UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ARTHUR SILVA LIMA GUEDES

UM ESTUDO DE CASO PARA AVALIAR O IMPACTO DO USO DE CODE REVIEW EM ASPECTOS DE QUALIDADE DE UM SISTEMA REAL

CAMPINA GRANDE - PB

2023

ARTHUR SILVA LIMA GUEDES

UM ESTUDO DE CASO PARA AVALIAR O IMPACTO DO USO DE CODE REVIEW EM ASPECTOS DE QUALIDADE DE UM SISTEMA REAL

Trabalho de Conclusão de Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Everton Leandro Galdino Alves.

CAMPINA GRANDE - PB

2023

ARTHUR SILVA LIMA GUEDES

UM ESTUDO DE CASO PARA AVALIAR O IMPACTO DO USO DE CODE REVIEW EM ASPECTOS DE QUALIDADE DE UM SISTEMA REAL

Trabalho de Conclusão de Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Everton Leandro Galdino Alves

Orientador – UASC/CEEI/UFCG

Professora Dr. Fábio Jorge Almeida Morais

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 14 de fevereiro de 2023.

CAMPINA GRANDE - PB

ABSTRACT

Software development is usually guided by a set of activities/phases for the product to reach the expected goal. During these phases, good practices aimed at maintenance and quality assurance, such as code review, are essential, even more so in scenarios of rapid growth and product approval, which require agile development. This work consists of carrying out a case study to evaluate the impact on code quality aspects of a real system at Hospital Israelita Albert Einstein, after the team introduced code review activities in its development process. For this, quantitative (e.g., number of bugs) and qualitative (e.g., reliability) metrics were used, comparing two moments of the project, pre and post the use of code review. In general, we found a tendency to reduce bugs and improve code quality. The main characteristics of the discussions between author and reviewer have changed and, unanimously, developers believe in the positive impact of code review on the project.

Um Estudo de Caso para Avaliar o Impacto do Uso de Code Review em Aspectos de Qualidade de um Sistema Real

Arthur Silva Lima Guedes
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
arthur.guedes@ccc.ufcg.edu.br

Everton Leandro Galdino Alves
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
everton@computacao.ufcg.edu.br

RESUMO

O desenvolvimento de software normalmente é guiado por um conjunto de atividades/fases para que o produto atinja o objetivo esperado. Durante essas fases, boas práticas voltadas para a manutenção e garantia da qualidade, como o *code review*, são imprescindíveis, ainda mais em cenários de rápido crescimento e aprovação do produto, que requerem um desenvolvimento ágil. Este trabalho consiste na realização de um estudo de caso para avaliar o impacto em aspectos de qualidade do código de um sistema real do Hospital Israelita Albert Einstein, após a equipe introduzir atividades de *code review* no seu processo de desenvolvimento. Para tal, foram usadas métricas quantitativas (e.g., número de *bugs*) e qualitativas (e.g., confiabilidade), comparando dois momentos do projeto, pré e pós o uso de *code review*. De modo geral, encontramos uma tendência à redução de *bugs* e melhoria na qualidade do código. As características majoritárias das discussões entre autor e revisor mudaram e, de maneira unânime, os desenvolvedores acreditam no impacto positivo do *code review* no projeto.

Palavras-chave

boas práticas, garantia da qualidade, *code review*, desenvolvimento ágil, análises, aspectos qualitativos, métricas.

1. INTRODUÇÃO

Cada vez mais acompanhamos o rápido crescimento e consolidação de aplicações software em diferentes ramos (e.g., alimentício, mobilidade), bem como o surgimento de novas tecnologias. As oportunidades e viabilidades de melhorias, percebidas pelas fábricas de software, resultam na satisfação do usuário final, consequência da qualidade do produto desenvolvido. Em contraponto, defeitos de software podem acarretar danos imensuráveis, além da frustração dos seus usuários. Desse modo, aspectos como escopo, custo e tempo devem estar equilibrados para que o software seja de qualidade, e essa, por sua vez, nunca deve ser sacrificada [14].

Em contextos de rápida expansão e aprovação dos produtos de software, o desenvolvimento ágil torna-se necessário. De acordo com os princípios do Manifesto Ágil [1], a prioridade máxima é a satisfação do cliente, por meio da entrega antecipada e contínua de um software com valor. Portanto, mirando satisfazer as necessidades do cliente por meio de boas práticas ágeis (e.g., entrega contínua), a qualidade do software deve ser garantida. Para isso, ao longo do processo de desenvolvimento, vários métodos podem ser implementados, dentre os quais destacamos o *Code Review* [7]. Nesse processo, o autor submete uma modificação na *codebase* do projeto, a qual deve ser avaliada por um ou mais revisores. Durante o *code review*, possíveis *bugs* podem ser identificados, melhorias podem ser propostas e outras discussões relevantes podem surgir, o que tende a fomentar a garantia e manutenção da qualidade do projeto.

Cenários de crescimento exponencial de um software podem representar também riscos ao mesmo, caso sua qualidade não seja bem gerida. Um dos riscos a ser considerado é a ocorrência de *bugs* em ambientes produtivos (também chamados de incidentes ao longo dessa pesquisa). Times de desenvolvimento ágil são movidos por entregas incrementais e de valor para o cliente, em determinado período de tempo (*sprint*). Esse requisito faz com que a necessidade de um desenvolvimento ágil e efetivo, em contraposição com a manutenção da qualidade e das boas práticas, surja como um possível *trade-off*. Esse, por sua vez, cria uma ideia de que requisitos como entregas ágeis e qualidade estão em lados opostos, quando, na verdade, deveriam caminhar em sincronia. Em um cenário similar, podemos destacar o sistema real que servirá como base do estudo de caso desta pesquisa. O Hospital Israelita Albert Einstein (HIAE), considerado o melhor hospital do Brasil e da América Latina [2], atualmente desenvolve soluções de software com o objetivo de revolucionar a jornada médica do paciente. O impacto imediato dessas soluções resultou em um considerável crescimento no número de agendamentos de consultas. E, para que mais funcionalidades e estabilidade pudessem ser entregues, as *sprints* recheadas de incidentes (uma das maiores dores do time) precisavam acabar. Nesse cenário, durante as cerimônias de retrospectiva, o *code review* começou a ganhar enfoque, surgindo como uma oportunidade de garantia da qualidade do software.

Sendo assim, nossa pesquisa tem como objetivo avaliar o impacto de um processo sistemático e bem definido de *code review* na qualidade de um dos sistemas mais recentes e de grande valor desenvolvido por um dos times do HIAE.

Para tal, conduzimos um estudo empírico com base em métricas quantitativas (e.g., número de *bugs*) e qualitativas (e.g., confiabilidade), comparando os resultados em um "antes" e "depois" da introdução do *code review* no projeto. Nossos resultados sugerem, de modo geral, uma tendência à redução de *bugs*, assim como melhoria na qualidade do código. Ainda, encontramos uma mudança na postura dos autores/revisores durante os *reviews*, bem como um consenso, entre os

desenvolvedores, em relação à importância e ao impacto positivo do *code review* no projeto.

2. FUNDAMENTAÇÃO TEÓRICA

Inúmeras boas práticas podem ser usadas durante o desenvolvimento ágil para assegurar a qualidade do software. Esta seção apresenta conceitos fundamentais para este TCC.

2.1 Garantia de Qualidade

Segundo Sommerville [3], Garantia de Qualidade (QA, do inglês *Quality Assurance*) diz respeito aos processos e padrões que levam a produtos de alta qualidade, bem como a introdução destes no ciclo de desenvolvimento. Ainda, segundo o autor, na maioria das empresas, a equipe de QA é responsável por um importante aspecto da qualidade de software: o processo de teste de *release*, ou seja, testes executados antes do software ser disponibilizado para o cliente, como acontece no contexto dos projetos do HIAE. Isso é fundamental para garantir que a aplicação esteja adequada aos requisitos e padrões esperados, além de ser importante aliado em processos ágeis como a Entrega Contínua.

2.2 Integração e Entrega Contínua

Integração Contínua (CI, do inglês *Continuous Integration*) e Entrega Contínua (CD, do inglês *Continuous Delivery*) são boas práticas pertencentes ao mundo DevOps e também ao desenvolvimento ágil.

2.2.1 Integração Contínua

De acordo com Fowler [4], CI é uma prática por meio da qual os membros dos times de desenvolvimento integram seus trabalhos constantemente, resultando em múltiplas integrações diárias, cada uma verificada e validada por um processo de *build* automatizado. Na prática, esse processo é traduzido, de forma geral, no seguinte fluxo: (1) desenvolvedor realiza o *commit* das alterações para o repositório; (2) um servidor específico é usado para realizar o *build* e possíveis outras tarefas (e.g., testes, análise estática do código). Evidenciamos a importância desse processo, uma vez que favorece à redução dos problemas de integração de código, haja visto que eventuais erros podem ser identificados e corrigidos o mais rápido possível.

2.2.2 Entrega Contínua

Fowler [5] define a Entrega Contínua como uma "disciplina" de desenvolvimento, por meio da qual construímos o software de modo que possa ser implantado em produção a qualquer momento. Segundo o autor, alcançamos a entrega contínua através da integração contínua. Fowler [6] também menciona o conceito de *Deployment Pipeline,* um conjunto de estágios que compõem o processo de *build* automatizado. Essa estratégia, inclusive, é usada nos projetos do HIAE, para que as *features* atinjam ambientes não produtivos (DEV, QAs, UAT) e produtivos (PROD).

2.3 Code Review Moderno

Bacchelli e Bird [7] definem o processo de *code review* moderno como uma revisão informal, baseada em ferramentas e que ocorre regularmente na prática. De acordo com os autores, atualmente, muitas organizações estão adotando um processo de revisão mais leve, objetivando limitar as ineficiências das clássicas inspeções de código. De maneira geral, o *code review* é uma boa prática do desenvolvimento de software. Para que esse processo aconteça, o desenvolvedor/autor submete seu código para revisão de um ou mais revisores, que, por sua vez, fornecem *feedbacks* para as alterações submetidas. A partir dos *feedbacks* recebidos, o autor realiza possíveis correções/melhorias e/ou completa a integração do código (o mesmo pode, também, iniciar uma *thread* de discussão com o revisor). Um processo de revisão forte e bem alinhado entre o time de desenvolvimento é um ponto chave na garantia da qualidade do produto.

3. OBJETIVO E QUESTÕES DE PESQUISA

Essa seção descreve o objetivo da nossa pesquisa e as Questões de Pesquisa que serão usadas para atingir esse objetivo.

3.1 Questões de Pesquisa

O objetivo principal desta pesquisa é avaliar a importância e o impacto do uso de um processo bem definido e consolidado de *code review* no contexto de um projeto real. Para tal, serão avaliados aspectos quantitativos e qualitativos. Nesse sentido, buscamos responder às seguintes questões de pesquisa:

QP_1: A utilização do *code review* levou a redução do número de incidentes no sistema estudado?

Com essa questão de pesquisa, estamos interessados em entender como uma de nossas métricas quantitativas (número de incidentes [1]) se comporta frente à implementação de uma política de *code review* durante as *sprints* do time de desenvolvimento. Aqui, tentamos encontrar uma primeira evidência de um possível impacto (seja este positivo ou negativo) do uso de *code review* no código do sistema.

QP_2: A utilização do *code review* levou a melhoria da qualidade do código no sistema estudado?

Um dos nossos focos, também, é analisar o impacto do *code review* em aspectos de qualidade do código. Sendo assim, buscamos avaliar a evolução de métricas quantitativas (*bugs*[2] e *code smells*[3]) no projeto, relacionando a mesma com o processo de *code review*. Vale ressaltar que, em nossa pesquisa, o termo incidente refere-se a *bugs* em produção, enquanto que o termo *bug* é usado, prioritariamente, como métrica da análise estática do código. Entendemos essa questão de pesquisa como uma alternativa para mensurar o impacto do *code review*, agora com um olhar voltado para a qualidade do código produzido.

QP_3: A utilização do *code review* levou a equipe a discutir aspectos importantes para o projeto?

Uma outra questão interessante está relacionada aos *feedbacks* fornecidos durante o processo de *review*. Aqui, estamos interessados em entender o quão significativas foram as discussões entre autor e revisor. Sendo assim, buscamos classificar os comentários coletados para então entendermos os temas mais recorrentes durante os *reviews*. É importante ressaltar que estas discussões vão muito além da prevenção de *bugs* e/ou problemas similares.

QP_4: Qual a perspectiva dos desenvolvedores sobre o uso de *code review* no projeto?

---

[1] No contexto do projeto, o termo "incidente" é usado para referenciar *bugs* no ambiente produtivo.

[2] Erro de codificação que pode ocasionar falhas e/ou comportamentos inesperados no sistema.

[3] Problemas, também de codificação, que podem dificultar a legibilidade, bem como a manutenibilidade do código.

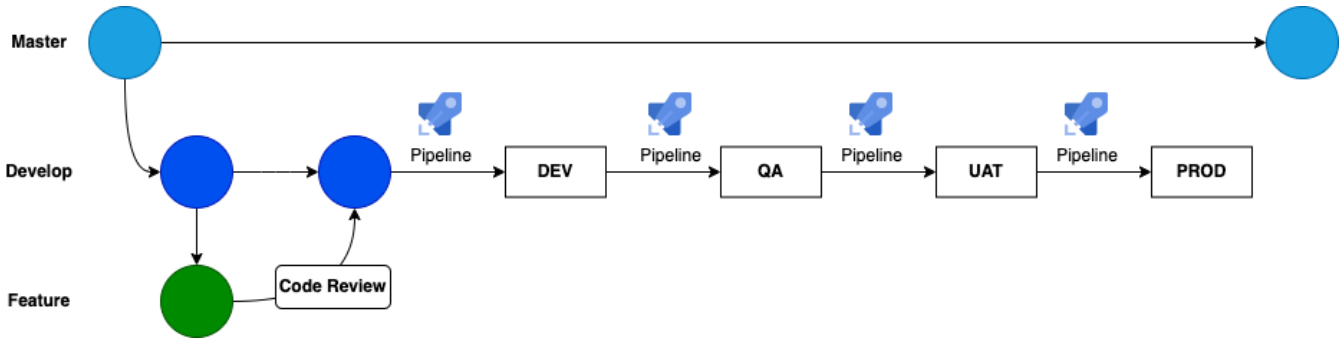

Figura 1: Fluxo de integração e entrega.

Por fim, em nossa última questão de pesquisa, nós estamos interessados em entender como os desenvolvedores (envolvidos com o projeto durante a pesquisa) percebem o processo de *code review* aplicado. Essa questão é fundamental para nossas considerações finais, uma vez que *feedbacks* dos próprios desenvolvedores podem revelar resultados adicionais. Aqui, focamos em entender aspectos como importância, satisfação e melhorias percebidas.

4. METODOLOGIA

Para compreender os impactos do *code review* em aspectos quantitativos e qualitativos, escolhemos um dos sistemas recentes e de grande valor desenvolvido por um dos times do HIAE. O projeto em questão utiliza *Scrum* como metodologia de desenvolvimento ágil. Apesar de repositórios distintos para *backend* e *frontend*, ambos possuem TypeScript como linguagem de programação predominante; em relação ao tamanho do sistema, o *backend* possui em torno de 12.000 linhas de código fonte e usa o *framework* NestJS[4], enquanto que o *frontend* possui em torno de 15.000 linhas e usa a famosa biblioteca React[5]. Além disso, o tamanho da *squad* (considerando desenvolvedores e QA) responsável pelo sistema variou entre 6 e 5 pessoas durante as *sprints* analisadas nesta pesquisa. Quando composta por 6 pessoas, a *squad* tinha a seguinte distribuição: 2 desenvolvedores *frontend*, 2 desenvolvedores *full stack*, 1 desenvolvedor *backend* e 1 QA. Quando composta por 5 pessoas, a seguinte distribuição foi utilizada: 1 desenvolvedor *frontend*, 2 desenvolvedores *full stack*, 1 desenvolvedor *backend* e 1 QA. Durante o período da pesquisa, o processo de *code review* fazia parte do fluxo exemplificado na Figura 1.

Considerando o desenvolvimento de determinado item do *backlog* da *sprint*, os desenvolvedores criam novas *branches* (*feature branches*) a partir da *branch* de desenvolvimento (*develop*), e nessas implementam as novas funcionalidades. Ao final da implementação e utilizando o ferramental do Azure DevOps[6], o desenvolvedor, que agora podemos considerar autor, cria um novo *Pull Request* (PR) e então submete o código produzido para revisão: inicia-se o processo de *code review* (Figura 2). É importante ressaltar que, durante a pesquisa, o *code review* sempre esteve presente. Entretanto, em um dos momentos, o qual denominamos de pré-*review*, o *review* caracterizava-se por ser um processo informal e burocrático; uma pré-condição para o *complete* dos PRs, que, por muitas vezes, era burlada (em muitos casos, o *code review* era inexistente e os PRs aprovados sem nenhum tipo de validação). Após obter as aprovações necessárias, o PR é completado e então a *branch develop* atualizada com as alterações mais recentes. Nesse momento, o *pipeline* (CI/CD) é acionado e então os ambientes podem ser atualizados. Por padrão, inicialmente o *pipeline* atualiza apenas o ambiente de desenvolvimento (DEV), sendo este usado para testes pelos próprios desenvolvedores. Após aprovação, o pipeline pode avançar e então fazer a implantação no ambiente de QA, a partir do qual o engenheiro de QA executa os cenários de testes. Caso nenhum problema seja detectado na *feature* recém-implementada, o *pipeline* avança até o ambiente de UAT (do inglês *User Acceptance Testing*); caso contrário, o *pipeline* não avança até que os defeitos apontados sejam corrigidos (sendo necessário refazer o processo até esse *step*). Em UAT, a validação é feita por potenciais usuários reais; trata-se de um ambiente destinado para testes da área de negócio. Caso a *feature* seja aprovada, então o item que a representa no *backlog* é movido para uma raia específica (no quadro *scrum*), que indica que o mesmo está pronto para *deploy*. Durante a realização desta pesquisa, os *deploys* eram feitos, comumente, após a finalização de cada *sprint*.

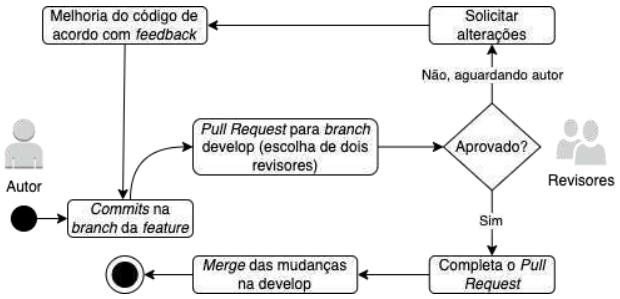

Figura 2: Processo detalhado de *code review*.

A Figura 2 mostra, em detalhes, o processo de *code review* mencionado. Reforçamos que esse processo não era constante no pré-*review*, sendo apenas executado de maneira precisa no momento que, nesta pesquisa, denominamos de pós-*review* (após a introdução sistemática do uso de *code review*). Após realizar o *commit* das alterações na *branch feature*, o autor cria um *Pull Request* e então escolhe dois revisores. Geralmente, essa escolha é baseada na familiaridade dos revisores com o código fonte, além de levar em consideração a preferência destes por *backend* e/ou *frontend*. Após isso, os revisores conseguem analisar o *diff* das mudanças propostas e então realizar o *review*, utilizando o próprio Azure Repos. Nesse ponto, melhorias podem ser propostas, *bugs*

[4] Disponível em: <https://nestjs.com/>. Acesso em: 20 de jan. de 2023.

[5] Disponível em: <https://reactjs.org/>. Acesso em: 20 de jan. de 2023.

[6] Disponível em: <https://azure.microsoft.com/en-us/products/devops>. Acesso em: 20 de jan. de 2023.

podem ser apontados e outras discussões relevantes podem surgir. É importante ressaltar que o autor pode responder a cada um dos comentários feitos, concordando ou não (inicia-se uma *thread*/discussão). Após finalizar a revisão, o revisor atualiza seu *status* para o PR, podendo ser:

- **Aprovado:** Quando o revisor julga que o código fonte submetido está apto para entrar na *branch* de desenvolvimento. Após a aprovação dos dois revisores, o PR pode ser completado e então as mudanças são mescladas com o código que já está na *branch* de desenvolvimento.
- **Aprovado com sugestões:** Similar ao *status* descrito acima, porém, o revisor pode deixar alguma sugestão não impeditiva. Essa sugestão pode ter, também, um viés educativo.
- **Aguardando pelo autor:** Quando o revisor encontra possíveis erros e/ou faltas no código, bem como pontos de melhoria em geral (e.g., *design* da solução, manutenção de normas). Ele, então, propõe mudanças e/ou correções e indica que está aguardando pelo autor, que por sua vez, consegue analisar os comentários e então realizar as alterações necessárias (caso não haja concordância com o *feedback*, o autor pode aprofundar a discussão com o revisor).
- **Rejeitado:** Quando o revisor julga que o código fonte submetido não está apto para ser mesclado na *branch* de desenvolvimento. Duas rejeições é o suficiente para que o PR seja descartado. É importante salientar que, apesar de descrevermos esse *status* aqui, o mesmo não foi utilizado durante o tempo da pesquisa.

4.1 Coleta de Dados

Para respondermos às nossas questões de pesquisa, coletamos dados quantitativos e qualitativos, resultantes do processo de desenvolvimento do sistema estudado. Esses incluem informações da "saúde" do projeto em diferentes momentos. Para recuperar parte dos dados (e.g., incidentes), usamos, prioritariamente, o Azure Boards e o Azure Repos, ferramentas do próprio Azure DevOps. Para coletar os dados referentes à qualidade do código desenvolvido, utilizamos o SonarQube[7] como ferramenta de análise estática, o que nos possibilitou uma visão geral da saúde do projeto em diferentes e estratégicos momentos do período de realização da pesquisa. Usamos, especificamente, as métricas de manutenibilidade e confiabilidade, calculadas com base no número de *code smells* e *bugs*, respectivamente.

4.1.1 Recuperando histórico de incidentes

Utilizando o Azure Boards, navegamos pelos *backlogs* de algumas *sprints* do projeto (no contexto do projeto, cada *sprint* durava 2 semanas). A escolha das *sprints* baseou-se, estrategicamente, em uma sprint *pivot*, sendo esta a sprint na qual formalizou-se o processo de *code review*, que passaria a ser não mais uma simples etapa burocrática de aprovação de mudanças na *codebase*, mas sim um processo bem alinhado entre o time de desenvolvimento. É importante ressaltar que o processo foi evoluindo e sendo aperfeiçoado à medida que o time engajava-se mais. Considerando a sprint *pivot*, extraímos o número de incidentes de cada uma das 5 *sprints* anteriores (pré-*review*) e posteriores (pós-*review*), para tentarmos realizar uma análise comparativa entre os dados e então mensurar o impacto do *code review*. A Figura 3 representa o que descrevemos.

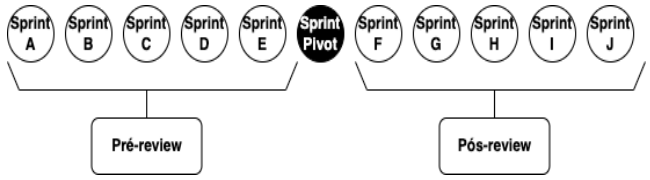

Figura 3: Representação da seleção de *sprints* para a pesquisa.

A identificação dos incidentes foi uma tarefa simples, uma vez que o Azure Board da squad responsável pelo sistema estudado já estava configurado com *cards* específicos para representar incidentes. Além disso, estes possuíam *tags* para sinalizar, também, se tratavam de problemas relacionados a *frontend* e/ou *backend*. Dessa forma, com as *sprints* definidas, coletamos e tabulamos o número de incidentes de cada *sprint*, segregando-os em *frontend*/*backend*. No caso estudado, em geral, acontecia um *deploy* após o final de cada *sprint* e então, incidentes poderiam ser mapeados na *sprint* seguinte (podendo estes estarem diretamente relacionados com as últimas funcionalidades implementadas ou não). Sempre que encontrados e registrados, incidentes entravam na *sprint* corrente com prioridade máxima.

4.1.2 Coletando métricas estáticas do código

Seguindo nossa proposta de realizar uma análise comparativa dos dados, coletamos também informações acerca da saúde do projeto. Aqui, nosso foco foi analisar a qualidade do código, utilizando o SonarQube. Através do Azure Repos, coletamos o histórico de *commits* da *codebase* e, com o auxílio do git, realizamos um *checkout* para pontos específicos da linha do tempo do projeto, sempre considerando a *branch master* (código em produção). Estrategicamente, as datas de início de cada *sprint* representaram os nossos pontos de *checkout*, uma vez que esperava-se que todas as alterações realizadas na *sprint* anterior já estivessem mescladas com a *branch master*. Para cada *checkout*, rodamos o SonarQube e coletamos o número de *bugs* e *code smells*, dados referentes às métricas qualitativas de confiabilidade e manutenibilidade, respectivamente.

4.1.3 Coletando comentários dos reviews

Ainda utilizando o Azure Repos, extraímos as discussões/comentários dos PRs abertos durante o período da pesquisa. Com esses dados, nosso objetivo foi avaliar se a definição de um processo de *code review* provocou mudanças na interação entre autor e revisor, haja visto que, no pré-*review*, essa interação era limitada (uma vez que o *code review* não acontecia de modo constante). Nesta etapa, ao coletarmos os comentários e aplicamos um modelo de *open coding* [12], considerando três possíveis categorias para cada comentário:

- **Manutenção de normas:** o *review* garantiu que normas (e.g., padrão de nomenclatura) continuem sendo seguidas.
- **Prevenção de acidentes:** o *review* foi fundamental para impedir a introdução de um novo *bug* no código.
- **Educação:** o *review* permitiu ao revisor/autor ensinar/aprender.

Desse modo, "etiquetamos" manualmente cada comentário com uma dessas categorias, buscando evidenciar possíveis mudanças

[7] Disponível em: <https://www.sonarsource.com/products/sonarqube/>. Acesso em: 20 de jan. de 2023.

no comportamento da relação entre autor e revisor, no pré e pós-*review*.

4.1.4 Survey

Finalmente, para responder nossa última Questão de Pesquisa, nós criamos um questionário online[8] para ser respondido pelo time responsável pelo desenvolvimento do projeto estudado. As perguntas foram elaboradas para que pudéssemos compreender as percepções dos desenvolvedores acerca do processo de *code review*. De modo geral, nossa *survey* foi composta por três questões com base na escala Likert [13], estas focadas em entender a frequência do processo no pré-*review*, bem como sua importância e o quão satisfeitos estavam os desenvolvedores no pós-*review*; duas questões de múltipla escolha para entendermos os principais ganhos que o pós-*review* trouxe, assim como os desafios ainda existentes; uma questão dicotômica para entendermos se os desenvolvedores perceberam melhorias no projeto com a formalização do *code review*; uma questão aberta para que os desenvolvedores pudessem explanar suas percepções acerca do *code review* no projeto. É importante ressaltar que todas as questões descritas acima, exceto a última (aberta), possuíam um campo de justificativa associado, o que nos permitiu extrair informações adicionais, com base na explicação fornecida. Ainda, o questionário contava com algumas outras questões iniciais, apenas para que pudéssemos traçar o perfil do desenvolvedor.

5. RESULTADOS

Nesta seção, discutimos os resultados obtidos baseando-se em cada uma de nossas Questões de Pesquisa, definidas na seção 3.1.

5.1 QP_1: A utilização do *code review* levou a redução do número de incidentes no sistema estudado?

Nossa pesquisa possui dois momentos bem definidos, entre os quais conseguimos realizar análises, prioritariamente, comparativas. O primeiro momento compreende o "antes" da consolidação de um processo bem definido de *code review* (pré-*review*). Nesse primeiro momento, o *code review* era apenas uma parte informal e burocrática do processo de desenvolvimento, visto apenas como uma pré-condição para o *complete* dos PRs, haja visto que esses precisavam de, pelo menos, duas aprovações. Sendo assim, no pré-*review*, o processo de revisão (e, por consequente, a interação entre autor e revisor, bem como o controle das alterações) era mínimo ou até mesmo inexistente e, portanto, compreende *sprints* mais propícias a problemas de qualidade (e.g., incidentes). O segundo momento, pós-*review*, caracteriza-se pela consolidação de um processo bem alinhado de *code review*, ainda que aberto à modificações e melhorias.

Baseando-se no pré e pós-*review* descritos acima e usando análises comparativas e quantitativas, encontramos uma tendência geral ao decrescimento do número de incidentes. No *frontend*, em quatro das *sprints* pré-*review* (⅘), o número de incidentes foi igual ou maior a 2; e em apenas uma (⅕), esse número foi igual a 1. Nas sprints pós-*review*, em apenas duas (⅖), o número de incidentes foi igual ou maior a 2 (redução de 50%); e, em três *sprints* (⅗), o número de incidentes foi igual a 1. De maneira geral, no *frontend*, o pré-*review* totaliza 12 incidentes, enquanto que no pós-*review*, esse número é reduzido para 8. No *backend*, a evolução é ainda mais evidente: no pré-*review*, em apenas uma *sprint* (⅕), não houve o registro de incidentes, enquanto que no pós-*review*, apenas nas duas *sprints* iniciais (⅖), incidentes foram mapeados.

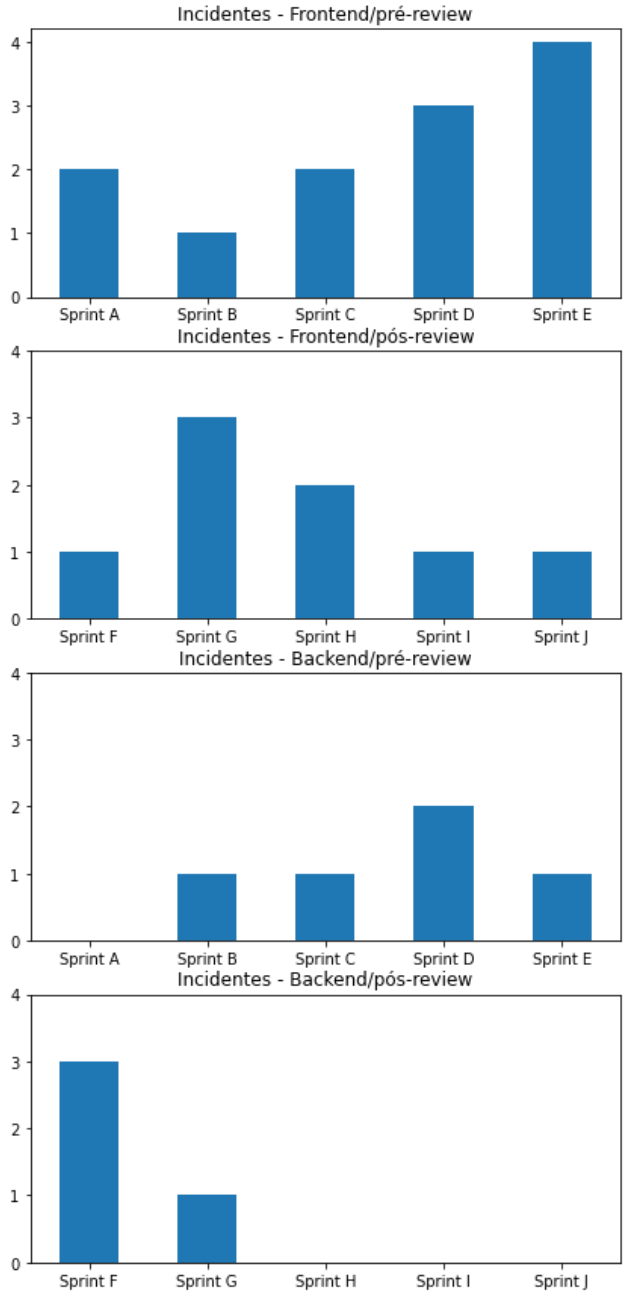

Figura 4: Evolução do número de incidentes no pré e pós-*review*.

A Figura 4 mostra a evolução descrita. Visualmente, é simples perceber a mencionada tendência geral à redução do número de incidentes. No entanto, ainda não é suficiente para afirmarmos que o *code review* é o responsável pelo comportamento observado. Os dados nos revelam a presença de alguns *"outliers"* e, com base nesses, as seguintes perguntas podem surgir:

[8] Disponível em: <https://drive.google.com/file/d/1q5_qEbjWUkOCTJR0WIhafbbVu6TlRciq/view>.

- Por que, na *sprint* G (pós-*review*, *frontend*), houve um aumento do número de incidentes?
- Por que, na *sprint* F (pós-*review*, *backend*), houve um aumento do número de incidentes?

Para tentarmos responder essas perguntas, primeiro, buscamos entender as alterações realizadas em cada *sprint*. Utilizando o git (especificamente o comando *diff*), geramos os dados tabelados abaixo:

Frontend	Arquivos Modificados	Inserções	Deleções
***Sprint* A**	60	3120	786
***Sprint* B**	105	5424	1464
***Sprint* C**	8	408	46
***Sprint* D**	88	1876	710
***Sprint* E**	12	150	105
***Sprint* F**	17	692	330
***Sprint* G**	23	750	363
***Sprint* H**	11	64	45
***Sprint* I**	10	69	44
***Sprint* J**	7	110	92

Tabela 1: Arquivos alterados, inserções e deleções por *sprint* analisada no *frontend*.

Backend	Arquivos Modificados	Inserções	Deleções
***Sprint* A**	32	2040	20
***Sprint* B**	26	660	53
***Sprint* C**	21	275	42
***Sprint* D**	41	751	69
***Sprint* E**	18	242	17
***Sprint* F**	23	449	128
***Sprint* G**	5	41	97
***Sprint* H**	6	120	22
***Sprint* I**	19	298	60
***Sprint* J**	25	475	180

Tabela 2: Arquivos alterados, inserções e deleções por *sprint* analisada no *backend*.

A partir desses dados, os resultados começam a fazer mais sentido. É possível observar que, a *sprint* G (pós-*review*, *frontend*), que apresenta um aumento do número de incidentes, possui os números mais expressivos da tabela, dentre todas as sprints do pós-*review*/*frontend*. Nessa *sprint*, 23 arquivos foram alterados, com 750 inserções e 363 deleções. Um cenário similar é evidenciado na *sprint* F (pós-*review*, *backend*), que também apresenta aumento nos incidentes. Nessa *sprint*, assim como na *sprint* G (pós-*review*, *frontend*), 23 arquivos foram alterados, agora com 449 inserções e 128 deleções, números extremamente próximos dos observados na *sprint* J (que detém os números mais expressivos dentre todas as *sprints* pós-*review*/*backend*). Dessa forma, podemos inferir que o alto número de modificações realizadas na *sprint* pode propiciar o surgimento de novos incidentes, como foi observado nas *sprints* G (*frontend*) e F (*backend*).

Resultado 1: Existe uma tendência à redução do número de incidentes no pós-review. Entretanto, outros fatores, como a quantidade de modificações realizadas na sprint, podem influenciar.

5.2 QP_2: A utilização do *code review* levou a melhoria da qualidade do código no sistema estudado?

Utilizando o SonarQube para análise estática do código, notamos um comportamento mais constante em relação ao número de *bugs* e parcialmente constante em relação ao número de *code smells*, no período pós-*review*. A Figura 5 mostra a evolução do número de *bugs* no *frontend* e *backend*.

Para ambas as aplicações (*backend* e *frontend*), no pré-*review*, evidenciamos uma tendência ao crescimento e surgimento de novos *bugs*. De acordo com os relatórios do SonarQube para as *sprints* que compreendem o pré-review, 3 e 2 novos *bugs* foram detectados no *frontend* e *backend*, respectivamente. Positivamente, no pós-*review*, o número de *bugs* se manteve constante para ambas as aplicações, sem nenhum novo *bug* detectado nesse período (ou seja, os *bugs* eram remanescentes das *sprints* pré-*review*).

Na Figura 6, observamos um comportamento distinto para os *code smells*. No *frontend*, o pré-*review* apresenta inconsistência e variância no número de *code smells*, com um pico na *sprint* C. Antes de qualquer outra análise, é importante relembrarmos que os pontos de *checkout* para a análise estática foram as datas de início das *sprints*, o que significa que resultados da *sprint* X podem ser influenciados por evoluções/modificações realizadas na *sprint* X - 1. Sabendo disso e buscando entender melhor o comportamento observado, analisamos as alterações realizadas em cada uma das *sprints* do *frontend* (Tabela 1). A *sprint* C (pré-*review*, *frontend*), que apresentou o maior número de *code smells*, teve o menor número de arquivos alterados e o segundo menor número de inserções dentre todas as *sprints* do pré-*review*/*frontend*. Entretanto, a *sprint* anterior (B), teve o maior número de arquivos modificados e inserções. Neste momento, podemos hipotetizar que, de fato, o alto número de alterações e inserções realizadas em *sprints* anteriores determina o aumento do número de *code smells* das próximas *sprints*. No entanto, a mesma hipótese não é confirmada para as *sprints* D e E (pré-*review*, *frontend*). A *sprint* D, que possui o segundo maior número de

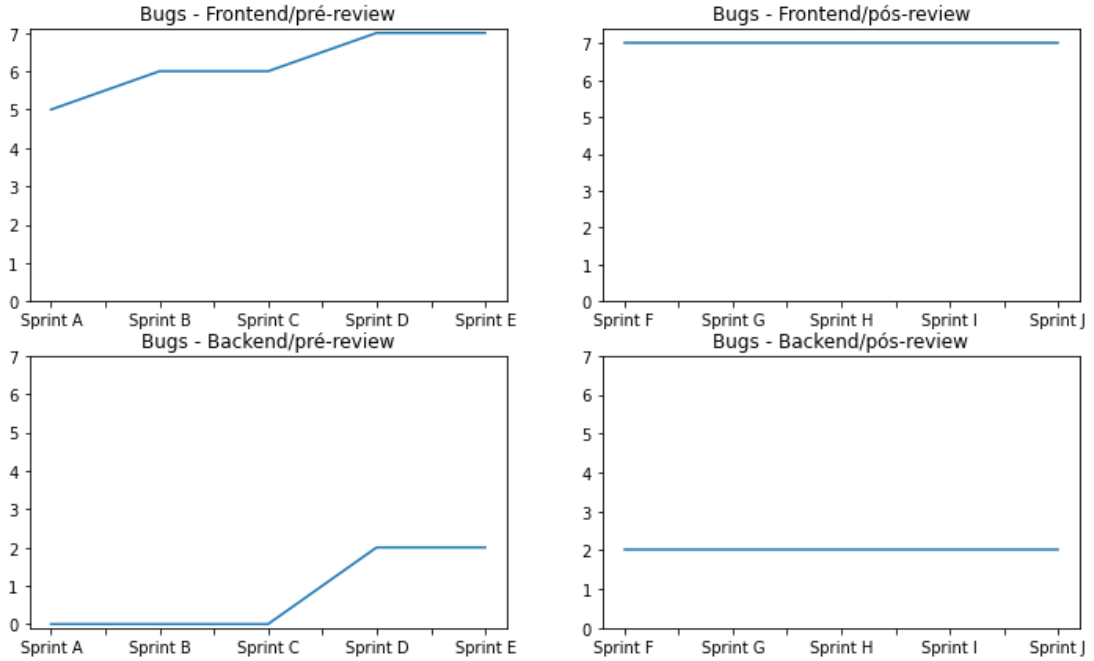

Figura 5: Evolução do número de *bugs* no pré e pós-*review*.

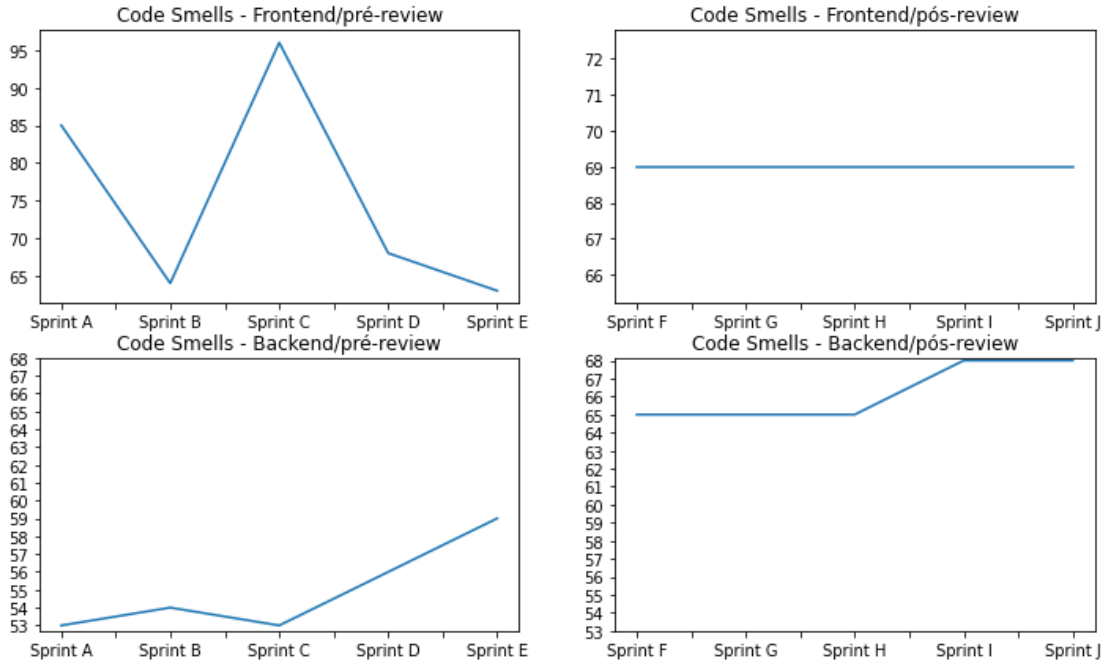

Figura 6: Evolução do número de *code smells* no pré e pós-*review*.

arquivos alterados (ficando atrás apenas da *sprint* B), não determina aumento do número de *code smells* da *sprint* seguinte (E), o que indica a influência de outros aspectos para problemas relacionados a *code smells*. Apesar disso, no pós-*review*, o número de *code smells* se manteve constante e, de acordo com relatórios do SonarQube, sem nenhum novo *code smell* detectado (no pré-review, 24 novos *code smells* foram encontrados).

No *backend*, a variância no pré-*review* é menor, com uma crescente constante a partir da *sprint* C. De maneira similar ao evidenciado no *frontend*, ao analisar o número de alterações, os 32 arquivos alterados e as 2040 inserções da *sprint* A parecem determinar o aumento de *code smells* da *sprint* B. No entanto, ao compararmos com os dados das *sprints* A e B, o baixo número de arquivos alterados e inserções da *sprint* C não determina uma redução do número de *code smells* para a *sprint* seguinte (D). Além disso, na *sprint* I (pós-*review*, *backend*), observamos um aumento do número de *code smells*, o que não é justificado pelos dados da Tabela 2, haja visto que as *sprints* anteriores (G e H) possuem os menores números de arquivos alterados e inserções dentre todas as outras. Além do mais, diferente do que acontece

no *frontend*, o pós-*review* no *backend* apresenta 7 novos *code smells*, número superior aos 3 novos detectados no pré-*review* (dados do relatório do SonarQube).

Apesar das variâncias observadas para os *code smells*, os resultados do SonarQube sugerem que, no pós-*review*, tanto *frontend* quanto *backend*, atendem as condições de qualidade pré-deploys. Em ambas as aplicações, das *sprints* A à E (pré-*review*), os *Quality Gates* falharam, indicando que as mesmas não atendiam aos padrões de qualidade esperados, enquanto que das *sprints* F à J (pós-*review*), os mesmos sucederam. Sendo assim, podemos afirmar que o *code review* melhorou, de maneira geral, a qualidade do código dos projetos.

Resultado 2: De maneira geral, o pós-review evidenciou melhorias nos resultados do SonarQube. Nenhum novo bug foi detectado neste período; em contrapartida, code smells ainda foram encontrados, o que pode indicar a influência de outros fatores. No pré-review, os Quality Gates (que mensuram padrões de qualidade) falharam, enquanto que, no pós-review, sucederam.

5.3 QP_3: A utilização do *code review* levou a equipe a discutir aspectos importantes para o projeto?

Nosso estudo gira em torno de dois momentos bem definidos: o pré e o pós-*review*. No momento pré-*review*, o processo de *code review* ainda não era uma parte efetiva do fluxo de desenvolvimento e, consequentemente, não existia um alinhamento prévio entre os desenvolvedores. Em muitas das vezes, os PRs eram simplesmente aprovados, sem que antes fossem, de fato, revisados. Portanto, quando acontecia, o *code review* possuía um número pequeno de comentários/*feedbacks* (quando existentes) e, por consequência, a interação entre autor e revisor era mínima. O pós-*review* caracteriza um fluxo de desenvolvimento que enxerga no *code review* uma grande oportunidade de garantia de qualidade do software, muito além do que uma simples parte burocrática do processo. Neste momento, espera-se uma maior interação entre autor e revisor, bem como discussões importantes em prol da qualidade do projeto. Para responder nossa Questão de Pesquisa 3, analisamos os dados coletados acerca dos *code reviews* realizados, buscando entender as principais mudanças, no que diz respeito às discussões entre autor/revisor, entre o pré e o pós-*review*.

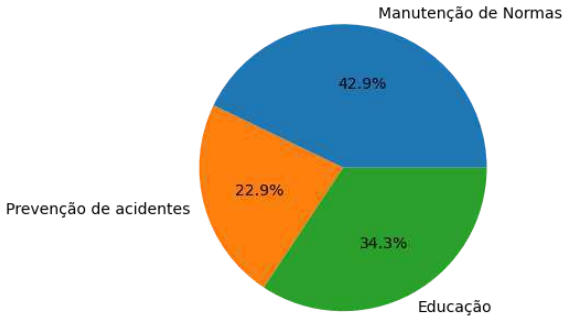

Figura 7: Gráfico de setores para as categorias de comentários no pré-*review*.

A Figura 7 apresenta, para o pré-*review*, o valor relativo de cada categoria considerada durante a classificação dos comentários. Evidenciamos que, neste momento, os comentários eram classificados, majoritariamente, como "Manutenção de Normas". Durante a coleta e análise dos comentários, reparamos que boa parte desses apontavam a existência de comandos `console.log` no código, sendo necessário a remoção dos mesmos como forma de manter uma das normas. Na sequência, a categoria "Educação" aparece: neste momento, observamos que grande parte dos comentários educativos eram simplesmente trechos de códigos, sem nenhum tipo de explicação ou justificativa. O trecho de código `data.languages?.split(',') ?? ''` é um exemplo desse tipo de comentário. Notamos que comentários pouco instigam a interação entre autor/revisor, sendo a falta de explicações mais detalhadas um dos principais motivos. Com a menor taxa de ocorrência, a categoria "Prevenção de acidentes", composta por comentários responsáveis por impedir a introdução de possíveis *bugs* no código fonte.

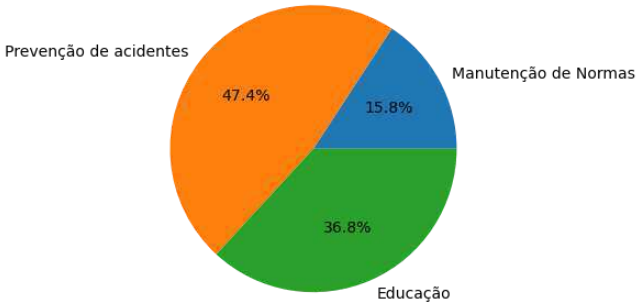

Figura 8: Gráfico de setores para as categorias de comentários no pós-*review*.

No pós-*review*, percebemos uma inversão dos resultados obtidos no momento anterior. Agora, o maior número de comentários coletados se enquadra na categoria "Prevenção de acidentes", o que revela uma maior preocupação com a qualidade do software (em termos de *bugs*/incidentes). O comentário *"Precisamos cobrir o caso em que o perfil do usuário não é encontrado (quando o método find retorna undefined). Da forma que está, possivelmente teremos um erro ao tentar ler a propriedade isEditable"* é um exemplo dessa categoria. Neste ponto, lembramos dos resultados obtidos para as Questões de Pesquisa anteriores (redução do número de incidentes e não surgimento de novos *bugs* no pós-*review*), que podem, também, embasar suas justificativas nessa mudança de comportamento nos *code reviews*. Além disso, outra alteração percebida foi que os comentários, independente da categoria, parecem estar melhor justificados, diferentemente do que foi observado no pré-*review*, no qual os comentários eram mais diretos. Por exemplo, o comentário *"Não podemos assinalar o ID do convênio dessa forma, porque o mesmo muda a depender do ambiente (dev, qas, uat, prd). O que acha de, primeiro, buscarmos o convênio e então usar o ID encontrado?"* foi de suma importância, uma vez que evitou um possível *bug* no código. Assim como aconteceu no pré-*review*, a categoria "Educação" aparece em segundo lugar, entretanto, uma diferença notada foi que, os comentários educativos do pós-*review* eram mais cuidadosos e menos diretos, favorecendo, inclusive, *threads* de discussões entre autor/revisor:

Acha válido mantermos esse comentário? (Revisor)

Então, acredito que facilita para o leitor do código entender que value.value trata-se de uma propriedade para os tipos de contato em geral. value.name, value.emails [...] são específicos para a secretária e values.landlines, [..] são específicos para o agendamento. O que acha? (Autor)

Faz sentido agora! (Revisor)

Por fim, em contraste com o pré-*review*, quando figurou em primeiro lugar, a categoria "Manutenção de normas" ocupa agora a terceira posição, ou seja, categoria de comentários menos recorrentes. Essa inversão do primeiro para o último lugar pode significar, sobretudo, uma maior atenção dos desenvolvedores em relação às normas que precisam ser respeitadas. De forma comparativa, não encontramos nenhum comentário referente a existência de comandos `console.log` no código, como aconteceu por algumas vezes no pré-*review*.

Resultado 3: Antes da consolidação do code review, a categoria de comentários "Prevenção de acidentes" era a menos recorrente. No cenário pós-review, a mesma passou a ser a categoria de comentários mais recorrentes. A categoria "Manutenção de normas" foi da mais (pré-review) para a menos recorrente (pós-review), o que pode indicar maior atenção dos desenvolvedores às normas consideradas. Além disso, houve uma mudança geral na formulação dos comentários, que passaram a ser melhor explicados e mais cuidadosos, favorecendo maior interação entre autor/revisor.

5.4 QP_4: Qual a perspectiva dos desenvolvedores sobre o uso de *code review* no projeto?

Por fim, em nossa última Questão de Pesquisa, investigamos as perspectivas/percepções dos desenvolvedores acerca do *code review* no projeto. Abaixo, destacamos os temas centrais da *survey* e seus respectivos resultados.

Experiência prévia e frequência pré-*review*: De todos os 5 desenvolvedores que responderam, 40% (2) destes não possuíam experiência prévia com *code review* antes da atuação no projeto. Além disso, no pré-*review*, apenas 1 desenvolvedor considerava realizar o *code review* como parte efetiva do fluxo de desenvolvimento. Outros 2 desenvolvedores raramente o faziam e os demais declararam realizar apenas às vezes.

Importância e impacto no projeto: De modo unânime, todos os respondentes acreditam na importância do processo de *code review*. Em 4 das 5 justificativas acerca da sua importância, o termo "qualidade" foi mencionado, o que indica um consenso geral do porquê do *code review*. Também unanimemente, todos os desenvolvedores acreditam que, em geral, houve melhoria no projeto (e.g., redução de incidentes/*bugs*) após a consolidação do *code review*. Inferimos, a partir das justificativas, que a redução de incidentes/*bugs* foi uma percepção geral entre todo o time.

Satisfação e principais ganhos: Em sua totalidade, todos os desenvolvedores que responderam à pesquisa estão satisfeitos com o processo atual de *code review*. No entanto, algumas justificativas reforçam a ideia de que o processo ainda pode ser melhorado. Uma possível melhoria apontada é tornar o processo cada vez mais ativo, ou seja, estimular ainda mais a interação entre autor e revisor, assim como diminuir e/ou eliminar possíveis dependências em relação aos desenvolvedores mais seniores, que, naturalmente, possuem um senso crítico maior. Além disso, em consonância com o nosso Resultado 3 (para o pós-*review*), os principais ganhos apontados pelos desenvolvedores são "Prevenção de bugs" e "Educação" (todos os respondentes votaram nessas opções), seguidos pela "Manutenção de normas", com 60% (3) das respostas. Nas justificativas, encontramos respostas que nos revelam que os desenvolvedores têm aprendido muito com o processo de *code review*. Em uma das respostas, temos o seguinte relato: *"Estou aprendendo muito com o code review e cada vez mais evoluindo em escrever código"*.

Principais desafios: No último tema abordado, buscamos entender melhor os desafios do atual processo de *code review*. Os principais desafios, apontados pelos desenvolvedores, são a demora para revisão e a (ainda) pouca interação entre autor/revisor, ambos com 40% (2) das respostas. Nas justificativas, os respondentes revelam que o processo pode ser demorado e que exige paciência, a depender do tamanho do PR. Além disso, outra resposta aponta para o fator demanda, que pode aumentar o tempo de espera pelas revisões. Uma terceira justificativa evidencia que, em alguns casos, revisores acabam por aguardar um primeiro *review* para, apenas depois, formular o seu próprio *feedback*. Essa última resposta está, inclusive, alinhada com a melhoria de tornar o processo mais ativo e menos dependente de desenvolvedores mais experientes

Resultado 4: Todos os desenvolvedores que participaram da survey acreditam na importância do processo de code review e, também, que este impactou positivamente o projeto, reduzindo o número de incidentes/bugs e proporcionando aprendizado. Apesar de satisfeitos, os desenvolvedores ainda enxergam pontos de melhoria para o processo de code review atual (e.g., tornar o processo ainda mais ativo), bem como desafios a serem superados (e.g., processo ainda pode ser demorado).

6. LIMITAÇÕES DO ESTUDO

Os dados coletados para este trabalho são provenientes das *sprints* de desenvolvimento, que, no contexto do projeto estudado, tinham um período definido de duas semanas. Por questões de tempo, limitamos a escolha dessas em 5 *sprints* para o pré-*review* e 5 *sprints* para o pós-*review*. Desse modo, o tempo representa uma possível limitação, haja visto que, no pós-*review*, o processo foi, também, sendo aprimorado. Um maior número de *sprints* consideradas poderia fornecer resultados mais significativos e claros. Além disso, como mencionamos, o time de desenvolvimento sofreu alterações durante o tempo da pesquisa, o que implicou em uma nova formação da equipe e, consequentemente, integrantes não habituados totalmente com o projeto e seus processos. Controlar a manutenção do time de desenvolvimento em longos períodos de tempo é uma tarefa difícil, haja visto a alta rotatividade da nossa área de atuação. Evidenciamos, também, que realizar as análises considerando o tamanho das modificações como a variável independente surge como uma perspectiva alternativa para nossa pesquisa. Nessa, nossos resultados seriam, possivelmente, mais consistentes, evitando a presença de *outliers*.

7. CONCLUSÕES E TRABALHOS FUTUROS

Nossa pesquisa, por meio de um estudo de caso, teve por objetivo avaliar o impacto do *code review* em aspectos qualitativos de um sistema real do Hospital Israelita Albert Einstein. Observamos, no pós-*review*, uma propensão à redução do número de incidentes, assim como melhoria na qualidade do código. A postura dos autores e revisores frente aos *reviews* mudou, indicando maior atenção dos mesmos a aspectos como manutenção de normas e prevenção de acidentes (e.g., *bugs*). Por fim, os desenvolvedores do projeto acreditam na importância e no impacto positivo do *code review*.

Ainda, traçar um paralelo entre os trechos de código revisados nos *code reviews* e os resultados do SonarQube, visando encontrar padrões, surge como uma oportunidade de trabalho futuro. Essa possível pesquisa evidenciaria se, de fato, o que é comentado durante as revisões refletem, positivamente ou negativamente, nos relatórios gerados pelo SonarQube e/ou outra(s) ferramenta(s) de análise estática.

8. AGRADECIMENTOS

A Deus, primeiramente, porque sem Ele nada disso seria possível. Ao meu pai, Osvaldo de Miranda Guedes, por ter feito do meu sonho, o dele também. A minha mãe e ao meu irmão, Lidineide da Silva Lima e Raul Lima Dantas, pelo apoio durante toda a trajetória. Aos amigos que fiz durante a jornada acadêmica, por toda parceria e crescimento profissional que tivemos juntos. A minha namorada, Izadora Nóbrega, por todo suporte durante o desenvolvimento desta pesquisa. Ao professor e orientador Everton, por todo conhecimento, suporte e compreensão durante o desenvolvimento deste trabalho. Meus sinceros agradecimentos, também, ao HIAE, por permitir que trabalhássemos com um de seus projetos, por meio do qual este TCC foi possível. Aos meus coordenadores, Raiane Pagnan e André Pires, por apoiarem a ideia deste trabalho. Agradeço também a todos meus companheiros de profissão no HIAE, por todo crescimento profissional que tivemos juntos, esse trabalho também é de vocês.

9. REFERÊNCIAS

[1] Beck, K., et al. (2001) The Agile Manifesto. Agile Alliance. Disponível em: <http://agilemanifesto.org>. Acesso em: 20 de jan. de 2023.

[2] Hospital Israelita Albert Einstein. Einstein: melhor hospital da América Latina, segundo a Newsweek. Disponível em: <https://www.einstein.br/newsweek>. Acesso em 20 de jan. de 2023.

[3] Sommerville, I. (2011) Software Engineering, 9th Edition, Pearson.

[4] Fowler, Martin. Continuous Integration, 2006. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 20 de jan. de 2023.

[5] Fowler, Martin. Continuous Delivery, 2013. Disponível em: <https://martinfowler.com/bliki/ContinuousDelivery.html>. Acesso em: 20 de jan. de 2023.

[6] Fowler, Martin. DeploymentPipeline, 2013. Disponível em: <https://martinfowler.com/bliki/DeploymentPipeline.html>. Acesso em 20 de jan. de 2023.

[7] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, 712–721.

[8] MICROSOFT. Azure DevOps: Plan smarter, collaborate better, and ship faster with a set of modern dev services, 2005. Disponível em: <https://azure.microsoft.com/en-us/products/devops>. Acesso em: 20 de jan. de 2023.

[9] SONARSOURCE. SonarQube, 2006. Disponível em: <https://www.sonarsource.com/products/sonarqube/>. Acesso em 20 de jan. de 2023.

[10] KAMIL MYSLIWIEC. NestJS - A progressive Node.js framework, 2017. Disponível em: <https://nestjs.com/>. Acesso em: 20 de jan. de 2023.

[11] META OPEN SOURCE. React: A JavaScript library for building user interfaces, 2013. Disponível em: <https://reactjs.org/>. Acesso em: 20 de jan. de 2023.

[12] Corbin, J., & Strauss, A. (1990). Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative Sociology, 13, 3-21.

[13] Likert, Rensis (1932). "A Technique for the Measurement of Attitudes". Archives of Psychology 140: 1–55.

[14] HEXACTA. Hexacta: Delivering software on-time vs quality: pursuing the right balance, 2021. Disponível em: <https://www.hexacta.com/delivering-software-on-time-vs-quality-pursuing-the-balance/>. Acesso em: 20 de jan. de 2023.