

# Estudo de Caso sobre a Qualidade do Código-Fonte Gerado por Inteligências Artificiais Generativas no Contexto do Desenvolvimento de Software

Florisvaldo A. Crepaldi Junior<sup>1</sup>, Fernando Sambinelli

<sup>1</sup>Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP)  
Câmpus Hortolândia – São Paulo - SP - Brasil

junior.crepaldi@aluno.ifsp.edu.br, sambinelli@ifsp.edu.br

**Abstract.** *This case study examines the quality of the source code generated by tools based on Generative Artificial Intelligence (GAI), such as ChatGPT, Gemini and GitHub Copilot, in the context of software development. The static analysis carried out with the SonarQube tool investigated aspects related to the maintainability, security and reliability of the codes produced. The results show that although the codes generated are, for the most part, functionally correct, in certain cases they have significant deficiencies in terms of maintainability, including problems such as inadequate encapsulation and redundancies. On the other hand, no problems directly associated with security were detected, and overall reliability was considered satisfactory. In conclusion, despite the high potential of AGI tools, their use requires caution, especially on the part of inexperienced developers, and it is essential that the content generated is subjected to rigorous analysis by the implementers.*

**Resumo.** *Este estudo de caso examina a qualidade do código-fonte gerado por ferramentas baseadas em Inteligência Artificial Generativa (IAG), como ChatGPT, Gemini e GitHub Copilot, no contexto do desenvolvimento de software. Por meio da análise estática realizada com a ferramenta SonarQube, foram investigados aspectos relacionados à manutenibilidade, segurança e confiabilidade dos códigos produzidos. Os resultados apontam que, embora os códigos gerados sejam, em grande parte, funcionalmente corretos, em determinados casos apresentam deficiências significativas no que diz respeito à manutenibilidade, incluindo problemas como encapsulamento inadequado e redundâncias. Por outro lado, não foram detectados problemas diretamente associados à segurança, e a confiabilidade geral foi considerada satisfatória. Conclui-se que, apesar do elevado potencial das ferramentas de IAG, seu uso exige cautela, especialmente por parte de desenvolvedores inexperientes, sendo essencial que o conteúdo gerado seja submetido a uma análise rigorosa por parte dos implementadores.*

## 1. Introdução

Nos últimos anos, ferramentas de Inteligência Artificial Generativa (IAGs) ganharam popularidade significativa, impulsionadas por avanços tecnológicos. Esse crescimento pode ser exemplificado pelo ChatGPT da OpenAI, que atingiu mais de 1 milhão de usuários em

apenas cinco dias de lançamento, conforme relatado pela [UBS 2023]. Outras ferramentas notáveis incluem o Gemini da Google e o Copilot da Microsoft, que geram conteúdos variados a partir de entradas simples fornecidas pelos usuários. Desde a geração de textos até trechos de código complexos com lógica, contexto e regras de negócios.

A Inteligência Artificial Generativa (IAG) é uma tecnologia emergente que busca criar novos conteúdos a partir de dados, simulando o processo humano de criação. Conforme [Wang et al. 2023] "é uma categoria específica de IA que visa gerar de forma autônoma novos conteúdos que imitem o conteúdo criado por humanos", assim possuem um enorme potencial para transformar a maneira como realizamos tais tarefas. Os desenvolvedores, em particular, são uma parcela da sociedade que pode se beneficiar de ferramentas como as citadas. Eles podem integrar o uso de tais ferramentas em suas rotinas, solicitando de maneira simples e rápida a geração de trechos de código com base em explicações de lógica, arquivos, contexto ou regras de negócios fornecidas como entradas.

Com a evolução contínua e a facilidade proporcionada pelas IAGs, surge uma questão crucial: qual é a qualidade das respostas geradas por essas ferramentas? Especificamente no contexto do desenvolvimento de *software*, a qualidade dos códigos produzidos é de suma importância, abrangendo aspectos como segurança, manutenibilidade, coesão e clareza. Essa preocupação se intensifica consideravelmente à medida que essas ferramentas são cada vez mais utilizadas, muitas vezes por usuários com conhecimento limitado ou básico sobre os princípios de desenvolvimento e qualidade de *software*. Essa crescente adoção pode resultar em *bugs* e problemas nos produtos finais, especialmente em ambientes de produção.

Este trabalho propõe um estudo de caso focado na avaliação da qualidade do código gerado por IAGs por meio de uma análise estática. Esta análise será conduzida utilizando uma ferramenta específica e reconhecida para esse propósito, o SonarQube desenvolvido pela SonarSource, que é uma ferramenta de revisão de código automática e auto gerenciada [SonarSource 2024]. O SonarQube possui amplos recursos de relatórios e *dashboards*, que facilitam o monitoramento contínuo da qualidade do código, ajudando a identificar áreas de melhoria, comunidade ativa e suporte robusto. Além disso, o SonarQube garante conformidade com padrões de codificação e regulamentos específicos do setor, reduzindo riscos e aumentando a confiança no *software* desenvolvido. Estes aspectos o diferenciam de outras ferramentas do mesmo segmento como Checkstyle ou Pylint.

O trabalho está organizado da seguinte forma: na Seção II, será apresentado o referencial teórico, que aborda os principais conceitos relacionados ao tema proposto. A Seção III traz uma revisão dos trabalhos correlatos, apresentando o estado da arte na área e destacando pesquisas relevantes. Na Seção IV, detalharemos a metodologia utilizada no estudo, enquanto a Seção V descreve, passo a passo, os procedimentos adotados para alcançar os resultados esperados. Por fim na seção VI, é exposta a conclusão do estudo e são apresentados possíveis trabalhos futuros que não foram abordados no atual trabalho.

## **2. Referencial Teórico**

Nesta seção, é apresentada uma breve descrição dos conceitos relevantes para o desenvolvimento do presente estudo.

## 2.1. Qualidade

O termo qualidade pode ter distintas definições de acordo com o contexto na qual está inserido, sob a visão mais ampla e generalista temos a óptica da administração onde a “qualidade é a excelência de seu produto, incluindo sua atratividade, ausência de defeitos, confiabilidade e segurança a longo prazo.”[Bateman and Snell 1998].

## 2.2. Qualidade de Software

A qualidade de *software* é definida como a conformidade com requisitos, sejam eles funcionais ou não funcionais. Segundo [Sommerville 2018], a indústria desenvolveu uma definição de qualidade baseada na conformidade com uma especificação detalhada do produto. Assim, a qualidade de *software* vai além de apenas fazer o sistema funcionar corretamente. O autor também afirma que a qualidade de *software* não se preocupa apenas com a correta implementação das funcionalidades, mas também depende dos atributos não funcionais do sistema. Esses atributos incluem, mas não se limitam a, eficiência, confiabilidade, usabilidade, manutenibilidade e portabilidade, que são essenciais para garantir que o *software* atenda às expectativas dos usuários e aos padrões da indústria.

Complementando essa visão, [Pressman 2016] define a qualidade de *software*, em um sentido mais geral, como “uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam”. Dessa forma, a qualidade é vista não apenas como uma questão técnica, mas também como um processo integrado à criação de valor para desenvolvedores e usuários.

## 2.3. Análise Estática de Código

A análise estática de código é o processo de examinar o código-fonte de um *software* sem executá-lo. Esse tipo de análise é realizado com o objetivo de identificar erros, *bugs*, vulnerabilidades, padrões de codificação inadequados e outras questões relacionadas à qualidade do código. Um diferencial importante da análise estática é a possibilidade de ser executada nas etapas iniciais do desenvolvimento, pois não necessita da execução do *software*. Como afirmam [Alqaradaghi and Kozsik 2024] “a análise estática do código-fonte (e *bytecode*) é uma técnica de detecção de vulnerabilidade que permite a revisão escalonável do código para segurança no início do ciclo de desenvolvimento de *software*. Não requer sistemas executáveis e pode ser aplicada a partes específicas da base de código”. Isso torna a análise estática uma ferramenta valiosa para identificar e corrigir problemas antecipadamente, contribuindo para a melhoria contínua da qualidade do *software*.

## 2.4. Inteligência Artificial Generativa (IAG)

A Inteligência Artificial Generativa é um ramo da IA que se concentra na criação de novos conteúdos a partir de grandes volumes de dados nos quais foi treinada. Ao invés de apenas pesquisar ou analisar dados existentes, a IAG compreende as solicitações e gera novos conteúdos com base em seu “conhecimento” adquirido durante o treinamento. Como descrito por [Gozalo-Brizuela and Garrido-Merchan 2023], a IA generativa é capaz de criar conteúdo novo, em vez de apenas atuar ou analisar dados previamente existentes. Além disso, [Wang et al. 2023] definem a IAG como uma categoria específica de IA que gera de forma autônoma novos conteúdos que simulam o conteúdo produzido por humanos em diversas modalidades.

### 3. Trabalhos correlatos

Nesta seção é apresentada uma revisão dos principais estudos relacionados ao tema, destacando suas metodologias e resultados. A partir dessa análise, busca-se contextualizar a presente pesquisa, evidenciando como ela se diferencia e contribui para o avanço do conhecimento na área.

Para fundamentar este trabalho, foram utilizados diversos estudos acadêmicos, como o artigo de [Liu et al. 2024]. O estudo analisou mais de 4.000 trechos de códigos gerados pelo ChatGPT em Java e Python, baseados em mais de 2.000 tarefas do LeetCode. Inicialmente, avaliou-se a precisão do ChatGPT na geração de código, identificando fatores que afetam sua eficácia, como dificuldade das tarefas, linguagem, tempo de introdução e tamanho do programa. Em seguida, foram identificados e caracterizados possíveis problemas na qualidade do código gerado, através de análise estática sobre sua manutenibilidade.

Os resultados mostraram que 67,78% dos códigos foram implementados corretamente, 26,61% com resultados inconsistentes e apenas 4,35% com erros de compilação ou em tempo de execução. Além disso, 53% dos códigos em Java e 37% em Python, mesmo após passarem nos testes, apresentaram problemas de estilo e manutenção. O estudo destaca a necessidade de abordar essas questões para garantir a eficácia do código gerado por IA a longo prazo. A capacidade de autocorreção do ChatGPT foi testada, revelando que ele pode corrigir parcialmente os problemas de qualidade quando recebe *feedback* de ferramentas de análise estática e erros de tempo de execução. A eficácia dessa autocorreção depende das informações de *feedback*, linguagem de programação e tipo de problema.

Outro estudo relevante é o de [Kabir et al. 2024] que investigaram a qualidade dos códigos gerados pelo ChatGPT em comparação às respostas humanas no Stack Overflow. A análise envolveu 500 respostas geradas pela IA, considerando critérios como correção, abrangência e clareza. Os resultados indicaram que, embora 52% das respostas apresentassem informações incorretas, o estilo articulado da IA foi preferido por 35% dos participantes. Isso sugere que, em alguns contextos, o apelo linguístico pode superar a precisão técnica. Ainda assim, foi identificado que 39% dos participantes ignoraram a desinformação presente nas respostas do ChatGPT, destacando a necessidade de combater a propagação de informações erradas em respostas geradas por IA. A análise manual, combinada com a avaliação linguística e o estudo com usuários, demonstrou que, apesar do bom desempenho em diversos casos, o ChatGPT frequentemente comete erros e tende a prolongar desnecessariamente suas respostas. Contudo, seus recursos linguísticos mais ricos explicam a preferência de alguns usuários pelas respostas da IA em detrimento das humanas, mesmo diante de inconsistências.

### 3.1. Comparativo entre trabalhos

**Tabela 1.** Apresenta uma comparação entre este estudo e os principais trabalhos correlatos, destacando aspectos como metodologia utilizada e métodos de análise.

Características	Trabalhos		
	[Lui et al. 2024]	[Kabir et al. 2024]	Este Trabalho
Análise em múltiplas IAGs	Não	Não	Sim
Múltiplas linguagens	Sim	Sim	Não
Problemática dentro de contexto	Não	Não	Sim
Uso da análise estática	Sim	Não	Sim

## 4. Metodologia

Esta seção apresenta o processo proposto para avaliar a qualidade dos trechos de código gerados pelas ferramentas de IAGs. A metodologia foi organizada em seis etapas principais, conforme ilustrado na Figura 1, contemplando desde a definição da problemática e a seleção das ferramentas utilizadas, até a análise dos resultados obtidos. Cada etapa foi cuidadosamente planejada para garantir uma avaliação precisa e comparativa da eficácia e das limitações das IAGs no desenvolvimento de código-fonte.

Inicialmente, a primeira etapa focou na definição da problemática central, que consistia em avaliar a eficácia e limitações das IAGs no desenvolvimento de código. Nesta fase, foram escolhidas as ferramentas que dariam suporte à análise estática e ao desenvolvimento das aplicações, bem como as IAGs específicas (como ChatGPT, Gemini, e GitHub Copilot) a serem examinadas. Com a problemática e os recursos definidos, a etapa seguinte (passo II) envolveu a formulação de pedidos padronizados submetidos a cada IAG. Essa padronização visou manter a uniformidade dos resultados, garantindo que as respostas das IAGs fossem comparáveis e refletissem padrões consistentes de geração de código.

Após a geração dos códigos, os passos III e IV se concentraram na estruturação desses trechos em projetos utilizando o *framework* Spring, um ambiente amplamente usado para desenvolvimento de aplicações em Java. Essa organização dos códigos foi essencial para que, na quinta etapa (passo V), os projetos pudessem ser submetidos a ferramentas de análise estática, especificamente o SonarQube. Este *software* permitiu quantificar e avaliar a qualidade dos códigos, oferecendo uma visão detalhada das métricas de qualidade em segurança, confiabilidade e manutenibilidade.

Finalmente, na sexta e última etapa, os dados provenientes das análises realizadas pelo SonarQube foram cuidadosamente examinados para destacar os pontos fortes e fracos dos códigos gerados por cada IAG. Essa análise conclusiva possibilitou identificar as áreas de desempenho satisfatório e aquelas que requerem melhorias, assegurando assim que os objetivos do estudo fossem alcançados e que as contribuições da pesquisa fossem fundamentadas em dados consistentes e detalhados.

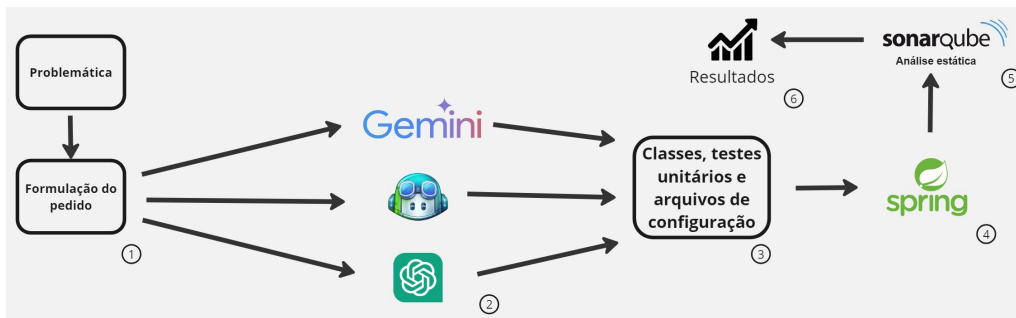


Figura 1. Etapas da Metodologia deste Trabalho

## 5. Desenvolvimento

Esta seção apresenta e descreve com detalhes as atividades em cada uma das etapas de desenvolvimento do atual estudo.

### 5.1. Pesquisa bibliográfica

A pesquisa bibliográfica constituiu o ponto de partida deste estudo. A revisão aprofundada de conceitos como qualidade, qualidade de *software*, IAGs e análise estática foi fundamental para construir uma base sólida de conhecimento e estabelecer um entrosamento dos pensamentos. Além das definições e conceitos, a análise de trabalhos relacionados foi crucial para direcionar a pesquisa, permitindo identificar questões relevantes e lacunas a serem exploradas. Essa etapa foi crucial para o desenvolvimento de uma compreensão abrangente do tema e para a formulação das hipóteses, objetivos e limitações da pesquisa.

### 5.2. Definições

As ferramentas utilizadas para a elaboração do estudo foram selecionadas por aspectos específicos, a seguir será apresentado as ferramentas e a justificativa para o uso de tais no contexto do estudo.

#### 5.2.1. IAGs

As IAGs selecionadas para este estudo de caso foram o ChatGPT da OpenAI, o Gemini da Google e o GitHub Copilot Chat. O ChatGPT, desenvolvido pela OpenAI, é um modelo que utiliza uma interface de diálogo (*chat online*) para responder amplamente às perguntas dos usuários. Ele tem a capacidade de reconhecer seus próprios erros e rejeitar solicitações inapropriadas [OpenAI 2022]. Já o Gemini, da Google, é semelhante ao ChatGPT, porém com foco em processamento *multimodal*, ou seja, ele pode interpretar e gerar respostas a partir de diferentes fontes, como áudio, vídeo, documentos e textos combinados [Pichai and Hassabis 2023]. Por outro lado, o GitHub Copilot Chat se diferencia por ser especializado no suporte ao desenvolvimento de códigos e comandos. Além disso, o acesso a essas ferramentas varia: enquanto o ChatGPT e o Gemini estão disponíveis via aplicativo móvel ou navegador, o Copilot só pode ser acessado através de integração com uma IDE, funcionando como uma extensão [GitHub 2024].

## 5.2.2. Ferramentas de análise estática

Para realizar a análise estática, optou-se pelo SonarQube, uma ferramenta *on-premise*, ou seja, instalada e executada localmente na infraestrutura do usuário, o que garante maior controle dos dados e processos. Reconhecida por sua capacidade de detectar problemas de codificação em diversas linguagens, *frameworks* e plataformas [SonarSource 2024], o SonarQube também oferece recursos robustos para a geração de relatórios e *dashboards*, facilitando o acompanhamento contínuo da qualidade do código e a identificação de oportunidades de melhoria. A ferramenta destaca-se ainda por garantir a conformidade com padrões de codificação, reduzindo riscos e aumentando a confiança no *software* desenvolvido.

## 5.2.3. Problemática

Através da problemática, buscou-se uma avaliação minuciosa das ferramentas, com o intuito de identificar seus pontos fortes e fracos. Foi definido um cenário comum no dia a dia de um desenvolvedor júnior, com pouca experiência ou conhecimento em arquitetura, segurança e boas práticas de programação. Esse perfil foi escolhido porque, ao lidar com trechos de código que contenham erros ou *bugs*, as chances de esses problemas passarem despercebidos são maiores em comparação a desenvolvedores mais experientes. Dessa forma, podemos simular situações em que essas falhas não são corrigidas, gerando consequências potencialmente graves para o produto final.

A solução proposta envolve a criação de um sistema/API para gerenciamento de usuários e postagens, desenvolvido em Java 17 e utilizando o banco de dados MySQL 8 para a persistência dos dados. O sistema proposto deve fazer uso do Spring Boot, o *framework* Java mais popular no mundo, conhecido por tornar o desenvolvimento em Java mais ágil, simples e seguro. O Spring Boot facilita a criação de aplicações robustas ao fornecer uma configuração mínima, permitindo que os desenvolvedores se concentrem nas funcionalidades do sistema[Spring 2024].

Por ser um projeto que implementa o CRUD, acrônimo de *Create, Read, Update e Delete*, refere-se às operações básicas para gerenciar dados em um banco de dados, sendo a base de muitas aplicações, como sistemas web ou *mobile*. Essas operações permitem criar novos registros, ler e exibir informações armazenadas, atualizar dados existentes e deletar registros. Geralmente, são implementadas com bancos de dados relacionais, como MySQL ou PostgreSQL, e *frameworks* que facilitam sua aplicação, sendo comum seu desenvolvimento entre desenvolvedores, especialmente no início da carreira, e permite a prática de diversos conceitos essenciais, como:

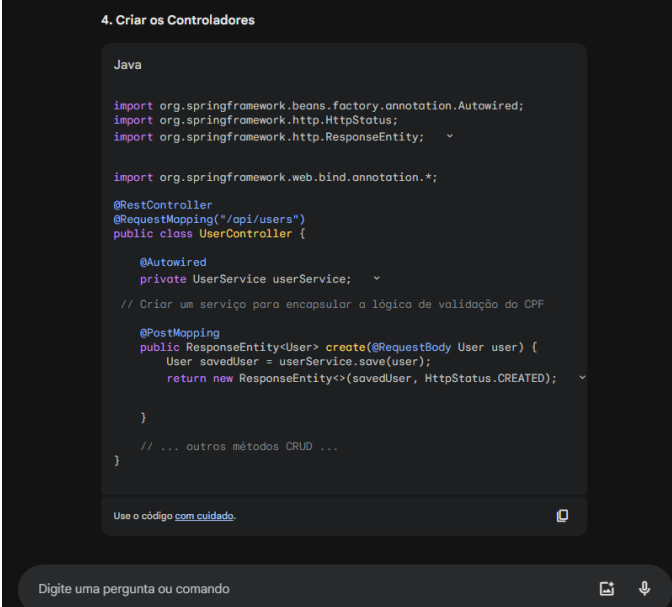
- Acesso a Banco de Dados: Configuração e conexão com o MySQL, além da execução eficiente e segura de operações CRUD (*Create, Read, Update, Delete*).
- Orientação a Objetos (OO): Modelagem das entidades Usuario e Postagem, com uso adequado de encapsulamento e boas práticas de design orientado a objetos.
- Relacionamento entre Entidades: Definição de relacionamentos entre as entidades, como a associação entre Usuario e Postagem (por exemplo, um usuário pode ter várias postagens), praticando a criação de relações um-para-muitos e muitos-para-um.

- Uso de Interfaces e Abstração: Implementação de serviços por meio de interfaces para garantir baixo acoplamento e alta coesão, além da aplicação do padrão DAO (*Data Access Object*) para separar a lógica de negócios do acesso ao banco de dados.
- Arquitetura MVC: Organização do código em camadas distintas, separando o Modelo, a Visão e o Controle para facilitar o desenvolvimento, a manutenção e a escalabilidade do sistema.
- Boas Práticas de Programação: Estruturação organizada do projeto, com a divisão lógica em pacotes, como *controller*, *service*, *repository* e *model*, garantindo modularidade e facilidade de manutenção.

### 5.3. Formulação e realização do pedido

Com as definições sobre as ferramentas e a problemática estabelecidas, iniciou-se a elaboração do pedido, que seria uniformizado para todas as plataformas de IAGs. Esse pedido único detalhou as regras e funcionalidades necessárias, incluindo as entidades, seus respectivos atributos, relacionamentos, regras de negócio, estrutura (*controller*, *model*, *repository*), validações, como a de CPF, entre outras definições.

Após a submissão do pedido às ferramentas, conforme ilustrado nas Figura 2 e Figura 3, que mostram trechos das respostas da ferramenta Gemini, do Google, foi possível observar um comportamento interessante. Devido à complexidade e extensão do pedido, a ferramenta apresentou tanto trechos de código concretos quanto partes em pseudocódigo ou meros comentários sobre a implementação. Esse resultado, no entanto, foge ao objetivo do estudo, que é avaliar a qualidade do código gerado pelas ferramentas, e não suas sugestões ou explicações em pseudocódigo.



```
4. Criar os Controladores

Java

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    // Criar um serviço para encapsular a lógica de validação do CPF

    @PostMapping
    public ResponseEntity<User> create(@RequestBody User user) {
        User savedUser = userService.save(user);
        return new ResponseEntity<>(savedUser, HttpStatus.CREATED);
    }

    // ... outros métodos CRUD ...
}

Use o código com cuidado.
```

Figura 2. Captura de tela que ilustra a resposta gerada pela ferramenta Gemini da Google após o pedido de geração de código. Observa-se como a ferramenta apresentou tanto trechos de código concretos quanto pseudocódigo, destacando as limitações no atendimento completo do pedido.





The screenshot shows a dark-themed code editor with the title "5. Criar o Serviço para Validação do CPF". The code is in Java and defines a class `UserService`. The code is as follows:

```
Java

public class UserService {
    // Lógica para validar o CPF, conforme as regras especificadas
    // ...
}
```

At the bottom of the editor, there is a footer that says "Use o código [com cuidado](#)." and a copy icon.

**Figura 3. Captura de tela que ilustra a resposta gerada pela ferramenta Gemini da Google após o pedido de geração de código. Observa-se como a ferramenta apresentou uma resposta parcial sobre o questionamento inicial.**

Esse comportamento foi observado também nas demais ferramentas testadas. Por esse motivo, o pedido original, que consistia em um único bloco, foi fragmentado em seis partes, a fim de obter respostas mais precisas e evitar sugestões genéricas ou pseudocódigo, priorizando a geração de código completo pelas ferramentas.

A divisão seguiu o seguinte fluxo: no primeiro pedido, foi solicitada a geração do arquivo de configuração (*application.properties*) do Spring Boot para o acesso ao banco de dados. Na segunda etapa, a solicitação foi para a criação de uma *Annotation* personalizada para a validação de CPF. Na terceira, foi requisitada a geração da estrutura do projeto Spring Boot, incluindo a criação da API com as entidades necessárias, seus relacionamentos, atributos e o uso de pacotes específicos. Na quarta parte, foi solicitada a geração das classes de repositórios para o acesso às informações. A quinta etapa contemplou a criação dos controladores, com os *endpoints* necessários para realizar operações básicas nas entidades. Por fim, na sexta etapa, a solicitação envolveu a criação de testes unitários para os métodos presentes nos controladores.

Na (Figura 4), será possível observar um exemplo de como a fragmentação do pedido em seis partes impactou positivamente os resultados gerados pelas ferramentas. A divisão permitiu que cada solicitação fosse tratada de forma mais precisa, resultando em uma maior qualidade e completude dos códigos gerados.

Mesmo com a fragmentação do pedido, algumas IAGs apresentaram dificuldades. No caso da Gemini, por exemplo, não foi possível gerar as classes de testes completas. Ao ser solicitada, a ferramenta retornava uma mistura de trechos de código concretos e pseudocódigo ou comentários. Mesmo após novas tentativas, insistindo na geração do código completo, a ferramenta não conseguiu executar a tarefa de forma satisfatória. A (Figura 5) ilustra um exemplo dessas tentativas.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable
Long id) {
        Optional<User> user = userService.getUserById(id);
        return user.map(ResponseEntity::ok).orElse(ResponseEntity.notFound());
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User createdUser = userService.createUser(user);
        return ResponseEntity.created(URI.create("/api/users/" + createdUser.getId()));
    }

    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable
Long id, @RequestBody User user) {
        User updatedUser = userService.updateUser(id, user);
        return ResponseEntity.ok(updatedUser);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        return ResponseEntity.noContent();
    }
}
```

Figura 4. Captura de tela que ilustra a resposta gerada pela ferramenta Gemini da Google após o pedido fragmentado para a geração de código. Observa-se como a ferramenta apresentou uma resposta mais completa.

```
Exemplo de Testes para o Controlador PostController :
```

```
Java

@RunWith(SpringRunner.class)
@WebMvcTest(PostController.class)
public class PostControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private
    UserRepository userRepository;

    @Autowired
    private PostRepository postRepository;

    @Before
    public void setUp() {
        // Criar objetos mock para testar
        User user = new User(1L, "João Silva", "joaosilva@email.com", "123456789");
        userRepository.save(user);

        Post post1 = new Post(1L, "Meu Primeiro Post", "Este é meu primeiro post");
        Post post2 = new Post(2L, "Outro Post", "Este é outro post que eu escrevi");

        postRepository.save(post1);
        postRepository.save(post2);
    }

    // ... testes para métodos do PostController ...
}
```

Figura 5. Captura de tela que ilustra a resposta gerada pela ferramenta Gemini da Google, Observa-se como a ferramenta apresentou dificuldades, mesmo após o pedido fragmentado, para a geração das classes de testes.

## 5.4. Estruturação dos projetos

Após a geração das classes e trechos de código pelas três ferramentas de IAGs, a etapa seguinte do estudo envolveu a organização desses códigos dentro de um projeto Spring Boot. Vale ressaltar que as ferramentas de IAGs utilizadas neste estudo não possuem a capacidade ou o objetivo de gerar toda a estrutura de um projeto utilizando um *framework* específico como o Spring Boot. Assim, para criar a estrutura inicial do projeto, foi utilizado o Spring Initializr, uma ferramenta amplamente reconhecida que facilita a criação ágil de projetos Java com o uso do Spring. O Spring Initializr permite configurar o projeto de forma personalizada, selecionando dependências, a versão do Java, o tipo de empacotamento (Maven ou Gradle), entre outras opções [Nicoll et al. 2024].

Para este estudo, foram utilizadas as seguintes dependências: Spring Web, MySQL Driver, Spring Data JPA e Spring DevTools, que são essenciais para a operação da API, o acesso ao banco de dados e a configuração do projeto. A (Figura 6) ilustra um exemplo da interface do Spring Initializr em sua versão online. Todas essas configurações foram aplicadas uniformemente na estruturação dos três projetos analisados.

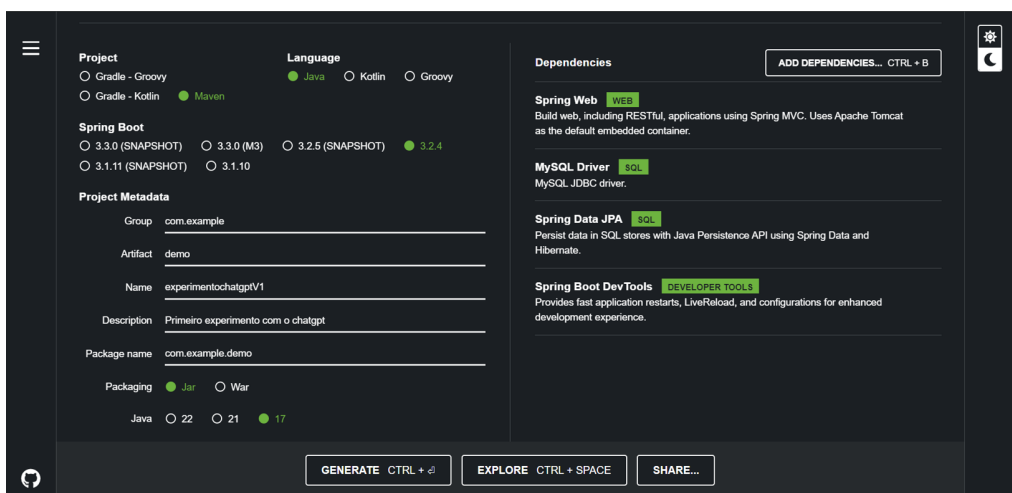


Figura 6. Captura de tela da configuração inicial do projeto Spring por meio da ferramenta Spring Initializr.

Com os trechos de código gerados pelas ferramentas de IAGs e a estrutura básica do projeto obtida por meio do Spring Initializr, a próxima etapa consistiu em integrar esses códigos à estrutura do projeto utilizando a IDE IntelliJ IDEA. O Spring Initializr foi configurado para gerar o esqueleto do projeto Spring Boot, incluindo as dependências. Com essa base gerada, o IntelliJ foi utilizado como ambiente de desenvolvimento para implementar os trechos de código dentro dessa estrutura. O processo envolveu importar o projeto gerado pelo Spring Initializr para o IntelliJ e, em seguida, incorporar manualmente os trechos de código nas classes e pacotes correspondentes. Isso incluiu a criação de controladores, repositórios, entidades, alinhando cada parte do código gerado pelas IAGs com as convenções e boas práticas do Spring Boot.

Na (Figura 7), pode-se visualizar a interface do IntelliJ IDEA durante o processo de integração dos códigos à estrutura do projeto. Após essa integração, os projetos estavam prontos para serem executados localmente e, posteriormente, submetidos a ferramen-

tas de análise de qualidade de código, para avaliar a manutenibilidade, coesão, segurança e outros aspectos relacionados à qualidade dos trechos gerados.

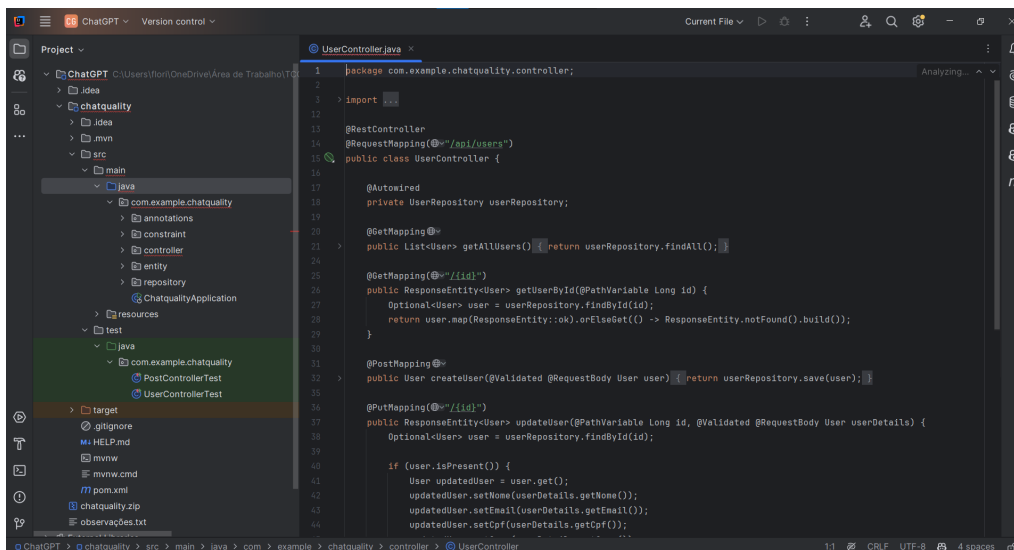


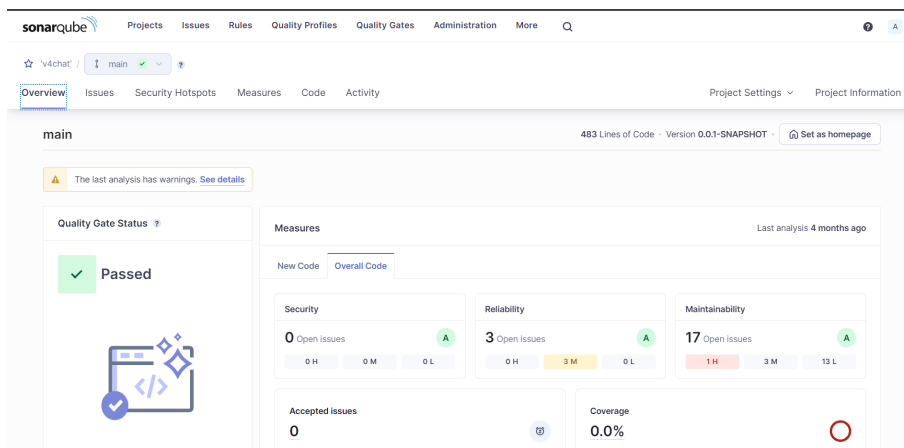
Figura 7. Captura de tela da estruturação de um dos projetos na IDE IntelliJ.

## 5.5. Análise estática

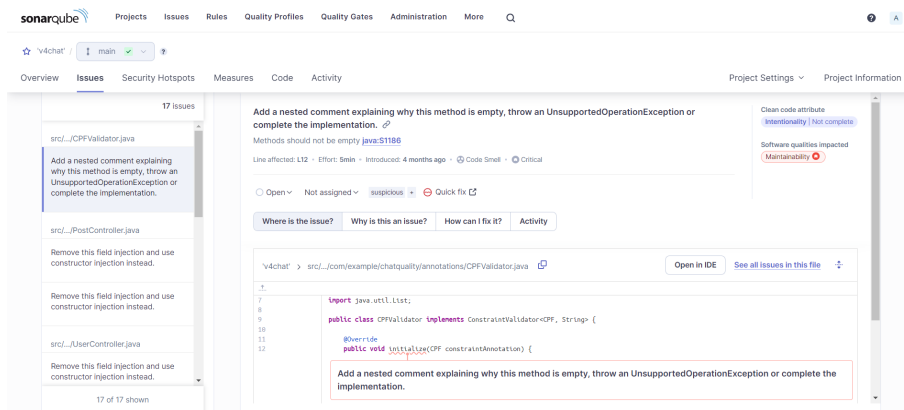
Com os projetos devidamente estruturados, o passo seguinte foi realizar a análise de qualidade dos códigos gerados. Para isso, os três projetos foram submetidos à análise estática utilizando a ferramenta SonarQube, que categoriza os problemas em três áreas principais: *Security* (Segurança), *Reliability* (Confiabilidade) e *Maintainability* (Manutenibilidade). Os problemas são identificados com base em regras pré-configuradas que funcionam como perguntas aplicadas aos trechos de código. Por exemplo: "Este código pode ser explorado por um invasor?" Se a resposta for afirmativa, trata-se de uma vulnerabilidade; caso contrário, novas questões são feitas para enquadrar o problema em uma das categorias da ferramenta, as regras são divididas em quatro grupos *Bugs*, *Vulnerabilities*, *Security Hotspots* ou *Code Smells*. Cada uma dessas categorias recebe uma classificação de qualidade que varia de A (melhor) a E (pior) [SonarSource 2024].

Adicionalmente, os problemas identificados são classificados por níveis de criticidade, seguindo uma escala crescente: *Low* (Baixo), *Medium* (Médio) e *High* (Alto). A gravidade de cada problema é determinada pelo impacto que ele pode causar nas qualidades do *software*, muitas vezes relacionado ao não cumprimento de boas práticas recomendadas [SonarSource 2024]. Essa subdivisão detalhada fornece uma visão clara sobre o impacto de cada questão, facilitando tanto a avaliação dos riscos quanto a identificação das áreas que mais afetam a qualidade do código.

A estrutura de classificação do SonarQube oferece tanto uma visão geral quanto um detalhamento específico dos pontos levantados. Para cada alerta gerado, a ferramenta apresenta a natureza do problema, sua localização no código e possíveis soluções. Essa abordagem, que combina uma visão macro e micro dos problemas, contribui para uma análise mais eficiente e direcionada, conforme ilustrado nas figuras 8 e 9.



**Figura 8. Captura de tela da ferramenta SonarQube, apresentando uma visão geral dos resultados obtidos pela análise estática.**



**Figura 9. Captura de tela da ferramenta SonarQube, apresentando um detalhamento de um dos alertas indicados.**

## 5.6. Análise dos resultados

Os dados gerados pelo SonarQube foram organizados e analisados para obter conclusões sobre a qualidade dos códigos gerados pelas IAGs. A análise permitiu identificar pontos fracos e áreas de melhoria das ferramentas.

Na Tabela 2, temos uma visão geral dos problemas apontados pelo SonarQube. Observa-se que a maioria dos alertas está relacionada parcial ou completamente à manutenibilidade do código, e não foram identificados problemas de segurança em nenhum dos projetos analisados.

Tanto o ChatGPT quanto o Copilot apresentaram resultados semelhantes, com a maioria dos problemas sendo de baixo impacto para a manutenibilidade. No entanto, o Gemini teve uma maior quantidade de alertas com impactos médios ou altos, o que indica uma maior necessidade de melhorias. É importante destacar que o Gemini enfrentou dificuldades na geração de testes, o que é evidenciado por um dos alertas de alto impacto, que está diretamente relacionado à falta de testes adequados.

Na comparação entre as ferramentas, o ChatGPT apresentou o maior número de ocorrências relacionadas ao uso desnecessário do modificador *public*, com 13 casos. Em-

bora tenha identificado menos problemas críticos de implementação, houve um erro significativo envolvendo métodos vazios, o que ressalta uma vulnerabilidade pontual. Já o Gemini apresentou uma maior diversidade de problemas, incluindo o uso inadequado de expressões regulares e a inclusão de *imports* não utilizados. Um ponto que se destacou foi a recomendação sobre a sintaxe de uma expressão regular usada para a identificação de números de documentos, sugerindo que a ferramenta pode gerar código mais específico e técnico. O Copilot, por sua vez, teve um perfil muito semelhante ao do ChatGPT, mas com menos ocorrências do modificador *public* desnecessário e um problema adicional relacionado ao uso de tipos genéricos.

Em relação à repetição de problemas, a injeção de dependência via campo foi identificada em todas as três ferramentas, indicando uma tendência de utilizarem essa abordagem em vez de construtores, o que pode impactar negativamente a confiabilidade e a manutenibilidade do código gerado. Além disso, tanto o ChatGPT quanto o Copilot foram sugeridos a remover o modificador *public* em diversas ocasiões (13 ocorrências no ChatGPT e 12 no Copilot). Esse problema comum aponta para uma falta de encapsulamento adequado, o que expõe funcionalidades internas de maneira desnecessária.

**Tabela 2. Visão geral dos resultados obtidos pela análise estática realizada com a ferramenta SonarQube.**

	Descrição	Quantidade	Impacto	Categoria
Código Gerado pelo ChatGPT	Remova esta injeção de campo e use injeção de construtor em seu lugar	6	Médio	Confiabilidade e Manutenibilidade
	Adicione um comentário aninhado explicando por que esse método está vazio, gere uma <code>UnsupportedOperationException</code> ou conclua a implementação.	1	Alto	Manutenibilidade
	Remova este modificador 'public'.	13	Baixo	Manutenibilidade
Código Gerado pelo Gemini	Descrição	Quantidade	Impacto	Categoria
	Remova esta injeção de campo e use injeção de construtor em seu lugar.	3	Médio	Confiabilidade e Manutenibilidade
	Use a sintaxe de classe de caracteres concisa <code>"\D"</code> em vez de <code>"[^\d-]"</code> .	1	Baixo	Manutenibilidade
	Remova esta injeção de campo e use injeção de construtor em seu lugar.	3	Médio	Manutenibilidade
	Remova o uso do tipo curinga genérico.	2	Alto	Manutenibilidade
	Remova esta importação não utilizada <code>'java.util.Date'</code> .	1	Baixo	Manutenibilidade
	Adicione pelo menos uma afirmação a este caso de teste.	1	Alto	Manutenibilidade
Código Gerado pelo Copilot	Descrição	Quantidade	Impacto	Categoria
	Remova esta injeção de campo e use injeção de construtor em seu lugar.	4	Médio	Confiabilidade e Manutenibilidade
	Adicione um comentário aninhado explicando por que esse método está vazio, gere uma <code>UnsupportedOperationException</code> ou conclua a implementação.	1	Alto	Manutenibilidade
	Remova o uso do tipo curinga genérico.	2	Alto	Manutenibilidade
	Remova este modificador 'public'.	12	Baixo	Manutenibilidade

## 6. Conclusão

Este estudo de caso teve como objetivo avaliar a qualidade dos códigos gerados por ferramentas de IAGs amplamente utilizadas atualmente. Os resultados revelaram que, no cenário analisado, essas ferramentas enfrentaram dificuldades na geração de códigos com boa qualidade, especialmente no quesito de manutenibilidade. Com frequência, práticas recomendadas de encapsulamento foram quebradas, o que resultou em exposição desnecessária de funcionalidades internas.

Embora a maioria dos códigos gerados fosse funcional, a preocupação inicial deste estudo, o uso dessas ferramentas por profissionais com pouca ou nenhuma experiência, torna-se ainda mais preocupante. A produção de códigos que, apesar de funcionarem,

apresentam falhas de qualidade pode gerar complicações em ambientes de produção, aumentando o risco de problemas a longo prazo.

Por outro lado, um ponto positivo foi a ausência total de alertas de segurança no contexto delimitado por este estudo, além da baixa quantidade de alertas relacionados à confiabilidade. Esses resultados sugerem que, ao menos nesses aspectos, os códigos gerados apresentaram um bom nível de robustez.

Para trabalhos futuros, sugere-se uma análise mais aprofundada sobre a capacidade de autocorreção dessas ferramentas, avaliando como elas podem aprimorar os códigos gerados a partir de *feedbacks* sobre erros e melhorias sugeridas. Além disso, seria interessante investigar o impacto de técnicas como a engenharia de *prompt*, o processo de formular instruções precisas para direcionar os sistemas de IA na geração de códigos mais apropriados e adequados ao contexto do projeto, garantindo assim maior funcionalidade e correção [Microsoft 2024].

Este trabalho possibilitou a aplicação prática de uma ampla gama de conhecimentos adquiridos ao longo do Curso Superior de Análise e Desenvolvimento de Sistemas. Entre as disciplinas que contribuíram para esse desenvolvimento, destacam-se: Algoritmos e Programação, Comunicação e Expressão, Linguagem de Programação I e II, Engenharia de *Software*, Banco de Dados I e II, Análise Orientada a Objetos, Arquitetura de *Software*, Metodologia de Pesquisa Científica e Tecnológica, Desenvolvimento *Web*, Desenvolvimento de Sistemas *Web*, Qualidade de *Software*.

## Referências

- Alqaradaghi, M. and Kozsik, T. (2024). Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code. *IEEE Access*, 12:55824–55842.
- Bateman, T. S. and Snell, S. A. (1998). *Administração Construindo Vantagem Competitiva*. Atlas S.A., 1th edition.
- GitHub (2024). Github copilot · seu programador de par de ia. Acessado em: 12 de fevereiro de 2024.
- Gozalo-Brizuela, R. and Garrido-Merchan, E. C. (2023). Chatgpt is not all you need. a state of the art review of large generative ai models. *arXiv*.
- Kabir, S., Udo-Imeh, D. N., Kou, B., and Zhang, T. (2024). Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions. *Nome do Jornal*, 1:1–17.
- Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., and Lo, D. (2024). Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*, 33(5).
- Microsoft (2024). Fundamentos de engenharia e melhores práticas de prompts - training — microsoft learn. Acessado em: 10 de Julho de 2024.
- Nicoll, S., Syer, D., and Bhave, M. (2024). Spring initializr reference guide. Acessado em: 22 de fevereiro de 2024.
- OpenAI (2022). Introducing chatgpt — openai. Acessado em: 12 de fevereiro de 2024.

- Pichai, S. and Hassabis, D. (2023). Apresentando gemini: o modelo de ia mais capaz do google até agora. Acessado em: 12 de fevereiro de 2024.
- Pressman, R. (2016). *Engenharia de Software*. AMGH editora Ltda., 8th edition.
- Sommerville, I. (2018). *Engenharia de Software*. Pearson, 10th edition.
- SonarSource (2024). Sonarqube. Acessado em: 12 de fevereiro de 2024.
- Spring (2024). Spring — why spring. Acessado em: 22 de fevereiro de 2024.
- UBS (2023). Chatgpt: O que é, seu impacto e oportunidades — ubs assuntos globais. Acessado em: 15 de fevereiro de 2024.
- Wang, Y.-C., Xue, J., Wei, C., and Kuo, C. C. J. (2023). An overview on generative ai at scale with edge–cloud computing. *IEEE Open Journal of the Communications Society*, 4:2952–2971.



# Documento Digitalizado Público

## Artigo - Versão Final do TCC

**Assunto:** Artigo - Versão Final do TCC  
**Assinado por:** Fernando Sambinelli  
**Tipo do Documento:** Formulário  
**Situação:** Finalizado  
**Nível de Acesso:** Público  
**Tipo do Conferência:** Documento Digital

Documento assinado eletronicamente por:

- **Fernando Sambinelli, PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 13/12/2024 17:39:55.

Este documento foi armazenado no SUAP em 13/12/2024. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse <https://suap.ifsp.edu.br/verificar-documento-externo/> e forneça os dados abaixo:

**Código Verificador:** 1882239

**Código de Autenticação:** 062988a6a8

