade e eficácia. Segundo Russell e Norvig (2021), os sistemas de IA avançaram bastante na automação de tarefas complexas e na análise de grandes volumes de dados, mas seu uso em contextos críticos exige cuidado com a transparência dos modelos, a qualidade dos dados e a confiabilidade dos resultados. Na Engenharia de *Software* isso pesa ainda mais, porque uma decisão errada pode deixar passar uma falha real ou, no outro extremo, gerar tantas recomendações irrelevantes que sobrecarregam quem faz a manutenção do código.

Este trabalho também contribui ao apresentar uma prova de conceito em que a análise humana e a automatizada examinam o mesmo código-fonte, gerando evidências que podem servir de referência para pesquisas futuras e para profissionais que queiram conhecer o que uma ferramenta de análise identifica em comparação com uma revisão manual.

1.3. Objetivo geral

O objetivo geral deste trabalho é comparar a capacidade de identificar problemas de qualidade de *software* de uma análise humana e de uma análise automatizada com o SonarQube, verificando em que medida os resultados se complementam. A Inteligência Artificial entra como contexto e como ferramenta de apoio ao processo.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

1.4. Objetivos específicos

Para alcançar o objetivo geral proposto, foram definidos os seguintes objetivos específicos:

a) identificar os métodos tradicionais de garantia da qualidade de *software*, como testes manuais, revisões de código e uso de métricas específicas;

b) contextualizar o uso da Inteligência Artificial na Engenharia de *Software*, diferenciando-a da análise estática baseada em regras predefinidas;

c) investigar as principais aplicações de ferramentas automatizadas na análise de qualidade de *software*, considerando tanto a análise estática quanto assistentes baseados em modelos de linguagem;

d) desenvolver uma prova de conceito a partir de um código-fonte elaborado, com o auxílio do Claude, para conter falhas intencionais, e catalogar os defeitos conhecidos desse código em uma taxonomia comum;

e) comparar os resultados de uma análise humana e de uma análise automatizada com o SonarQube, considerando principalmente o tipo e a natureza dos problemas identificados e, como métrica secundária, o tempo de execução;

f) avaliar os benefícios, limitações e desafios de cada abordagem, discutindo em que medida podem se complementar no trabalho do profissional de desenvolvimento de *software*.

1.5. Problema de pesquisa

A qualidade de *software* é uma etapa importante do desenvolvimento, pois envolve achar erros, falhas e melhorias possíveis antes de o sistema chegar aos usuários. Com o avanço da Inteligência Artificial, surgiram ferramentas que ajudam os desenvolvedores a analisar código, identificar problemas e fazer testes.

Mesmo assim, ainda há dúvidas sobre a eficiência dessas ferramentas e sobre o papel do profissional nesse processo. Elas podem facilitar e agilizar algumas atividades, mas os resultados precisam ser analisados, porque podem ocorrer interpretações erradas ou situações que dependem do conhecimento do contexto do *software*.

Diante disso, estabelece-se como problema de pesquisa: quais tipos de problemas de qualidade de *software* são identificados por uma análise humana e por uma análise automatizada com o SonarQube e em que medida seus resultados são complementares?

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

1.6. Hipótese

Com base na revisão da literatura, parte-se da hipótese de que a análise automatizada com o SonarQube e a análise humana identificam conjuntos de problemas diferentes e só em parte coincidentes. Espera-se que a ferramenta vá melhor na varredura rápida e sistemática de problemas que as regras predefinidas conseguem detectar, como manutenibilidade e convenções de código, e que o avaliador humano vá melhor nos problemas que dependem de contexto, de lógica e de regra de negócio. Assim, as duas abordagens seriam complementares. Considera-se também que a interpretação dos resultados e a priorização das correções continuam sendo responsabilidade do profissional, já que a intenção original do desenvolvedor dificilmente é capturada por completo por uma ferramenta automatizada.

2. REFERENCIAL TEÓRICO

2.1. Qualidade de *software* e garantia da qualidade

A garantia da qualidade de *software* é parte essencial da Engenharia de *Software* e pode ser entendida como o conjunto de atividades planejadas e sistemáticas necessárias para garantir que o produto atenda aos requisitos técnicos definidos (PRESSMAN; MAXIM, 2016). A confiabilidade e a eficiência dos sistemas afetam organizações de vários setores, como saúde, educação e indústria (PRESSMAN; MAXIM, 2016; SOMMERVILLE, 2019). Por isso, uma falha de qualidade é um problema técnico e também um risco operacional e financeiro para quem depende do sistema. Tradicionalmente, a qualidade é garantida com testes manuais, revisões de código e métricas como complexidade ciclomática, cobertura de testes e taxa de duplicação de código. Esses processos, porém, podem ser limitados em velocidade, precisão e capacidade de detectar falhas complexas, principalmente em sistemas grandes (HARMAN; MANSOURI; ZHANG, 2012). Isso motivou a busca por abordagens complementares, que ampliem a cobertura e a agilidade das inspeções.

2.2. Inteligência artificial aplicada à engenharia de *software*

A Inteligência Artificial (IA) vem sendo usada para transformar a garantia da qualidade. Técnicas de aprendizado de máquina, redes neurais e algoritmos evolutivos analisam grandes volumes de dados e identificam padrões que ajudam a prever defeitos no *software* (RUSSELL; NORVIG, 2021). Allamanis et al. (2018), em uma revisão sobre aprendizado de máquina aplicado a código-fonte, mostram que modelos treinados em grandes repositórios conseguem captar regularidades estatísticas da linguagem de programação. Isso permite tarefas como sugerir

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

código, detectar anomalias e reconhecer padrões associados a defeitos. A ideia é que o código-fonte, como a linguagem natural, tem padrões que se repetem e que modelos computacionais podem aprender, o que permite apontar trechos com mais chance de conter defeitos e até sugerir correções automáticas. Hellendoorn e Devanbu (2017) discutem os limites e o potencial de modelos estatísticos de linguagem aplicados ao código-fonte.

Mais recentemente, os modelos de linguagem de grande escala (*Large Language Models*, LLMs) ampliaram o uso de IA generativa na Engenharia de *Software*, inclusive na geração e na revisão de código (CHEN et al., 2021). Esses modelos são treinados com muito código disponível publicamente, e por isso conseguem completar trechos a partir do contexto dado pelo desenvolvedor e, em muitos casos, transformar instruções em linguagem natural em implementações que funcionam. O uso dessas abordagens ajuda a detectar erros, melhora os processos de teste e apoia decisões sobre manutenção e melhorias do código (MALHOTRA, 2015; BATARSEH et al., 2020).

2.3. Ferramentas automatizadas de análise de qualidade

Entre as ferramentas mais usadas na indústria para análise estática de código está o SonarQube. Ele inspeciona o código-fonte e classifica as ocorrências em três categorias principais, *Bugs*, *Vulnerabilities* e *Code Smells*, e também calcula métricas como cobertura de testes, duplicação de código e complexidade ciclomática (SONARSOURCE, 2024). Cada ocorrência recebe um nível de severidade, o que permite às equipes corrigir primeiro os problemas mais críticos. Há ainda ferramentas parecidas baseadas em regras e assistentes de código com IA generativa, como o GitHub Copilot, que usa modelos de linguagem para sugerir código e apontar possíveis problemas durante o desenvolvimento (CHEN et al., 2021).

Estudos recentes indicam que usar IA em atividades de qualidade de *software* pode reduzir custos, aumentar a eficiência do desenvolvimento e melhorar a confiabilidade dos sistemas (MENZIES; ZIMMERMANN, 2013). Modelos de aprendizado supervisionado e não supervisionado têm sido usados para prever falhas em módulos, o que ajuda a localizar áreas propensas a erros e a priorizar testes conforme o risco (HALL et al., 2012). Essa capacidade de previsão pesa mais em projetos grandes, nos quais não dá para testar todos os módulos de forma exaustiva e é preciso concentrar a verificação nos componentes com maior chance de ter defeitos.

Apesar dos benefícios, adotar IA na qualidade de *software* traz desafios. Entre eles estão a necessidade de bases de dados consistentes e de boa qualidade, a dificuldade de construir modelos preditivos e a interpretação dos resultados, que muitas vezes exige conhecimento

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

especializado (HALL et al., 2012; BATARSEH et al., 2020). Algumas ferramentas automatizadas também geram muitas recomendações de baixa severidade, e isso atrapalha a priorização das correções quando não há um processo claro de triagem. Essas limitações mostram lacunas no conhecimento e abrem espaço para novas pesquisas sobre metodologias mais eficazes e flexíveis.

A revisão da literatura mostra, portanto, que integrar IA e ferramentas automatizadas aos processos de garantia da qualidade é um avanço importante para a Engenharia de *Software*, com potencial de aumentar a eficiência, reduzir falhas e melhorar a gestão da qualidade de sistemas complexos. É preciso, porém, separar duas categorias. Ferramentas de análise estática, como o SonarQube, aplicam regras predefinidas e tendem a dar sempre o mesmo resultado para o mesmo código. Já modelos de aprendizado de máquina e de linguagem inferem padrões a partir de dados e podem responder de forma diferente à mesma entrada. É nesse contexto que entra a prova de conceito deste trabalho, que examina na prática como uma ferramenta de análise estática se compara a um avaliador humano sobre o mesmo código-fonte.

3. METODOLOGIA

3.1. Caracterização da pesquisa

A pesquisa é de abordagem mista e natureza exploratória. Ela combina dados quantitativos descritivos (tempo de execução, contagem e classificação das ocorrências) com a interpretação qualitativa dos resultados e tem duas etapas: uma revisão bibliográfica e uma prova de conceito. Na prova de conceito, a ferramenta de IA generativa Claude (ANTHROPIC, 2026) atuou como auxiliar: gerou o código-fonte Java, escrito com falhas intencionais, e executou pelo ambiente de linha de comando (CMD) os comandos necessários à análise. A análise automatizada foi feita com o SonarQube Community Build 26.8.0, instalado localmente com JDK 21 e sem regras personalizadas. Os resultados foram comparados com os de uma inspeção manual do mesmo código, feita por um avaliador independente, com conhecimento intermediário em Java e sem ajuda de ferramentas automatizadas, que anotou cada problema encontrado e cronometrou o tempo gasto. O SonarQube é uma ferramenta de análise estática baseada em regras predefinidas. Versões recentes têm recursos apoiados por IA, como a sinalização de código gerado por IA e recomendações de correção (SONARSOURCE, 2024), mas os resultados medidos aqui vêm da análise por regras. Por isso, a IA é tratada como contexto e ferramenta auxiliar, e o que se compara é a abordagem de análise (humana ou automatizada). Os procedimentos detalhados estão na seção 3.2.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

3.2. Procedimentos da pesquisa

A pesquisa foi feita em duas etapas. A primeira foi um levantamento bibliográfico sobre qualidade de *software*, Engenharia de *Software*, Inteligência Artificial e ferramentas automatizadas de análise de código. A seleção foi narrativa, e não sistemática: foram incluídos livros de referência de Engenharia de *Software* e de Inteligência Artificial, artigos de periódicos e eventos científicos (ACM, IEEE e Elsevier, entre outros) e a documentação oficial das ferramentas, desde que tratassem de qualidade de *software*, de IA aplicada à Engenharia de *Software* ou de análise estática de código e tivessem autoria, veículo de publicação e, quando existente, DOI verificáveis. As referências com DOI foram conferidas nas bases Crossref e DBLP, e as fontes sem esses dados foram descartadas. Como não houve protocolo formal de busca, a revisão não pretende ser exaustiva, o que é registrado como limitação.

A segunda etapa foi a prova de conceito, que comparou a capacidade de identificar problemas de qualidade de *software* da análise humana e da análise automatizada. O código gerado foi submetido ao SonarQube Community Build 26.8.0, ferramenta muito usada na indústria para análise estática, sem regras personalizadas (projeto “poc-cadastro-alunos”, com fontes no diretório “src” e binários compilados em “bin”).

Na prova de conceito foi usado um código-fonte Java de pequeno porte, gerado com a ajuda do Claude para fins acadêmicos. Ele implementa um sistema simples de cadastro e gerenciamento de alunos (cadastro, busca, remoção, atualização de notas e geração de relatórios), e o código completo está nas Figuras 1 a 4. O código passou por duas avaliações. A primeira foi uma análise manual, feita por um avaliador com conhecimento intermediário em Java, que considerou legibilidade, organização do código, possíveis falhas de implementação, tratamento de exceções e boas práticas de desenvolvimento. A segunda foi uma análise automatizada com o SonarQube, executado pela linha de comando, voltada a vulnerabilidades, *bugs*, duplicações de código e problemas de manutenção.

A Figura 1 mostra a classe Aluno, que guarda os dados de cada estudante cadastrado: identificador, nome, e-mail, as três notas usadas no cálculo da média final e o status de aprovação. Todos os atributos foram declarados como públicos, o que quebra o princípio do encapsulamento, recomendado na literatura de Engenharia de *Software*. Essa foi uma das falhas depois identificadas tanto pela análise manual quanto pela automatizada.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 1. Código-fonte da classe Aluno.java

```
package br.uniara.poc;

public class Aluno {
    public int id;
    public String nome;
    public String email;
    public double nota1;
    public double nota2;
    public double nota3;
    public String status;

    public Aluno(int id, String nome, String email, double nota1, double nota2, double
nota3) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.nota1 = nota1;
        this.nota2 = nota2;
        this.nota3 = nota3;
        this.status = "PENDENTE";
    }

    public double calcularMedia() {
        return (nota1 + nota2 + nota3) / 3;
    }
}
```

Fonte: dados da pesquisa (2026).

A Figura 2 mostra a classe AlunoRepositorio, que armazena e gerencia a coleção de alunos cadastrados, com operações de inclusão, busca, remoção e atualização de notas. Ela concentra parte das falhas estruturais do sistema: usa ao mesmo tempo um vetor de tamanho fixo e uma lista dinâmica para guardar os mesmos dados e não verifica valores nulos no método de atualização de notas, o que pode causar exceção em tempo de execução se for informado um identificador que não existe.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 2. Código-fonte da classe AlunoRepositorio.java

```
package br.uniara.poc;

import java.util.ArrayList;

public class AlunoRepositorio {

    private Aluno[] alunos = new Aluno[100];
    private int total = 0;

    private ArrayList<Aluno> historico = new ArrayList();

    public void adicionar(Aluno aluno) {
        alunos[total] = aluno;
        total = total + 1;
        historico.add(aluno);
    }

    public Aluno buscarPorId(int id) {
        for (int i = 0; i < total; i++) {
            if (alunos[i].id == id) {
                return alunos[i];
            }
        }
        return null;
    }

    public void removerPorId(int id) {
        for (int i = 0; i < total; i++) {
            if (alunos[i].id == id) {
                for (int j = i; j < total - 1; j++) {
                    alunos[j] = alunos[j + 1];
                }
                total--;
                return;
            }
        }
    }

    public Aluno[] listarTodos() {
        return alunos;
    }

    public int getTotal() {
        return total;
    }

    public void atualizarNotas(int id, double n1, double n2, double n3) {
        Aluno a = buscarPorId(id);
        if (a.id == id) {
            a.nota1 = n1;
            a.nota2 = n2;
            a.nota3 = n3;
        }
    }
}
```

Fonte: dados da pesquisa (2026).

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

A Figura 3 mostra a classe RelatorioService, que calcula a média de cada aluno, define sua situação (aprovado, em recuperação ou reprovado) e gera relatórios completos e resumidos. Nela há uma credencial de acesso escrita direto no código-fonte (*SENHA_ADMIN*) e uma comparação de strings feita com o operador "==" em vez do método "*equals()*", prática que a literatura reconhece como fonte comum de erros em aplicações Java.

Figura 3. Código-fonte da classe RelatorioService.java

```
package br.uniara.poc;

public class RelatorioService {

    public static final String SENHA_ADMIN = "admin123";

    public String gerarRelatorioCompleto(Aluno[] alunos, int total) {
        String relatorio = "";
        for (int i = 0; i < total; i++) {
            Aluno a = alunos[i];
            double media = (a.nota1 + a.nota2 + a.nota3) / 3;
            if (media >= 6.0) {
                a.status = "APROVADO";
            } else if (media >= 4.0) {
                a.status = "RECUPERACAO";
            } else {
                a.status = "REPROVADO";
            }
            relatorio = relatorio + "Aluno: " + a.nome + " | Email: " + a.email
                    + " | Nota1: " + a.nota1 + " | Nota2: " + a.nota2 + " | Nota3: " +
a.nota3
                    + " | Media: " + media + " | Status: " + a.status + "\n";
        }
        return relatorio;
    }

    public String gerarRelatorioResumido(Aluno[] alunos, int total) {
        String relatorio = "";
        for (int i = 0; i < total; i++) {
            Aluno a = alunos[i];
            double media = (a.nota1 + a.nota2 + a.nota3) / 3;
            if (media >= 6.0) {
                a.status = "APROVADO";
            } else if (media >= 4.0) {
                a.status = "RECUPERACAO";
            } else {
                a.status = "REPROVADO";
            }
            relatorio = relatorio + a.nome + " - " + a.status + "\n";
        }
        return relatorio;
    }

    public boolean autenticarAdmin(String senhaDigitada) {
        if (senhaDigitada == SENHA_ADMIN) {
            return true;
        }
        return false;
    }
}
```

Fonte: dados da pesquisa (2026).

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

A Figura 4 mostra a classe Main, ponto de entrada do sistema, que exibe o menu de opções e encaminha as ações escolhidas para as classes de repositório e de relatório. Ela trata exceções de forma genérica com o método *printStackTrace* e tem o método *exportarParaArquivo*, no qual um recurso de escrita em arquivo (*FileWriter*) é aberto sem garantia de fechamento. Essa falha foi depois sinalizada pela análise automatizada do SonarQube.

Figura 4. Código-fonte da classe Main.java

```
package br.uniara.poc;

import java.util.Scanner;
import java.util.ArrayList;
import java.io.IOException;

public class Main {

    static AlunoRepositorio repositorio = new AlunoRepositorio();
    static RelatorioService relatorioService = new RelatorioService();
    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        int opcao = -1;
        while (opcao != 0) {
            System.out.println("1 - Cadastrar aluno");
            System.out.println("2 - Buscar aluno");
            System.out.println("3 - Remover aluno");
            System.out.println("4 - Atualizar notas");
            System.out.println("5 - Emitir relatorio completo");
            System.out.println("6 - Emitir relatorio resumido");
            System.out.println("0 - Sair");

            try {
                opcao = Integer.parseInt(scanner.nextLine());
            } catch (Exception e) {
                e.printStackTrace();
            }

            if (opcao == 1) {
                cadastrarAluno();
            } else if (opcao == 2) {
                buscarAluno();
            } else if (opcao == 3) {
                removerAluno();
            } else if (opcao == 4) {
                atualizarNotas();
            } else if (opcao == 5) {

System.out.println(relatorioService.gerarRelatorioCompleto(repositorio.listarTodos(),
repositorio.getTotal()));
            } else if (opcao == 6) {

System.out.println(relatorioService.gerarRelatorioResumido(repositorio.listarTodos(),
repositorio.getTotal()));
            }
        }
    }
```


Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

```
    static void cadastrarAluno() {
        System.out.println("Nome:");
        String nome = scanner.nextLine();
        System.out.println("Email:");
        String email = scanner.nextLine();

        if (nome == null || nome.length() == 0) {
            System.out.println("Nome invalido");
            return;
        }
        if (email == null || email.length() == 0 || email.indexOf("@") == -1) {
            System.out.println("Email invalido");
            return;
        }

        System.out.println("Nota 1:");
        double n1 = Double.parseDouble(scanner.nextLine());
        System.out.println("Nota 2:");
        double n2 = Double.parseDouble(scanner.nextLine());
        System.out.println("Nota 3:");
        double n3 = Double.parseDouble(scanner.nextLine());

        Aluno aluno = new Aluno(repositorio.getTotal() + 1, nome, email, n1, n2, n3);
        repositorio.adicionar(aluno);
        System.out.println("Aluno cadastrado com sucesso");
    }

    static void buscarAluno() {
        System.out.println("Informe o ID:");
        int id = Integer.parseInt(scanner.nextLine());
        Aluno a = repositorio.buscarPorId(id);
        if (a != null) {
            System.out.println(a.nome + " - " + a.email + " - Media: " +
a.calcularMedia());
        } else {
            System.out.println("Aluno nao encontrado");
        }
    }

    static void removerAluno() {
        System.out.println("Informe o ID para remover:");
        int id = Integer.parseInt(scanner.nextLine());
        repositorio.removerPorId(id);
        System.out.println("Removido");
    }

    static void atualizarNotas() {
        System.out.println("Informe o ID:");
        int id = Integer.parseInt(scanner.nextLine());
        System.out.println("Nova nota 1:");
        double n1 = Double.parseDouble(scanner.nextLine());
        System.out.println("Nova nota 2:");
        double n2 = Double.parseDouble(scanner.nextLine());
        System.out.println("Nova nota 3:");
        double n3 = Double.parseDouble(scanner.nextLine());
        repositorio.atualizarNotas(id, n1, n2, n3);
    }

    static void exportarParaArquivo(String caminho) throws IOException {
        java.io.FileWriter writer = new java.io.FileWriter(caminho);

writer.write(relatorioService.gerarRelatorioCompleto(repositorio.listarTodos(),
repositorio.getTotal()));
```


Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

```
        }
    }
```

Fonte: dados da pesquisa (2026).

O avaliador humano revisou o código-fonte sem nenhuma ferramenta automatizada, anotando cada problema encontrado e cronometrando o tempo total da revisão. Depois que o código foi gerado, o Claude executou a análise automatizada com o SonarQube pelo ambiente de linha de comando, e os resultados da ferramenta (*bugs*, vulnerabilidades, *code smells*, duplicação de código e tempo de execução) serviram de referência para a avaliação automatizada.

Os resultados das duas abordagens foram comparados principalmente pelo tipo e pela natureza dos problemas identificados (*bugs*, vulnerabilidades, manutenibilidade e duplicação) e pela sobreposição entre os conjuntos encontrados. O tempo de análise foi registrado como métrica secundária. Depois da coleta, os dados foram organizados e analisados de forma quantitativa descritiva e qualitativa, para identificar as contribuições e as limitações de cada abordagem e de seu uso conjunto na avaliação da qualidade de *software*.

Para favorecer a reprodução, registram-se os parâmetros do experimento. A análise automatizada usou o SonarQube Community Build 26.8.0 com JDK 21, perfil de regras padrão da ferramenta e arquivo de configuração com chave de projeto “poc-cadastro-alunos”, fontes em “src”, binários em “bin” e codificação UTF-8. O avaliador humano recebeu apenas o código-fonte e um roteiro de registro, com a orientação de não usar ferramentas automatizadas, anotar cada problema encontrado, com o arquivo e a linha quando possível, e cronometrar o tempo total. O código completo está nas Figuras 1 a 4.

O SonarQube e o avaliador humano usam classificações próprias, que não são totalmente equivalentes. Por isso, os defeitos do código foram agrupados em uma taxonomia comum com três grupos: (G1) problemas detectáveis por análise estática, como comparação de strings com “==”, recurso não fechado, código duplicado, uso de *printStackTrace* e campos públicos; (G2) problemas de lógica e de contexto, em que o código é válido mas se comporta de modo diferente do esperado; e (G3) problemas de segurança, como credenciais escritas no código e validação insuficiente. Cada defeito foi atribuído a um único grupo conforme a informação necessária para reconhecê-lo: G1 quando basta o padrão do código, sem conhecer a intenção do programa; G2 quando é preciso saber o comportamento esperado ou a regra de negócio; e G3 quando o problema afeta a confidencialidade, o controle de acesso ou a validação de entradas. O Quadro 1 traz o catálogo de defeitos conhecidos do código-fonte, elaborado a partir da leitura do código depois de ele ser gerado. Ele, portanto, não é um gabarito definido antes do experimento, e essa limitação é discutida na seção 4.4. Cada defeito do catálogo foi

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

depois confrontado com os 16 itens do avaliador humano e com as 35 ocorrências exportadas do SonarQube. O defeito contou como identificado quando um item ou ocorrência apontava o mesmo trecho de código e o mesmo problema. A classificação e o mapeamento foram feitos pela primeira autora, sem verificação independente, o que também é registrado como limitação.

Quadro 1. Catálogo de defeitos conhecidos do código-fonte

ID	Grupo	Defeito conhecido	Local no código
D01	G1	Atributos públicos (quebra de encapsulamento)	Aluno
D02	G1	Comparação de strings com "==" em vez de equals()	RelatorioService.autenticarAdmin
D03	G1	Recurso FileWriter aberto e nunca fechado	Main.exportarParaArquivo
D04	G1	Lógica de cálculo da média e do status duplicada	RelatorioService (dois métodos de relatório) e Aluno.calcularMedia
D05	G1	Uso de printStackTrace no tratamento de exceção	Main.main
D06	G1	Concatenação de strings dentro de laço	RelatorioService
D07	G1	Uso de tipo genérico bruto (new ArrayList())	AlunoRepositorio
D08	G1	Importação não utilizada (java.util.ArrayList)	Main
D09	G1	Uso de System.out em vez de mecanismo de log	Main
D10	G2	Desreferência nula: buscarPorId pode retornar null e atualizarNotas não verifica	AlunoRepositorio.atualizarNotas
D11	G2	Identificador gerado por getTotal() + 1 pode repetir valores após remoções	Main.cadastrarAluno
D12	G2	Capacidade fixa de 100 alunos sem verificação de limite; vetor e lista armazenam os mesmos dados, e a lista não é atualizada na remoção	AlunoRepositorio
D13	G2	Falha na leitura do menu mantém a opção anterior e reexecuta a ação	Main.main
D14	G2	Entradas numéricas sem tratamento de erro (NumberFormatException)	Main (cadastro e atualização)
D15	G2	Notas sem validação de faixa (aceita valores negativos ou acima de 10)	Main / Aluno
D16	G2	Mensagem "Removido" exibida mesmo quando o identificador não existe	Main.removerAluno
D17	G2	Método listarTodos expõe a estrutura interna do repositório (vetor com posições nulas)	AlunoRepositorio
D18	G2	Geração de relatório altera o status do aluno (efeito colateral)	RelatorioService
D19	G3	Credencial de administrador fixada no código (SENHA_ADMIN)	RelatorioService
D20	G3	Método de autenticação nunca é chamado: operações sem controle de acesso	RelatorioService / Main
D21	G3	Validação de e-mail limitada à presença de "@"	Main.cadastrarAluno
D22	G3	Caminho de arquivo recebido sem validação na exportação	Main.exportarParaArquivo

Fonte: elaborado pelos autores a partir da leitura do código-fonte da prova de conceito (2026).

4. RESULTADOS E DISCUSSÃO

4.1. Aplicação da prova de conceito

O código-fonte escolhido foi primeiro submetido a uma análise manual, feita por um avaliador com conhecimento intermediário em Java. Nessa etapa foram avaliados legibilidade,

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

organização do código, possíveis falhas de implementação, tratamento de exceções e conformidade com boas práticas de desenvolvimento.

Em seguida, o mesmo código foi analisado com o SonarQube, que gerou relatórios com métricas de qualidade, vulnerabilidades, *bugs* e problemas de manutenção.

Neste experimento, a análise automatizada com o SonarQube foi mais rápida que a inspeção manual: o avaliador humano levou 29 minutos e 55 segundos para concluir a revisão, e o SonarQube processou o mesmo código em cerca de 13,4 segundos, uma redução de aproximadamente 99,3%. Como essa diferença já era esperada entre uma ferramenta e a leitura humana, o tempo é tratado como métrica secundária e não como principal evidência do estudo. O SonarQube também registrou bem mais ocorrências no total, principalmente de manutenibilidade (*code smells*), o que mostra a capacidade da análise estática de varrer o código de forma sistemática e rápida. Já a análise manual registrou mais problemas classificados como *bugs* (falhas que afetam o funcionamento do código). Isso sugere que, neste código, a leitura humana reconheceu melhor certas falhas lógicas e estruturais, embora tenha catalogado menos violações de convenções de estilo. Esses achados conversam com as limitações dos processos tradicionais de garantia da qualidade discutidas por Harman, Mansouri e Zhang (2012) e indicam que a automação, sozinha, não substitui a capacidade de interpretação do avaliador humano.

4.2. Resultados obtidos

Na análise manual, feita por um avaliador com conhecimento intermediário em Java, foram identificados 16 problemas de qualidade, dos quais o próprio avaliador classificou 6 como *bugs*, 1 como falha de segurança, 2 como duplicação de código e 7 como violações de boas práticas e de organização. Entre os problemas apontados estão a quebra de encapsulamento por atributos públicos, a falta de verificação de valores nulos antes de acessos que podem lançar exceções, a falta de tratamento de exceções para entradas inválidas do usuário, uma senha escrita direto no código-fonte, o cálculo duplicado da média do aluno em dois métodos, o uso ineficiente de concatenação de strings em laço e a comparação incorreta de strings com o operador “==” em vez do método “*equals()*”.

A análise do SonarQube identificou 35 ocorrências, sendo 2 *bugs*, 1 vulnerabilidade e 32 *code smells*, que é a categoria da própria ferramenta para problemas de manutenibilidade (SONARSOURCE, 2024). Nos *bugs*, a ferramenta apontou um recurso não fechado (“*FileWriter*”) e a mesma comparação incorreta de strings com “==” que o avaliador tinha achado. Como vulnerabilidade, sinalizou o uso de uma instrução de depuração (“*printStackTrace*”) deixada no código. A maior parte dos *code smells* foi repetição do mesmo tipo de recomendação (por

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

exemplo, trocar chamadas de "*System.out*" por um mecanismo de *log*). Isso ajuda a explicar por que o total da ferramenta foi muito maior que o da análise manual, sem que isso signifique uma quantidade proporcional de falhas distintas e graves. Além disso, só 2 das 35 ocorrências foram classificadas como *bugs*, número menor que o da análise manual. Grande parte da vantagem numérica do SonarQube vem, portanto, de apontar desvios de convenção de estilo, e não de achar mais falhas funcionais graves.

A Tabela 1 apresenta uma comparação entre os resultados obtidos.

Tabela 1. Comparação entre análise humana e análise automatizada (SonarQube)

Critério avaliado	Análise humana	SonarQube
Tempo de análise (métrica secundária)	29 min e 55 s	≈ 13,4 segundos (redução de ≈ 99,3%)
Problemas identificados (total)	16	35
Bugs	6	2
Vulnerabilidades detectadas	1	1
Problemas de manutenção (code smells / boas práticas)	7	32
Duplicação de código	2	2 blocos

Fonte: dados da pesquisa (2026).

4.3. Discussão dos resultados

Para ir além da contagem de ocorrências, cada defeito do catálogo (Quadro 1) foi comparado com os itens do avaliador humano e com as ocorrências do SonarQube. Isso permitiu calcular a taxa de detecção por abordagem e por grupo da taxonomia. A Tabela 2 traz o resumo e o Quadro 2 mostra o mapeamento defeito a defeito. O SonarQube identificou 7 dos 22 defeitos (31,8%) e o avaliador humano, 10 (45,5%). Apenas 4 foram identificados pelas duas abordagens, 6 somente pelo humano, 3 somente pelo SonarQube e 9 por nenhuma delas. Nos defeitos detectáveis por análise estática (G1), o SonarQube teve a maior taxa (7 de 9, ou 77,8%, contra 66,7% do humano) e achou itens que o humano não apontou, como o recurso não fechado, o uso de *printStackTrace* e o uso de *System.out*. Nos defeitos de lógica e contexto (G2) e de segurança (G3), o SonarQube não identificou nenhum dos 13, e o humano identificou 4 (3 de G2 e 1 de G3). Ainda assim, 9 dos 13 defeitos desses dois grupos não foram identificados por nenhuma abordagem, como a geração de identificadores repetidos depois de remoções, a falta de controle de acesso e a validação insuficiente de entradas.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Tabela 2. Defeitos do catálogo identificados por abordagem e grupo

Grupo	Defeitos	Humano	SonarQube	Ambos	Nenhum
G1 - Análise estática	9	6 (66,7%)	7 (77,8%)	4	0
G2 - Lógica e contexto	9	3 (33,3%)	0 (0,0%)	0	6
G3 - Segurança	4	1 (25,0%)	0 (0,0%)	0	3
Total	22	10 (45,5%)	7 (31,8%)	4	9

Fonte: dados da pesquisa (2026), a partir do mapeamento entre o Quadro 1, a análise manual e o relatório do SonarQube.

Além dos defeitos do catálogo, o avaliador humano apontou 2 itens que não correspondem a defeitos catalogados (o status inicial "PENDENTE" e o status "RECUPERACAO" não declarado na classe, ambos sem efeito no funcionamento). O SonarQube registrou 3 ocorrências fora do catálogo, de estilo (duas sugestões de uso de *isEmpty()* e uma de simplificação de retorno). Das 35 ocorrências do SonarQube, 26 repetem uma única recomendação (usar *log* em vez de *System.out*), o que mostra que o total de ocorrências não mede a quantidade de defeitos distintos. Da mesma forma, 5 dos 16 itens do avaliador tratam do mesmo defeito (atributos públicos).

Quadro 2. Mapeamento dos defeitos do catálogo por abordagem de análise

ID	Avaliador humano	SonarQube (regra)	Identificado por
D01	Itens 2, 3, 6, 7 e 10	-	Só humano
D02	Item 16	S4973	Ambos
D03	-	S2095	Só SonarQube
D04	Item 13	Duplicação (2 blocos)	Ambos
D05	-	S4507	Só SonarQube
D06	Item 15	S1643 (2x)	Ambos
D07	Item 1	-	Só humano
D08	Item 11	S1128	Ambos
D09	-	S106 (26x)	Só SonarQube
D10	Item 5	-	Só humano
D11	-	-	Nenhum
D12	-	-	Nenhum
D13	-	-	Nenhum
D14	Item 9	-	Só humano
D15	-	-	Nenhum
D16	-	-	Nenhum
D17	Item 4	-	Só humano
D18	-	-	Nenhum
D19	Item 12	-	Só humano
D20	-	-	Nenhum
D21	-	-	Nenhum
D22	-	-	Nenhum

Fonte: dados da pesquisa (2026).

A análise automatizada teve vantagens em velocidade, em padronização dos critérios e em apontar de forma sistemática um grande volume de violações de boas práticas. Um ponto interessante é que cada abordagem identificou exatamente uma vulnerabilidade de segurança, mas não a mesma: o avaliador humano reconheceu uma credencial escrita no código-fonte, e o SonarQube sinalizou uma instrução de depuração deixada no código. Isso indica que ferramenta

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

e avaliador tendem a enxergar tipos diferentes de risco, e reforça que as duas abordagens se complementam em vez de uma substituir a outra.

A quantidade de ocorrências não equivale à qualidade da detecção. Das 35 ocorrências do SonarQube, 32 são *code smells*, muitos deles repetições da mesma recomendação, enquanto o avaliador humano classificou 6 itens como *bugs* e apontou problemas funcionais que a ferramenta não sinalizou. Por isso, a taxa de detecção sobre o catálogo (Tabela 2) informa mais que a contagem bruta, mas não deve ser lida como prova de que uma abordagem é melhor que a outra: o catálogo foi elaborado depois da geração do código, o mapeamento foi feito por uma única pessoa e o estudo teve um único avaliador. Também não foi possível avaliar falsos positivos no sentido estrito, pois todas as ocorrências do SonarQube correspondem a recomendações válidas, ainda que de baixa severidade.

A análise humana também trouxe contribuições relevantes. Na inspeção manual foram identificadas oportunidades de melhoria na lógica de negócio e na organização funcional do sistema, que o SonarQube não apontou diretamente.

Os 13 defeitos de lógica, contexto e segurança que o SonarQube não identificou ilustram uma limitação conhecida da análise estática baseada em regras predefinidas: ela examina padrões no código e, por isso, tende a não reconhecer defeitos que dependem da intenção do desenvolvedor, da regra de negócio ou de fluxos de dados e de controle de acesso que atravessam várias partes do sistema. Os resultados deste estudo apenas ilustram essa limitação em um sistema pequeno, com a ferramenta configurada com suas regras padrão, e não permitem estimar sua extensão em sistemas corporativos maiores. Em ambientes reais, usar a análise automatizada isoladamente pode gerar uma falsa sensação de segurança, o que reforça a necessidade da revisão humana.

Neste cenário, portanto, a análise automatizada deve ser vista como complemento da avaliação humana e não como substituta. Combinar conhecimento humano e ferramentas automatizadas se mostrou mais adequado para chegar a avaliações completas. Na prática, o SonarQube foi mais útil como triagem inicial, pois sinalizou rapidamente muitos desvios de boas práticas, e coube ao avaliador humano interpretar esses achados conforme o contexto do sistema e priorizar as correções mais importantes para a qualidade final do produto. Como o SonarQube trabalha com regras predefinidas, este estudo também não permite concluir nada sobre o desempenho de modelos de IA generativa na detecção de defeitos, o que fica como sugestão para trabalhos futuros.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

4.4. Considerações sobre a prova de conceito

A prova de conceito permitiu observar, em um cenário controlado, os benefícios e as limitações de cada abordagem na avaliação da qualidade de *software*. Os resultados sugerem que a análise automatizada pode reduzir o tempo de análise e ampliar a cobertura das inspeções, apontando problemas que passariam despercebidos em uma inspeção só manual, enquanto a análise humana contribuiu com problemas de lógica e de contexto.

O estudo tem limitações importantes. Foi feito em um único sistema de pequeno porte, com apenas um avaliador humano, e usou um catálogo de defeitos elaborado depois da geração do código (Quadro 1) e não definido antes do experimento, o que pode favorecer a identificação dos defeitos que os próprios autores reconheceram. Além disso, a classificação e o mapeamento dos defeitos foram feitos por uma única pessoa, a classificação dos problemas pelo avaliador humano e pelo SonarQube não é totalmente equivalente, e a revisão da literatura foi narrativa. Esses fatores limitam a generalização dos resultados para projetos mais complexos ou para outros perfis de avaliadores, então os resultados devem ser lidos como indícios e não como conclusões definitivas. Pesquisas futuras poderiam (i) definir antes da análise um conjunto de defeitos conhecidos, organizado em grupos de análise estática, de lógica e de segurança, para calcular taxa de detecção, precisão e falsos positivos sem depender de um catálogo posterior; (ii) incluir um modelo de linguagem (LLM) como terceira abordagem, repetindo a análise várias vezes para medir a consistência dos resultados em comparação com o SonarQube, que tende a ser determinístico; (iii) usar códigos de tamanhos e complexidades diferentes, para ver como cada abordagem se comporta quando o código cresce; e (iv) ampliar o número de avaliadores humanos.

5. CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo comparar a capacidade de identificar problemas de qualidade de *software* de uma análise humana e de uma análise automatizada com o SonarQube, por meio de uma revisão bibliográfica e de uma prova de conceito. Os resultados permitem responder ao problema de pesquisa: neste cenário, as duas abordagens identificaram conjuntos de problemas diferentes e só em parte coincidentes. A análise automatizada foi mais rápida e sistemática nos problemas de manutenibilidade e de convenções de código, enquanto a análise humana registrou mais *bugs* e problemas de lógica e de contexto, e cada abordagem apontou uma vulnerabilidade diferente. No catálogo de 22 defeitos, o avaliador humano

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

identificou 10 e o SonarQube, 7, com apenas 4 em comum, e 9 defeitos não foram identificados por nenhuma das abordagens.

A hipótese inicial foi corroborada de forma exploratória, nos limites do experimento. A ferramenta registrou mais ocorrências em menos tempo, mas isso se deveu sobretudo a *code smells*, e a análise humana continuou indispensável para interpretar aspectos de lógica de negócio e de organização funcional que a ferramenta não sinalizou. Assim, análise humana e análise automatizada devem ser tratadas como abordagens complementares e não substitutas.

As principais contribuições deste trabalho são reunir evidências empíricas, ainda que em escala reduzida, sobre as capacidades diferentes da análise humana e da automatizada, sistematizar a literatura recente sobre o tema e diferenciar a Inteligência Artificial da análise estática baseada em regras. As limitações (um único sistema, um único avaliador e um catálogo de defeitos elaborado depois da geração do código) e as sugestões de trabalhos futuros estão detalhadas na seção 4.4.

DECLARAÇÃO SOBRE O USO DE INTELIGÊNCIA ARTIFICIAL

Em atendimento às normas da revista, declara-se que o assistente de Inteligência Artificial Claude (Anthropic) foi usado para gerar o código-fonte Java analisado na prova de conceito, executar os comandos de análise automatizada e apoiar a edição, a formatação e a revisão linguística do manuscrito. A análise humana foi feita por um avaliador, e os autores continuam responsáveis pela interpretação dos resultados e pelo conteúdo final do artigo.

REFERÊNCIAS

ALLAMANIS, M. et al. A Survey of Machine Learning for Big Code and Naturalness. **ACM Computing Surveys**, v. 51, n. 4, art. 81, 2018. DOI: 10.1145/3212695.

ANTHROPIC. **Claude Code**. [S. l.]: Anthropic, 2026. Disponível em: https://docs.claude.com/en/docs/claude-code/overview. Acesso em: 13 set. 2026.

BATARSEH, F. A. et al. The application of artificial intelligence in *software* engineering: a review challenging conventional wisdom. In: BATARSEH, F. A.; YANG, R. (ed.). **Data Democracy: At the Nexus of Artificial Intelligence, *Software* Development, and Knowledge Engineering**. [S. l.]: Elsevier, 2020. cap. 10, p. 179-232. DOI: 10.1016/B978-0-12-818366-3.00010-1.

CHEN, M. et al. **Evaluating Large Language Models Trained on Code**. arXiv preprint arXiv:2107.03374, 2021. Disponível em: https://arxiv.org/abs/2107.03374. Acesso em: 27 out. 2026.

HALL, T. et al. A Systematic Literature Review on Fault Prediction Performance in *Software* Engineering. **IEEE Transactions on *Software* Engineering**, v. 38, n. 6, p. 1276-1304, 2012. DOI: 10.1109/TSE.2011.103.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-Based *Software* Engineering: Trends, Techniques and Applications. **ACM Computing Surveys**, v. 45, n. 1, art. 11, 2012. DOI: 10.1145/2379776.2379787.

HELLENDOORN, V. J.; DEVANBU, P. Are Deep Neural Networks the Best Choice for Modeling Source Code? In: **Proceedings of the 2017 11th Joint Meeting on Foundations of *Software* Engineering (ESEC/FSE 2017)**. Paderborn, 2017. p. 763-773. DOI: 10.1145/3106237.3106290.

ISO/IEC. **ISO/IEC 25010:2023: Systems and *software* engineering — Systems and *software* Quality Requirements and Evaluation (SQuaRE) — Product quality model**. Geneva: ISO, 2023.

MALHOTRA, R. A Systematic Review of Machine Learning Techniques for *Software* Fault Prediction. **Applied Soft Computing**, v. 27, p. 504-518, 2015. DOI: 10.1016/j.asoc.2014.11.023.

MENZIES, T.; ZIMMERMANN, T. *Software* Analytics: So What? **IEEE *Software***, v. 30, n. 4, p. 31-37, 2013. DOI: 10.1109/MS.2013.86.

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

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: a modern approach**. 4. ed. Hoboken: Pearson, 2021.

SOMMERVILLE, I. **Engenharia de *software***. 10. ed. São Paulo: Pearson, 2019.

SONARSOURCE. **SonarQube documentation**. 2024. Disponível em: https://docs.sonarsource.com/sonarqube/latest/. Acesso em: 29 ago. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.