Avaliação de Qualidade de Código em Projetos Open Source

Anais do Computer on the Beach

Guilherme Pereira
Universidade de Pernambuco
Garanhuns, PE, Brasil

Matheus Albuquerque
Universidade de Pernambuco
Garanhuns, PE, Brasil

Victor Ferreira
Universidade de Pernambuco
Garanhuns, PE, Brasil

Gabriel Silva
Universidade de Pernambuco
Garanhuns, PE, Brasil

Jonas Santos
Universidade de Pernambuco
Garanhuns, PE, Brasil

Pedro Rito
Universidade de Pernambuco
Garanhuns, PE, Brasil

Abstract

This study presented an analysis of code quality in open-source projects, an essential topic for understanding system maintainability and evolution. It consisted of an applied research study, with a mixed-methods approach, of a descriptive and exploratory nature, whose objective was to identify the main quality issues present in publicly accessible software repositories. A total of 122 projects developed in Java and Python were examined, using static analysis tools such as SonarQube, PMD, and Prospector, considering metrics such as code smells, defects, code duplication, technical debt, and automated quality assessments. The results revealed relevant differences between the ecosystems, with Java presenting greater internal variability and a higher average density of issues, while Python demonstrated lower averages but greater asymmetry in metrics such as technical debt and reliability. The analysis revealed a strong relationship between code smells, defects, and technical debt, indicating that the accumulation of inadequate practices increases maintenance effort, whereas external factors, such as project size and number of contributors, had limited impact. It was concluded that internal code quality depends mainly on the practices adopted by developers and that static analysis constitutes a fundamental resource to support maintenance, standardization, and system evolution in collaborative environments.

Keywords

Qualidade de código, Software de código aberto, Análise estática, Métricas de software, Manutenibilidade.

1 Introdução

A qualidade de software é um fator central para garantir confiabilidade, manutenibilidade e evolução de sistemas ao longo do ciclo de vida. O padrão ISO/IEC 25010 [1] define um modelo abrangente de qualidade, abrangendo atributos como eficiência de desempenho, segurança e manutenibilidade, que continuam sendo referência em pesquisas atuais [2, 3].

A qualidade do código desempenha papel decisivo nesse contexto, pois impacta diretamente o custo de manutenção, a ocorrência de falhas e a facilidade de evolução de sistemas [4, 5]. Mais do que um aspecto estético, código de qualidade envolve legibilidade, baixo acoplamento, ausência de duplicações e aderência a boas práticas, fatores que aumentam a confiabilidade e reduzem riscos em ambientes dinâmicos e competitivos, aspectos amplamente discutidos por Fowler [6].

Projetos open source constituem um campo de estudo relevante na engenharia de software, dada sua ampla adoção e o modelo de desenvolvimento colaborativo e distribuído [7]. Essa estrutura descentralizada, sustentada por comunidades diversas e globais, impulsiona a inovação e o aprendizado contínuo [8]. Entretanto, a variedade de contribuidores com diferentes níveis de experiência introduz desafios relacionados à manutenção da qualidade e à padronização do código [8, 9].

A ausência de processos formais de engenharia, como documentação e controle de mudanças, pode gerar inconsistências e dificultar a rastreabilidade, aumentando o débito técnico [10]. Além disso, estudos indicam que a sustentabilidade comunitária nem sempre se correlaciona com a qualidade técnica: o crescimento acelerado da base de código, sem governança adequada, tende a elevar a complexidade e a duplicação [11].

Projetos open source frequentemente apresentam oscilações na manutenibilidade, influenciadas por fatores como complexidade ciclomática e tamanho do código [8]. A presença de *code smells*, ainda que moderada, está associada ao aumento de defeitos e riscos de manutenção, evidenciando a importância da padronização e do controle de qualidade [9]. Esses aspectos reforçam a necessidade de avaliar sistematicamente a qualidade de código para mitigar problemas que comprometem a evolução do software.

Diante desse cenário, este trabalho propõe uma análise empírica da qualidade de código em projetos open source, buscando compreender seus impactos sobre a manutenção e evolução do software. O estudo investiga 122 repositórios públicos desenvolvidos em Java e Python, selecionados de forma sistemática, e baseia-se em métricas de qualidade de código. Para a coleta e avaliação dessas métricas, são empregadas ferramentas de análise estática, incluindo SonarQube, PMD e Prospector, permitindo uma comparação estruturada entre os ecossistemas analisados.

Assim, busca-se responder à seguinte questão de pesquisa: *Quais são os problemas de qualidade de código mais recorrentes em projetos open source, e de que forma a identificação desses problemas contribui para compreender os desafios de manutenção e padronização do código em ambientes colaborativos?*

A realização deste estudo justifica-se pela crescente relevância dos projetos *open source* no ecossistema de desenvolvimento de software e pela necessidade de compreender como práticas e decisões técnicas influenciam a qualidade do código desses projetos, especialmente em ambientes colaborativos e distribuídos [8, 11]. Avaliar métricas objetivas de qualidade permite identificar padrões e fragilidades em projetos colaborativos, oferecendo subsídios para

a melhoria de processos de desenvolvimento, manutenção e governança de código em comunidades de software livre.

1.1 Objetivo Geral

Identificar e analisar os problemas de qualidade de código em projetos *open source*, com base nas métricas e resultados obtidos por ferramentas de análise estática.

1.2 Objetivos Específicos

- Identificar métricas relevantes para avaliação de qualidade de código;
- Selecionar e configurar ferramentas de análise adequadas;
- Aplicar as ferramentas em um conjunto ampliado de projetos *open source*;
- Coletar, organizar e analisar os dados obtidos;
- Investigar correlações entre métricas internas de qualidade de código e atributos externos dos projetos.

2 Fundamentação Teórica

A qualidade de software é fundamental para garantir confiabilidade e manutenibilidade. No modelo ISO/IEC 25010, a dimensão de manutenibilidade é particularmente relevante para este estudo, pois estabelece a relação entre características internas do código e seus impactos diretos na manutenção e evolução dos sistemas [12]. Na prática, a qualidade de código reflete esses aspectos, já que complexidade e duplicação impactam diretamente a manutenção e a estabilidade do software [13].

2.1 Qualidade de código e métricas

Qualidade de código refere-se ao grau em que o código-fonte favorece sua compreensão, manutenção e evolução ao longo do tempo, refletindo atributos internos como organização, clareza e simplicidade [14]. A qualidade de código reúne características estruturais e semânticas que influenciam diretamente a manutenção. Código legível, modular e com baixo acoplamento tende a ser mais confiável e fácil de evoluir, enquanto a ausência desses atributos favorece o acúmulo de débito técnico, elevando o esforço de manutenção e o risco de falhas [15, 16].

Métricas de qualidade oferecem uma forma objetiva de avaliar e monitorar o estado do código, permitindo identificar pontos críticos e orientar melhorias por meio da mensuração de propriedades como complexidade, acoplamento, coesão, duplicação e tamanho, diretamente relacionadas à manutenibilidade [17, 18]. Consideradas em conjunto, essas métricas auxiliam na detecção de débitos técnicos e na antecipação de riscos de deterioração do código [19].

Como indicadores operacionais, as métricas traduzem propriedades técnicas do código em informações úteis para a manutenção e evolução do software. Tendências de aumento na complexidade ou duplicação indicam maior custo de manutenção, enquanto baixa cobertura de testes e a presença de *code smells* estão associadas ao aumento de defeitos, reforçando a importância da análise combinada dos indicadores [9, 11, 20].

2.2 Ferramentas de análise de código

Ferramentas de análise de código, especialmente as de análise estática, inspecionam o código fonte sem executá-lo para identificar violações, *code smells* e indícios de defeitos, convertendo propriedades estruturais em evidências quantitativas de qualidade. Estudos comparativos demonstram a utilidade de ferramentas amplamente utilizadas, como SonarQube, PMD e Checkstyle, na detecção de problemas em múltiplas linguagens [21]. Essas medições ajudam a operacionalizar dimensões do modelo ISO/IEC 25010, com destaque para manutenibilidade e confiabilidade. Hashmat et al. [22] evidenciam, em estudo de larga escala, que a precisão dos alertas e a cobertura variam significativamente entre as soluções analisadas.

As abordagens de avaliação de qualidade de código diferenciam-se pelo tipo de análise adotado. A análise estática examina o código sem executá-lo, enquanto a análise dinâmica observa o comportamento do software em execução para detectar falhas [23]. Entre as técnicas estáticas, destacam-se aquelas baseadas em regras sintáticas e padrões de estilo, comuns em *linters* [24], e as fundamentadas em métricas estruturais e de complexidade, associadas à manutenibilidade [25]. Abordagens híbridas combinam múltiplas dimensões para ampliar a precisão dos diagnósticos [26], refletindo diferentes concepções teóricas de qualidade de código [14].

Essas ferramentas utilizam mecanismos automatizados, como *parsers* e árvores sintáticas abstratas (ASTs), para extrair métricas como complexidade, acoplamento, coesão, duplicação e cobertura de testes, associadas a atributos de manutenibilidade e confiabilidade [27]. Esses mecanismos permitem identificar *code smells* e possíveis débitos técnicos, tornando mensuráveis aspectos internos da qualidade do código [28].

Do ponto de vista teórico, as ferramentas de análise estática ligam modelos abstratos de qualidade à sua mensuração prática [29]. Ainda assim, apresentam limitações, como falsos positivos, foco predominantemente estrutural e dificuldade de captar nuances contextuais [30, 31]. Por esse motivo, a literatura recomenda combinar análises automatizadas com avaliações qualitativas para uma visão mais holística da qualidade do software [32, 33].

2.3 Trabalhos Relacionados

Estudos recentes têm investigado como a qualidade de código em projetos *open source* pode ser mensurada e associada à manutenibilidade e à evolução de software, buscando compreender como métricas e análises automatizadas revelam a relação entre qualidade interna, facilidade de manutenção e dinâmicas colaborativas do desenvolvimento aberto.

Jin et al. [13] analisaram mais de 36 mil projetos *open source* utilizando métricas do SonarQube e do conjunto CK, demonstrando que complexidade, duplicação e acoplamento influenciam diretamente a qualidade e a adoção dos projetos. De forma complementar, Iftikhar et al. [34] sintetizaram evidências de quinze revisões sistemáticas, apontando linhas de código, acoplamento e complexidade como indicadores consistentes de manutenibilidade e confiabilidade. Em conjunto, esses estudos reforçam o papel das métricas estruturais na compreensão dos impactos da qualidade interna sobre a manutenção e evolução do software.

A eficácia e as limitações das ferramentas de análise de código também têm sido exploradas na literatura. Hashmat et al. [22]

aplicaram 24 ferramentas de análise estática em milhares de repositórios, evidenciando diferenças na cobertura e consistência dos diagnósticos. Yeboah e Popoola [21] compararam SonarQube, PMD e Checkstyle em diferentes linguagens, observando variações na precisão conforme o tipo de defeito. Esses resultados reforçam a necessidade de avaliar cuidadosamente a combinação de ferramentas.

Apesar dos avanços, persistem lacunas, como a limitação a poucas linguagens, a ausência de análises longitudinais e a comparação restrita entre ferramentas. Diante disso, este trabalho propõe uma análise empírica que combina múltiplas ferramentas de análise estática em repositórios *open source* de diferentes linguagens, relacionando métricas de qualidade a indicadores de manutenção e evolução.

3 Metodologia

A Figura 1 apresenta o desenho metodológico adotado no estudo, sintetizando as etapas executadas desde a definição dos objetivos até a análise e discussão dos resultados. O fluxo resume o processo de seleção dos projetos, configuração das ferramentas de análise estática, coleta e normalização das métricas, bem como os procedimentos estatísticos aplicados para comparar a qualidade de código entre os ecossistemas Java e Python.

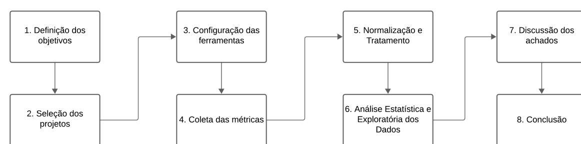


Figure 1: Percurso Metodológico.

3.1 Tipo de pesquisa e abordagem

Este estudo é de natureza aplicada e empírica, pois utiliza métricas e ferramentas de análise estática para investigar problemas concretos de qualidade de código em projetos *open source* de grande uso prático. Segundo Gil (2008), a pesquisa aplicada "visa gerar conhecimentos para aplicação prática, dirigidos à solução de problemas específicos".

Quanto aos objetivos, caracteriza-se como descritiva e exploratória. A pesquisa exploratória tem como finalidade proporcionar maior familiaridade com o problema, tornando-o mais explícito, enquanto a descritiva busca observar, registrar, analisar e correlacionar fatos ou fenômenos sem manipulá-los [35]. Assim, pretende-se compreender como diferentes práticas de desenvolvimento afetam a qualidade do código e, consequentemente, a manutenção e evolução de software.

A abordagem é predominantemente quantitativa, baseada em métricas coletadas automaticamente. Complementarmente, adota-se uma interpretação qualitativa dos resultados à luz da literatura, discutindo implicações para manutenção e evolução.

O método adotado é o estudo comparativo, em que 122 projetos *open source* são analisados de forma sistemática com base nas mesmas métricas e ferramentas, possibilitando identificar padrões, diferenças e boas práticas que influenciam a qualidade do código.

3.2 Seleção dos projetos

A seleção dos projetos open source analisados foi realizada de forma sistemática na plataforma GitHub, utilizando consultas avançadas (*strings* de busca) que permitiram filtrar repositórios de acordo com critérios de relevância, tamanho e atividade. Esse processo reduz vieses de seleção, embora restrinja a análise a projetos hospedados no GitHub com licenças específicas e determinado nível de popularidade. O objetivo foi garantir que os projetos escolhidos fossem representativos, ativos e maduros o suficiente para permitir uma análise consistente de qualidade de código.

As buscas foram realizadas separadamente para as linguagens Java e Python, permitindo uma comparação entre ecossistemas distintos. As consultas incluíram filtros de linguagem, licença, popularidade e atividade recente, além da exclusão de repositórios não representativos (tutoriais, exemplos e demonstrações). Os critérios utilizados foram:

- **Linguagem:** Java ou Python;
- **Licença:** MIT, Apache-2.0 ou BSD-3-Clause;
- **Popularidade:** entre 300 e 1000 estrelas;
- **Participação da comunidade:** mais de 10 forks;
- **Tamanho:** entre 1 e 50 MB;
- **Atividade recente:** commits após abril de 2025;
- **Exclusões:** NOT tutorial, NOT example, NOT demo.

As *strings* completas utilizadas foram:

Para Java: `language:Java`
```
license:mit license:apache-2.0 license:bsd-3-clause
stars:300..1000 forks:>10 size:1000..50000
pushed:>2025-04-01 NOT tutorial NOT example NOT demo
```

Para Python: `language:Python`
```
license:mit license:apache-2.0 license:bsd-3-clause
stars:300..1000 forks:>10 size:1000..50000
pushed:>2025-04-01 NOT tutorial NOT example NOT demo
```

A execução dessas consultas retornou 316 repositórios Java e aproximadamente 1200 repositórios Python. O cálculo amostral considerou a população de repositórios retornados para cada linguagem e teve como objetivo estimar, com 90% de confiança e margem de erro de 10%, a proporção de projetos que apresentam determinados níveis de problemas de qualidade, adotando-se $p = 0{,}5$ para maximizar a variabilidade. Para isso, utilizou-se a fórmula de amostragem para populações finitas:

$$n = \frac{\frac{z^2\, p(1-p)}{e^2}}{1 + \left(\frac{z^2\, p(1-p)}{e^2 N}\right)}$$

onde:

- n é o tamanho da amostra;
- N é o número total de projetos retornados pela busca;
- $z = 1{,}645$ corresponde ao nível de confiança de 90%;
- $p = 0{,}5$ maximiza a variabilidade da população;
- $e = 0{,}10$ é a margem de erro.

Aplicando a fórmula:

- Para Java ($N = 316$): resultado de 57 projetos;
- Para Python ($N \approx 1200$): resultado de 65 projetos.

Assim, a amostra final utilizada neste estudo foi composta por 57 projetos Java e 65 projetos Python, totalizando 122 repositórios

analisados, garantindo representatividade estatística dentro dos critérios definidos.

3.3 Ferramentas de análise

Para a avaliação da qualidade de código nos projetos selecionados, foram utilizadas três ferramentas de análise estática: *SonarQube*, *Prospector* e *PMD*.

O *SonarQube* foi empregado em todos os projetos pela sua capacidade de integrar múltiplos analisadores e consolidar métricas em um modelo unificado de qualidade, abrangendo aspectos como duplicação, vulnerabilidades e *code smells*. Além de fornecer indicadores quantitativos, a ferramenta permite configurar perfis de qualidade e limiares de aceitação, favorecendo uma comparação padronizada entre sistemas.

Nos projetos desenvolvidos em Python, foi utilizado o *Prospector*, que agrega diferentes analisadores estáticos, como *Pylint*, *McCabe* e *Radon*, com o objetivo de identificar problemas de estilo, complexidade e legibilidade.

Para os projetos implementados em Java, aplicou-se o *PMD*, que realiza a inspeção da árvore sintática do código-fonte a fim de detectar *code smells*, más práticas e padrões de projeto inadequados.

As ferramentas foram escolhidas por sua ampla adoção em estudos empíricos e pela capacidade de fornecer métricas comparáveis de qualidade interna em diferentes ecossistemas. Embora apresentem diferenças no escopo e nas regras, sua combinação permite uma análise consistente e comparável entre linguagens, o que é essencial para o caráter comparativo deste estudo. As mesmas foram utilizadas com suas configurações padrão, sem customização de regras ou ajustes adicionais, a fim de reduzir vieses experimentais e favorecer a reprodutibilidade.

3.4 Métricas avaliadas

As métricas analisadas neste estudo foram selecionadas por refletirem aspectos essenciais da qualidade e manutenibilidade de código, abrangendo desde a estrutura e organização interna até a presença de defeitos e o esforço de manutenção. As informações coletadas pelo *SonarQube* incluíram:

- **Linhas de Código (LOC)**: quantidade total de linhas analisadas, utilizada como base de normalização;
- **Code Smells**: indicações de trechos de código que podem comprometer a manutenibilidade;
- **Bugs**: possíveis falhas de lógica ou defeitos detectados;
- **Duplicação de Código (%)**: percentual de código repetido dentro do projeto;
- **Débito Técnico (minutos)**: estimativa do esforço necessário para correção dos problemas de qualidade;
- **Notas de Manutenibilidade, Confiabilidade e Segurança**: avaliações automáticas atribuídas pelo *SonarQube*;
- **Quality Gate**: resultado geral da análise, indicando se o projeto atinge o nível mínimo de qualidade.

Nos projetos desenvolvidos em Java, foram coletadas as seguintes métricas do *PMD*:

- **Best Practices, Code Style, Design, Documentation, Error-Prone, Multithreading e Performance**: categorias de violações identificadas;
- **Violações Totais**: soma de todas as ocorrências reportadas.

Para os projetos em Python, as métricas obtidas a partir do *Prospector* foram:

- **Issues Totais**: número total de alertas e problemas detectados;
- **Issues por tipo**: classificadas em *Convention*, *Warning* e *Error*.

3.5 Tratamento e análise dos dados

Após a coleta das métricas, os dados foram organizados em planilhas consolidadas e submetidos a um processo de tratamento para garantir consistência e comparabilidade entre os projetos. Inicialmente, foram realizadas verificações para eliminar valores nulos, padronizar unidades de medida e corrigir possíveis inconsistências nas exportações das ferramentas.

Para possibilitar comparações proporcionais entre projetos de diferentes tamanhos, todas as métricas quantitativas foram normalizadas por HLOC (cem linhas de código). Essa abordagem evitou distorções nos resultados, especialmente em projetos menores, permitindo calcular indicadores relativos, como *Code Smells/HLOC*, *Bugs/HLOC*, *Débito Técnico/HLOC* e *Violações/Issues por HLOC*. A normalização tornou os resultados mais equilibrados e facilitou a análise comparativa entre linguagens e repositórios.

Durante a análise, foi considerada a distinção entre os projetos Java e Python, uma vez que cada grupo foi avaliado por ferramentas específicas. Assim, as métricas derivadas do *PMD* (Java) e do *Prospector* (Python) foram interpretadas de forma independente, sendo comparadas apenas dentro de suas respectivas linguagens. Essa separação garantiu a coerência dos resultados e evitou vieses decorrentes das diferenças de escopo entre os analisadores.

Os dados tratados foram organizados em tabelas consolidadas para permitir identificação de padrões e relações entre métricas. Considerando o interesse em analisar relações lineares entre métricas quantitativas de qualidade de código, além das estatísticas descritivas foi aplicada a correlação de Pearson entre variáveis como *code smells*, bugs, duplicação e débito técnico.

A correlação de Pearson foi calculada pela fórmula:

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \cdot \sum(y_i - \bar{y})^2}}$$

em que x_i e y_i representam os valores observados das métricas comparadas, $\bar{x}$ e $\bar{y}$ suas médias, e r varia entre -1 e 1, indicando respectivamente correlação negativa perfeita, positiva perfeita ou ausência de relação linear.

A aplicação da correlação permitiu medir de forma objetiva a intensidade das relações entre deterioração estrutural (por exemplo, code smells e duplicação) e indicadores de esforço de manutenção (como débito técnico e bugs), possibilitando conclusões mais robustas sobre os fatores que influenciam a qualidade interna dos sistemas analisados.

4 Resultados

Os resultados obtidos a partir dos projetos selecionados foram separados em diferentes formas de análise. A organização presente para a elaboração dos resultados permite compreender progressivamente como os dados foram consolidados.

4.1 Comparação das Métricas de Qualidade entre as Linguagens

A Tabela 1 e a Tabela 2 apresentam as estatísticas descritivas completas das principais métricas de qualidade de código para os 57 projetos Java e 65 projetos Python analisados. As métricas incluem valores normalizados por HLOC (como Code Smells/HLOC, Bugs/HLOC, Débito Técnico/HLOC e Violações/HLOC), além de indicadores absolutos (LOC e duplicação de código).

De forma geral, observa-se que os projetos Java apresentam maior variabilidade interna em praticamente todas as métricas analisadas. Embora a média de Code Smells/HLOC em Java (4,10) seja superior à de Python (1,74), o desvio padrão significativamente maior em Java (3,28) indica que os projetos desse ecossistema são mais heterogêneos em termos de estilo, organização interna e maturidade do código. Essa tendência se repete em Bugs/HLOC, no qual Java apresenta média ligeiramente maior (0,19), mas com grande dispersão entre projetos.

Outro destaque importante é o comportamento do débito técnico. Embora os projetos Java apresentem maior débito técnico médio por HLOC (66,67), o ecossistema Python exibe uma dispersão muito mais acentuada (desvio padrão 67,70), revelando a existência de projetos extremamente problemáticos convivendo com repositórios muito mais estáveis e bem estruturados. Esse padrão aponta para uma diferença fundamental entre os ecossistemas: enquanto Java tende a manter uma distribuição mais regular de problemas, Python possui casos extremos que elevam drasticamente a variabilidade global.

A duplicação de código também apresenta comportamento semelhante entre as linguagens. Python exibe uma média levemente superior (6,59% contra 6,04% em Java), mas novamente com maior variabilidade, sugerindo práticas inconsistentes entre projetos. Por fim, em Violações/HLOC, métrica influenciada por PMD (Java) e Prospector (Python), a diferença entre os ecossistemas torna-se mais evidente. Java apresenta valores muito mais altos e uma variabilidade substancial (média de 290,43; desvio de 1498,47), enquanto Python mantém valores bem menores e mais estáveis (média de 8,18; desvio de 11,35), refletindo diferenças tanto na rigidez das ferramentas quanto nos estilos típicos de codificação.

Esses resultados sugerem que, embora Python tenda a apresentar menor densidade média de problemas, sua variabilidade interna evidencia a coexistência de projetos muito bem estruturados com outros altamente suscetíveis a problemas de manutenção. Já Java, por outro lado, mantém valores médios mais elevados, mas com uma distribuição mais uniforme entre os repositórios analisados.

Table 1: Estatísticas Descritivas — Projetos Java

Métrica	Média	Mínimo	Máximo	Desvio Padrão
LOC	21116.22	48	288987	42487.15
Code Smells/HLOC	4.10	0.01	14.25	3.28
Bugs/HLOC	0.19	0	2.03	3.28
Débito Técnico/HLOC	66.67	0	402.39	30.00
Duplicação (%)	6.04	0	59.40	10.31
Violações/HLOC	290.43	12.6	11362.5	1498.47

Table 2: Estatísticas Descritivas — Projetos Python

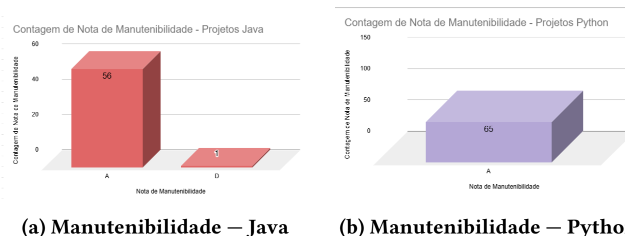
Métrica	Média	Mínimo	Máximo	Desvio Padrão
LOC	19260.87	151	162632	29844.37
Code Smells/HLOC	1.74	0.32	6.49	1.37
Bugs/HLOC	0.15	0	3.52	0.45
Débito Técnico/HLOC	32.86	2.63	132.62	69.14
Duplicação (%)	6.59	0	92.50	14.40
Violações/HLOC	8.18	0.05	57.56	11.35

4.2 Avaliação de Manutenibilidade, Confiabilidade e Segurança dos Projetos

Além das métricas quantitativas normalizadas, foram analisadas as notas automáticas de Manutenibilidade, Confiabilidade e Segurança atribuídas pelo SonarQube aos projetos. As avaliações foram separadas por linguagem para permitir comparação direta entre os ecossistemas.

As Figuras 2a e 2b apresentam a distribuição das notas de manutenibilidade. Nos projetos Java, observa-se predominância quase absoluta da nota A (56 dos 57 projetos), com apenas um repositório recebendo nota D. Esse padrão indica boa qualidade estrutural na maior parte dos projetos, apesar da variabilidade observada nas métricas normalizadas.

Em Python, todos os 65 projetos foram avaliados com nota A, refletindo maior uniformidade nas práticas associadas à manutenibilidade segundo os critérios do SonarQube. Ainda assim, outras métricas analisadas demonstram que essa homogeneidade não impede a existência de dispersão interna em aspectos como débito técnico.



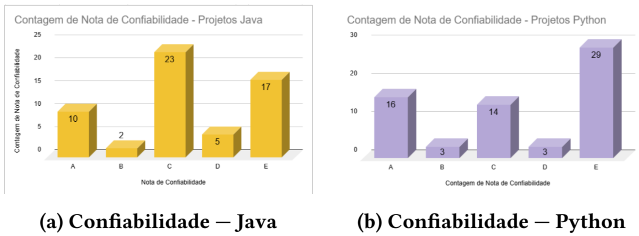
(a) Manutenibilidade — Java **(b) Manutenibilidade — Python**

Figure 2: Distribuição das notas de manutenibilidade atribuídas pelo SonarQube.

Em relação à confiabilidade, nos projetos Java (Figura 3a), observa-se que grande parte das avaliações está concentrada nas notas C e E, revelando distribuição heterogênea de problemas relacionados à lógica, tratamento de erros e defeitos potenciais.

Nos projetos Python (Figura 3b), a concentração em notas mais baixas — sobretudo a nota E — é ainda mais evidente.Embora existam projetos bem avaliados (nota A), a distribuição é mais assimétrica do que em Java, indicando fragilidades acentuadas na robustez de vários repositórios.

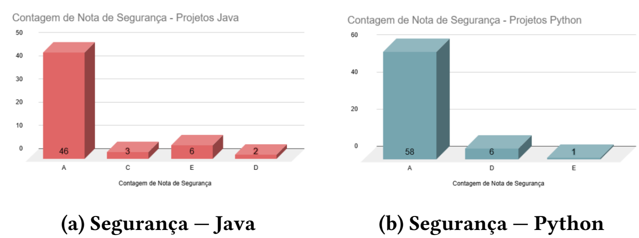
Em relação a segurança, nos repositórios Java (Figura 4a), a maior parte dos projetos recebeu nota A, indicando baixa ocorrência de vulnerabilidades detectáveis. Ainda assim, algumas notas inferiores mostram que certos repositórios acumulam práticas inseguras.



(a) Confiabilidade — Java **(b) Confiabilidade — Python**

Figure 3: Distribuição das notas de confiabilidade atribuídas pelo SonarQube.

A distribuição em Python (Figura 4b) segue padrão semelhante, com forte predominância de nota A. Entretanto, a existência de projetos avaliados com D e E demonstra que vulnerabilidades específicas estão presentes em um subconjunto reduzido.



(a) Segurança — Java **(b) Segurança — Python**

Figure 4: Distribuição das notas de segurança atribuídas pelo SonarQube.

De forma geral, os gráficos apontam que (i) ambos os ecossistemas apresentam forte predominância de nota A em segurança; (ii) em confiabilidade, os projetos Python exibem maior concentração de classificações baixas; (iii) Java apresenta distribuição mais equilibrada, mas também com incidência relevante de notas intermediárias.

4.3 Correlação entre Métricas de Qualidade

A análise das correlações de Pearson permite identificar padrões estruturais entre as métricas coletadas, evidenciando como determinados indicadores de qualidade interna tendem a se comportar em conjunto. Para uma visão abrangente e posteriormente segmentada por linguagem, são apresentados três heatmaps: o panorama geral dos 122 projetos (Figura 5), seguido pelas distribuições específicas dos projetos Java (Figura 6) e Python (Figura 7).

A Figura 5 apresenta o panorama geral das correlações considerando todos os 122 projetos analisados. Observa-se que a correlação mais expressiva ocorre entre Code Smells/HLOC e Débito/HLOC (0.93), indicando uma associação positiva e forte entre a presença de odores de código e o tempo estimado para correção. As demais interações entre métricas de qualidade (como Bugs e Duplicação) apresentam coeficientes próximos de zero, sugerindo fraca associação linear no conjunto geral de dados. Além disso, destaca-se a correlação praticamente nula entre o tamanho do projeto (LOC) e o número de contribuidores (0.02), bem como entre LOC e a duplicação de código (0.51), representando uma correlação moderada.

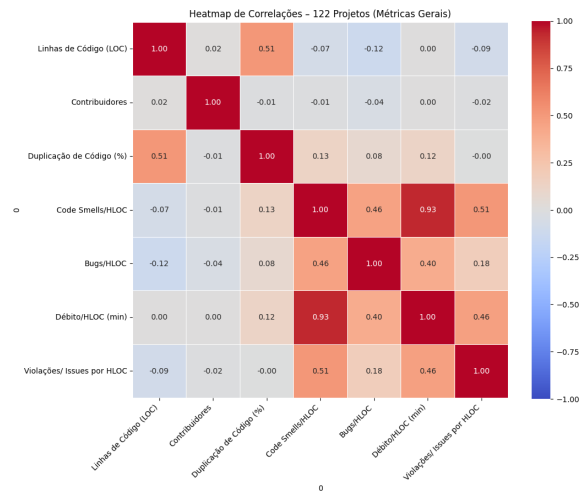


Figure 5: Heatmap de correlações para os 122 projetos analisados (métricas gerais).

Ao segmentar a análise por linguagem, a Figura 6 revela um comportamento distinto, no qual as métricas de qualidade apresentam maior interconectividade. Além da correlação forte entre Code Smells/HLOC e Débito/HLOC (0.94), identificam-se correlações moderadas a fortes entre Bugs/HLOC e Code Smells/HLOC (0.62), bem como entre Bugs/HLOC e Débito/HLOC (0.55). A duplicação de código em Java também demonstra correlação relevante com o tamanho do projeto (LOC), atingindo 0.52.

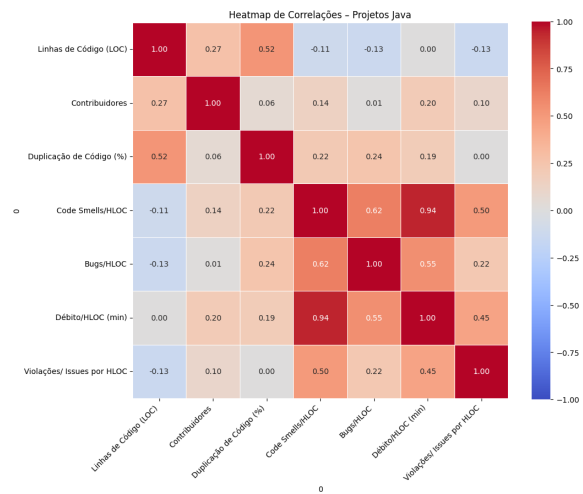


Figure 6: Heatmap de correlações para os projetos Java.

Por fim, a Figura 7 exibe um cenário de correlações mais polarizadas. Destaca-se a correlação muito forte entre Code Smells/HLOC e Débito/HLOC (0.86), um pouco semelhante ao visto nos projetos Java. Em contraste, a relação entre Bugs/HLOC e as demais métricas (Code Smells e Débito) mostra-se fraca ou inexistente (-0.04 e -0.10, respectivamente).A correlação entre LOC e Duplicação de

Código (0.52) configura associação moderada positiva, semelhante à observada nos projetos Java, enquanto a relação entre Bugs e Contribuidores apresenta uma tendência negativa (-0.09).

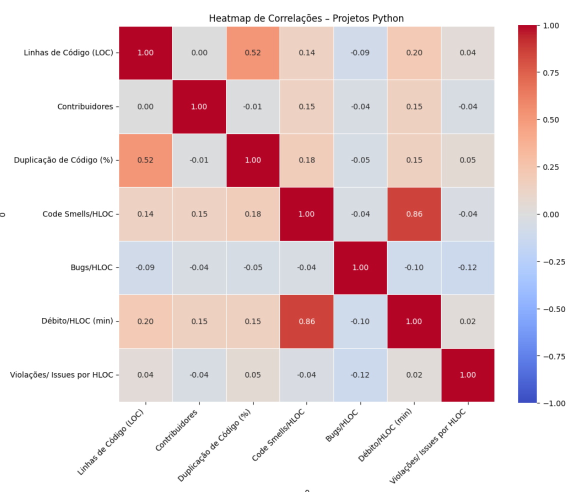


Figure 7: Heatmap de correlações para os projetos Python.

Esses resultados evidenciam que as inter-relações entre métricas de qualidade variam significativamente entre os ecossistemas. Enquanto Java apresenta uma rede de correlações moderadas distribuídas entre diferentes indicadores, Python exibe vínculos mais concentrados e intensos, especialmente entre code smells e débito técnico. No conjunto geral, com exceção da forte correlação entre code smells e débito técnico, a maior parte das interações mostra baixa associação linear, reforçando que tais métricas capturam aspectos distintos da qualidade interna.

4.4 Discussão dos Resultados

Os resultados obtidos revelam padrões consistentes sobre os principais problemas de qualidade presentes nos projetos analisados. De forma geral, observou-se que os ecossistemas Java e Python apresentam fragilidades estruturais semelhantes, embora com intensidades distintas. Projetos Java exibem maior variabilidade interna, com densidades mais elevadas de *code smells* e bugs por HLOC, enquanto Python apresenta valores médios menores, porém com maior assimetria em métricas como débito técnico e confiabilidade.

A análise das notas automáticas do SonarQube reforça esse cenário: ambos os ecossistemas mantêm níveis elevados de segurança, com predominância da nota A, mas divergem quanto à confiabilidade, especialmente em Python, onde há maior concentração de projetos avaliados com notas baixas. A ausência de variação na nota de manutenibilidade, classificada como A em todos os projetos, indica que esse indicador específico não contribui para diferenciar a qualidade entre linguagens.

As correlações de Pearson confirmam que os fatores mais críticos para a degradação da qualidade estão associados a aspectos estruturais internos do código, evidenciando que anomalias internas tendem a se acumular e impactar diretamente o esforço de manutenção. Em contraste, variáveis externas mostraram impacto mínimo, sugerindo que a qualidade está menos relacionada à escala e mais às práticas adotadas durante o desenvolvimento.

Esses achados convergem com a literatura ao indicar que a deterioração estrutural do código tende a aumentar o esforço de manutenção em projetos *open source*. Ao comparar sistematicamente os repositórios Java e Python por meio de métricas normalizadas e do uso combinado de múltiplas ferramentas de análise estática, este estudo contribui com uma visão empírica atualizada sobre como problemas de qualidade se distribuem entre ecossistemas distintos, oferecendo evidências relevantes para apoiar práticas de manutenção, padronização e evolução de software.

5 Conclusões

Este estudo realizou uma avaliação sistemática da qualidade de código em 122 projetos open source escritos em Java e Python, combinando métricas estruturais, ferramentas de análise estática e correlações estatísticas para compreender como características internas do código afetam a manutenibilidade, confiabilidade e esforço de manutenção. Os resultados mostram que ambos os ecossistemas apresentam fragilidades recorrentes, como code smells, duplicação e bugs, embora distribuídas de forma distinta entre as linguagens. Enquanto Java exibe maior densidade média de problemas e forte variabilidade interna, Python demonstra valores médios menores, porém maior assimetria em métricas como débito técnico e confiabilidade.

A análise de correlação evidenciou que code smells, bugs e débito técnico são fortemente inter-relacionados, principalmente em projetos Java, indicando que más práticas acumuladas amplificam o esforço necessário para evoluir o código. Por outro lado, nas correlações analisadas, variáveis externas como tamanho do projeto e número de contribuidores apresentaram impacto reduzido sobre as métricas internas de qualidade, sugerindo, neste conjunto de projetos, maior influência das práticas de desenvolvimento do que de características agregadas do contexto colaborativo. Em conjunto, os achados reforçam a relevância da análise estática e das métricas estruturais como instrumentos para apoiar decisões de manutenção e refatoração em ambientes de software livre.

As limitações deste estudo devem ser consideradas na interpretação dos resultados. A principal delas diz respeito ao uso de ferramentas específicas para cada linguagem analisada. Como o PMD é dedicado ao ecossistema Java e o Prospector ao ecossistema Python, os analisadores adotados possuem escopos, regras e níveis de rigor distintos, o que introduz assimetria metodológica e limita comparações diretas entre linguagens. Essa assimetria metodológica impede comparações totalmente equivalentes entre as linguagens e limita a generalização dos achados. Além disso, a natureza especializada dessas ferramentas restringiu a análise às duas linguagens abordadas, impossibilitando a inclusão de outras linguagens relevantes. Outra limitação relaciona-se ao fato de que o estudo se baseou exclusivamente em análise estática, sem considerar comportamentos de execução ou dados de testes, que poderiam fornecer uma visão mais abrangente sobre confiabilidade e desempenho.

Diante dessas limitações, recomenda-se que trabalhos futuros considerem as seguintes direções para aprofundar e ampliar os achados deste estudo:

- Explorar outras linguagens amplamente utilizadas, como JavaScript, TypeScript, C# e C++ [14], para obter uma visão mais diversa e abrangente sobre qualidade de código.
- Acompanhar a evolução das métricas de qualidade ao longo do tempo para identificar ciclos de deterioração, pontos de inflexão e o impacto de refatorações estruturais.
- Utilizar ferramentas multilíngues capazes de aplicar regras equivalentes entre diferentes ecossistemas, reduzindo assimetrias metodológicas.
- Combinar análises estáticas com técnicas dinâmicas, obtendo resultados sobre métricas estruturais com dados reais de execução, cobertura de testes e detecção de falhas, oferecendo uma avaliação mais completa dos atributos de confiabilidade.
- Incluir testes de significância estatística para as correlações obtidas, a fim de avaliar a robustez inferencial das relações identificadas entre as métricas.
- Investigar qualitativamente os fatores associados à assimetria acentuada observada nos projetos, discutindo o papel dos outliers na distribuição das métricas de qualidade.

Em síntese, os resultados destacam a relevância de métricas estruturais e de ferramentas de análise estática no apoio à manutenção de projetos open source, mas também revelam limitações ao comparar ecossistemas distintos. A análise de 122 repositórios Java e Python com métricas normalizadas oferece subsídios para decisões de refatoração, padronização e governança de código, além de apontar direções para futuras investigações que integrem análises estáticas e dinâmicas.

Para garantir transparência e reprodutibilidade, o dataset dos 122 repositórios e os scripts utilizados na geração das estatísticas e heatmaps estão disponíveis em: https://github.com/Guiioff/oss-code-quality-java-python. O repositório inclui os dados e instruções para replicação do experimento.

References

[1] Iso/iec 25010:2011 — systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, 2011.
[2] G. Cotrim, M. Maia, and C. de Souza. Software product quality evaluation: a systematic mapping study. *Information and Software Technology*, 121:106268, 2020.
[3] R. Alves, G. Santos, and R. Ribeiro. A systematic mapping on software quality models. *Journal of Software Engineering Research and Development*, 9(1):1–22, 2021.
[4] N. Yadav and S. S. Rathore. Software quality prediction: A systematic literature review and research directions. *Applied Soft Computing*, 111:107700, 2021.
[5] A. Sharma and A. Kumar. Analyzing software code quality attributes: A systematic review. *Software Quality Journal*, 30:1065–1090, 2022.
[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, Boston, MA, USA, 2 edition, 2018.
[7] E. M. Batista, G. Braga e Silva, and T. R. M. B. Silva. Diversidade de gênero em projetos open source: um estudo da relevância dos comentários postados em issues do github. In *Anais do Women in Information Technology (WIT 2022)*, pages 197–202, Porto Alegre, Brasil, 2022. SBC. doi: 10.5753/wit.2022.222628.
[8] G. Kapllani, I. Khomyakov, R. Mirgalimova, and A. Sillitti. An empirical analysis of the maintainability evolution of open source systems. In *Proceedings of the 16th International Conference on Open Source Systems (OSS 2020)*, pages 78–86, Cham, 2020. Springer. doi: 10.1007/978-3-030-47240-5_8.
[9] R. Brown and D. Greer. Software code smells and defects: An empirical investigation. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 570–580, Setúbal, Portugal, 2023. SCITEPRESS. doi: 10.5220/0011974500003464.
[10] J. R. Ramos and T. F. Malacrida. Processos, métodos e práticas de engenharia de software em projetos software livre: um estudo de caso owncloud e nextcloud. *Colloquium Exactarum*, 10(2):53–59, 2018. URL https://journal.unoeste.br/index.php/ce/article/view/2677.
[11] A. Alami, R. Pardo, and J. Linåker. Free open source communities sustainability: Does it make a difference in software quality? *Empirical Software Engineering*, 29(4):114, 2024. doi: 10.1007/s10664-024-10529-6.
[12] Harald Schaffernak, Birgit Moesl, Philipp Url, Ioana Victoria Koglbauer, and Wolfgang Vorraber. Towards sustainable software quality in use: a review of measures. *Next Research*, 2(3):100680, 2025. ISSN 3050-4759. doi: https://doi.org/10.1016/j.nexres.2025.100680. URL https://www.sciencedirect.com/science/article/pii/S3050475925005470.
[13] Siyuan Jin, Mianmian Zhang, Yekai Guo, Yuejiang He, Ziyuan Li, Bichao Chen, Bing Zhu, and Yong Xia. A quantitative analysis of open source software code quality: Insights from metric distributions, 2023. URL https://arxiv.org/abs/2307.12082. Acesso em: 19 out. 2025.
[14] A. Martini and J. Bosch. Theoretical perspectives on code quality and their implications for tooling. *Information and Software Technology*, 2023.
[15] A.-J. Molnar and S. Motogna. Long-term evaluation of technical debt in open-source software. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'20)*, Bari, Italy, 2020. doi: 10.1145/3382494.3410673. Acesso em: 05 out. 2025.
[16] S. Chowdhury, H. Kidwai, and M. Asaduzzaman. Evidence is all we need: Do self-admitted technical debts impact method-level maintenance? *arXiv preprint*, 2025. URL https://arxiv.org/abs/2411.13777. Acesso em: 05 out. 2025.
[17] I. Kapllani, S. Nepal, and X. Zhang. Measuring software code quality: A systematic mapping study. *Information and Software Technology*, 127:106368, 2020.
[18] A. Alami, J. Pardo, and J. Linåker. Understanding the relation between software metrics and technical debt in open source projects. *Journal of Systems and Software*, 178:110958, 2021.
[19] K. Sambaturu, S. Jain, and S. S. Rathore. A comprehensive review on software metrics for quality evaluation. *Software Quality Journal*, 30(3):845–872, 2022.
[20] V. S. S. Sambaturu. Evolution analysis of software quality metrics in an open-source java project: A case study on testng. https://arxiv.org/abs/2505.22884, 2025. Preprint, arXiv.
[21] E. Yeoboah and M. Popoola. Efficacy of static analysis tools for software defect detection on open-source projects. In *Proceedings of the CSCI Conference on Computer Science and Computational Intelligence*, Las Vegas, USA, 2023. IEEE Computer Society.
[22] R. Hashmat, J. Zhang, A. Pradhan, Y. Dang, L. Huang, C. Zhang, and D. Xu. Insights from running 24 static analysis tools on open source software repositories. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2024)*, Cham, 2024. Springer. doi: 10.1007/978-3-031-80020-7_13.
[23] L. Chen, R. Oliveira, and P. Gupta. Dynamic versus static techniques for software defect detection: A comparative review. *Empirical Software Engineering*, 2023.
[24] U. Mansoor, E. Fernandes, and J. Kim. Linting practices and syntactic rule-based analysis in modern development pipelines. *Software: Practice and Experience*, 2022.
[25] F. Alomari and Y. Liu. Structural metrics and their impact on software maintainability. *Journal of Software Quality and Evolution*, 2023.
[26] X. Zhang, A. E. Hassan, and G. Bavota. Hybrid approaches to code analysis: Combining syntactic, structural, and behavioral dimensions. *IEEE Transactions on Software Engineering*, 2024.
[27] M. Rahman, M. M. Rahman, and L. Williams. Software metrics and automated code analysis: a systematic mapping study. *Journal of Systems and Software*, 192: 111498, 2022.
[28] C. Liu, Y. Zou, and A. Hassan. Understanding code quality and evolution through automated static analysis: a large-scale study. *Empirical Software Engineering*, 29 (1):25–49, 2024.
[29] N. E. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach.* CRC Press, Boca Raton, 3 edition, 2014.
[30] G. Pinto, Y. Kamei, and Y. Zhang. Static analysis tools revisited: Adoption, challenges, and lessons learned in modern software development. *Empirical Software Engineering*, 28(2):31–58, 2023.
[31] F. Zampetti, G. Bavota, R. Pietrantuono, and S. Russo. Revisiting the impact of code smells on software maintainability. *Information and Software Technology*, 121:106268, 2020.
[32] M. Beller, H. Gall, and A. Zeller. The science of static analysis: What works, what doesn't, and why. *IEEE Software*, 39(3):46–54, 2022.
[33] M. A. Storey, J. Zhang, and A. Hassan. Beyond metrics: Combining automated and human-centric approaches for software quality evaluation. *Journal of Systems and Software*, 198:112009, 2023.
[34] Umar Iftikhar, Nauman Bin Ali, Jürgen Börstler, and Muhammad Usman. A tertiary study on links between source code metrics and external quality attributes. *Information and Software Technology*, 165:107348, 2024. doi: 10.1016/j.infsof.2023.107348.
[35] Antônio Carlos Gil. *Métodos e técnicas de pesquisa social.* Atlas, São Paulo, 6 edition, 2008.