UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS DA TERRA
CURSO DE BACHARELADO EM ENGENHARIA DE SOFTWARE

ALEX SANDRO DE PAIVA

SMART REVIEW: AUTOMAÇÃO DO PROCESSO DE COMUNICAÇÃO SOBRE REVISÃO DE CÓDIGO-FONTE

NATAL-RN
2025

ALEX SANDRO DE PAIVA

SMART REVIEW: AUTOMAÇÃO DO PROCESSO DE COMUNICAÇÃO SOBRE REVISÃO DE CÓDIGO-FONTE

Monografia apresentada ao curso de graduação em Engenharia de Software, da Universidade Federal do Rio Grande do Norte, como requisito parcial à obtenção do título de Bacharel em Engenharia de Software.

Orientador: Prof. Dr. Gibeon Soares de Aquino Junior.

NATAL-RN

2025

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Paiva, Alex Sandro de.
Smart Review: automação do processo de comunicação sobre revisão de código-fonte / Alex Sandro de Paiva. - 2025.
113f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Curso de Engenharia de Software. Natal, RN, 2025.
Orientação: Prof. Dr. Gibeon Soares de Aquino Junior.

1. Engenharia de software - Monografia. 2. Revisão de código moderna - Monografia. 3. Comunicação - Monografia. 4. Automação - Monografia. 5. Chatbots - Monografia. 6. Webhooks - Monografia. I. Aquino Junior, Gibeon Soares de. II. Título.

RN/UF/CCET CDU 004.41

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

ALEX SANDRO DE PAIVA

SMART REVIEW: AUTOMAÇÃO DO PROCESSO DE COMUNICAÇÃO SOBRE REVISÃO DE CÓDIGO-FONTE

Monografia apresentada ao curso de graduação em Engenharia de Software, da Universidade Federal do Rio Grande do Norte, como requisito parcial à obtenção do título de Bacharel em Engenharia de Software.

Aprovada em: 26/12/2024

BANCA EXAMINADORA

Prof. Dr. Gibeon Soares de Aquino Junior
Orientador
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Prof.ª Dra. Roberta de Souza Coelho
Membro interno
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Prof. Dr. Mário Andrade Vieira de Melo Neto
Membro externo
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO NORTE

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão, primeiramente a Deus, por Sua presença constante em minha vida, iluminando meus caminhos, fortalecendo-me nos momentos de dificuldade e abençoando cada etapa desta jornada acadêmica. Sem Sua graça e orientação, os desafios enfrentados teriam sido muito mais difíceis de superar. A Ele, dedico esta conquista com toda minha fé e reconhecimento.

Agradeço também à minha família, que esteve sempre ao meu lado, oferecendo apoio e incentivo em todos os momentos. Em especial, expresso minha gratidão ao meu pai, Pedro Firmo de Paiva, à minha mãe, Maria José Martins de Paiva, e ao meu irmão, Adson Luis de Paiva, por seu amor incondicional, dedicação incansável e palavras de encorajamento. Cada gesto de cuidado e motivação foi essencial para que eu chegasse até aqui.

Quero registrar meu profundo agradecimento aos colegas do SmartSuite, em especial, Thales Gomes Moreira, Eduardo Soares de Araujo Aquino, Thiago de Oliveira Cordeiro, Jonas Mateus Bezerra dos Santos, Jackson Dantas Junior, Song Jong Márcio Simioni da Costa e Guilherme Egle Pegado Lima Silva, por sua colaboração, apoio e disposição ao longo do desenvolvimento deste trabalho.

No mais, manifesto meu enorme agradecimento ao professor, orientador desta monografia e coordenador do SmartSuite, Gibeon Soares de Aquino Junior, pela orientação, paciência e dedicação durante todo o processo. Sua experiência e conselhos foram essenciais para o desenvolvimento deste trabalho, e sua constante motivação e apoio foram fundamentais para que eu alcançasse este importante marco acadêmico.

Successful management of any process requires planning, measurement, and control.

Michael Fagan

RESUMO

Este trabalho tem como objetivo propor práticas de automação na comunicação das etapas de revisão de código, com foco na integração de plataformas de gerenciamento de repositórios Git (com webhooks) e chatbots em Redes Sociais Corporativas (ESN). Como objetivos específicos, busca-se mapear o fluxo de revisão de código moderna (MCR), analisar ferramentas disponíveis no mercado nos últimos 15 anos, conceber um sistema para efetivar a comunicação do MCR, estabelecer métricas de avaliação e testar a solução em um ambiente real. A pesquisa adotou um método aplicado, baseado na metodologia de Engenharia de Software proposta por Sommerville, estruturado em quatro fases: elicitação de requisitos, análise e design de projeto, implementação e estudo de caso. Os resultados indicam que a automação no processo de revisão de código reduz tarefas repetitivas, melhora a clareza e consistência da comunicação, e promove maior alinhamento com as diretrizes da equipe. Conclui-se que, ao integrar dois sistemas bem definidos — um para controle de versões de artefatos e outro para comunicação — no contexto de revisão de código, é possível alcançar ganhos importantes em termos de eficiência, tornando-se uma estratégia promissora para aprimorar a produtividade das equipes de desenvolvimento de software.

Palavras-chave: revisão de código moderna; comunicação; automação; chatbots; webhooks.

ABSTRACT

This study aims to propose automation practices in the communication of code review stages, focusing on the integration of Git repository management platforms (with webhooks) and chatbots in Enterprise Social Networks (ESN). Specific objectives include mapping the modern code review flow (MCR), analyzing tools available in the market over the past 15 years, designing a system to facilitate MCR communication, establishing evaluation metrics, and testing the solution in a real-world environment. The research adopted an applied method, based on the Software Engineering methodology proposed by Sommerville, structured into four phases: requirements elicitation, analysis and project design, implementation, and case study. The results indicate that automation in the code review process reduces repetitive tasks, improves the clarity and consistency of communication, and promotes greater alignment with team guidelines. It is concluded that by integrating two well-defined systems — one for artifact version control and another for communication — within the context of code review, important efficiency gains can be achieved, making it a promising strategy to enhance the productivity of software development teams.

Keywords: modern code review; communication; automation; chatbots; webhooks.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE QUADROS

LISTA DE SIGLAS

API	*Application Programming Interface*
BPMN	*Business Process Model and Notation*
BSD	*Berkeley Software Distribution*
CD	*Continuous Delivery*
CI	*Continuous Integration*
CRUD	*Create/Read/Update/Delete*
CSRF	*Cross-Site Request Forgery*
CVCS	*Centralized Version Control System*
DRF	*Django Rest Framework*
DRY	*Don't Repeat Yourself*
DVCS	*Distributed Version Control System*
EER	*Extended Entity-Relationship*
ESN	*Enterprise Social Networks*
FN	Forma Normal
HTTP	*Hypertext Transfer Protocol*
AI	*Artificial Intelligence*
ITK	*Innovation Tech Knowledge*
IP	*Internet Protocol*
JPL	*Jet Propulsion Laboratory*
JSON	*JavaScript Object Notation*
LAN	*Local Area Network*
LGTM	*Looks Good to Me*
MCR	*Modern Code Review*
ML	*Machine Learning*
MVC	*Model-View-Controller*
MVT	*Model-View-Template*
NASA	*National Aeronautics and Space Administration*
NN	*Neural Network*
NoSQL	*Not Only SQL*
PGDG	*PostgreSQL Global Development Group*
ORM	*Object-Relational Mapping*
REST	*Representational State Transfer*

RF	Requisito Funcional
SCRUB	*Source Code Review User Browser*
SQL	*Structured Query Language*
TCP	*Transmission Control Protocol*
URL	*Uniform Resource Locator*
VCS	*Version Control System*
VTK	*The Visualization Toolkit*
WAN	*Wide Area Network*
WE	*Word Embedding*
XSS	*Cross-Site Scripting*

SUMÁRIO

1. INTRODUÇÃO

Cada vez mais empresas, negócios e pessoas necessitam de software para a tomada de decisões estratégicas e táticas, além de gerenciar e operacionalizar suas atividades cotidianas. Qualquer falha no software pode resultar em pequenos inconvenientes ou até mesmo em situações catastróficas para os envolvidos. Por isso, espera-se que a qualidade do software seja geralmente elevada, como enfatizado por Pressman e Maxim (2016).

Nesse contexto, a engenharia de software assume um papel crucial para garantir níveis satisfatórios de qualidade. Ela oferece um conjunto de métodos e ferramentas que auxiliam no processo de desenvolvimento e manutenção de software, visando atender a esses elevados padrões de qualidade (PRESSMAN; MAXIM, 2016).

Uma das práticas fundamentais dentro da engenharia de software é a revisão de código por pares. Esta prática visa não apenas manter, mas também promover a qualidade do código-fonte. Ela contribui para o fortalecimento da comunidade de desenvolvimento, facilitando a transferência de conhecimento e a aplicação de soluções de design e implementação que podem ter sido desenvolvidas por outros membros da equipe (BACCHELLI; BIRD, 2013).

A revisão de código moderna (*Modern Code Review* — MCR) caracteriza-se por ser um processo leve e informal. Com o uso de ferramentas, ela se torna assíncrona e foca em revisar apenas as alterações propostas no código, em vez de revisar toda a base de código (RIGBY; BIRD, 2013). Esse modelo proporciona maior agilidade e eficiência nas modificações, facilitando o trabalho das equipes de desenvolvimento e permitindo que o processo de revisão se adapte melhor à dinâmica de trabalho.

O processo de revisão de código moderno tende a se integrar a plataformas de gerenciamento de repositórios, como GitHub, GitLab e Bitbucket. Essas ferramentas permitem a criação de solicitações de mesclagem (*pull requests*, *merge requests*), facilitando revisões dinâmicas com comentários, aprovações e ajustes diretamente no código.

Instituições renomadas, como Google (SADOWSKI *et al*., 2018), Microsoft (BACCHELLI; BIRD, 2013), NASA (HOLZMANN, 2010), Chromium (BIASE;

BRUNTINK; BACCHELLI, 2016), e projetos como Qt, VTK e ITK (MCINTOSH *et al*., 2014), utilizam amplamente a MCR, destacando sua eficácia na otimização do processo de revisão de código em ambientes colaborativos.

Um elemento chave na prática de revisão de código é a comunicação. Ela não só facilita a troca de informações entre os membros da equipe, mas também alinha expectativas, compartilha conhecimento e contribui para a identificação de melhorias no código. A qualidade da comunicação entre os desenvolvedores impacta diretamente a eficácia da revisão e o resultado final do código.

Na última década, o modelo de comunicação em ambientes corporativos passou por transformações significativas, especialmente com os desafios impostos pela pandemia de COVID-19. A flexibilização do trabalho e a autonomia proporcionada por horários adaptados e pela independência do local de atuação mudaram a dinâmica das interações profissionais (MAJUMDAR; KRISHNA; BJORN, 2013). Esses fatores exigem uma adaptação nas abordagens de comunicação dentro dos processos de desenvolvimento de software, incluindo na revisão de código.

Na revisão de código moderna, surgem diversos desafios relacionados à comunicação, como a escolha do mecanismo de notificação, das ferramentas que facilitam a interação e da abordagem de comunicação mais adequada, seja ela síncrona, assíncrona ou híbrida. Tais decisões impactam na fluidez do processo de revisão.

Nesse contexto, as Redes Sociais Corporativas (*Enterprise Social Networks* — ESN) emergiram como ferramentas estratégicas para facilitar a comunicação, promover a colaboração e fortalecer a integração entre equipes, especialmente em ambientes de trabalho remotos (BEHRENDT; RICHTER; TRIER, 2014). Essas plataformas se destacam por sua capacidade de centralizar informações, fomentar discussões e viabilizar um fluxo contínuo de troca de conhecimento, elementos essenciais para a eficiência e inovação em organizações modernas.

1.1. Problema

No desenvolvimento de software que adota a prática de revisão de código moderna, baseada em plataformas de gerenciamento de repositórios, surge a

necessidade de acompanhar constantemente o progresso das revisões. Isso geralmente requer o uso de interfaces web com atualização manual de páginas ou a dependência de notificações, frequentemente produzidas manualmente por terceiros, como e-mails e mensagens em redes sociais empresariais, especialmente quando se está diretamente envolvido em uma revisão específica.

Essa dependência de interfaces e notificações manuais gera impactos negativos na eficiência e produtividade das equipes de desenvolvimento. O processo fragmentado exige que os desenvolvedores interrompam suas tarefas para verificar o andamento das revisões, resultando em perda de tempo e foco.

Ademais, no âmbito da comunicação durante o processo de revisão, um dos principais desafios é adaptar o processo às especificidades de cada equipe e projeto. A falta de personalização nas etapas de revisão compromete a eficiência do fluxo de trabalho, dificultando a definição de critérios claros de qualidade e prazos de entrega. A rigidez no processo pode levar a abordagens ineficazes, que não atendem às necessidades distintas de cada instituição.

Outro problema crítico é o controle inadequado do tempo gasto nas revisões. Esse tempo impacta diretamente o cronograma do projeto e os prazos de entrega, resultando em atrasos e sobrecarga nas equipes. Quando o processo não é gerido adequadamente, ele pode se arrastar, prejudicando a agilidade e a produtividade.

1.2. Objetivos

O objetivo geral deste trabalho é definir e implementar práticas de automação no processo de comunicação de eventos de revisão de código, com o intuito de aumentar a eficiência, a padronização e a colaboração entre os envolvidos, em comparação com a comunicação realizada de forma manual. Entre os principais objetivos específicos, destacam-se:

- Mapear as etapas do processo de revisão de código e identificar os fluxos de comunicação envolvidos;
- Analisar ferramentas e práticas de automação disponíveis no mercado para a comunicação durante a revisão de código;

- Desenvolver uma ferramenta para automatizar a comunicação nas etapas de revisão de código moderna;
- Estabelecer métricas e indicadores de desempenho para avaliar a eficiência da automação implementada;
- Testar a automação no ambiente real de desenvolvimento.

1.3. Metodologia

O presente estudo caracteriza-se como aplicado, com enfoque no desenvolvimento de um sistema para lidar com a comunicação de revisão de código moderna. Adotou-se uma abordagem exploratória para conceber os requisitos do sistema, seguida de uma abordagem qualitativa para avaliar os resultados obtidos. O trabalho seguiu uma metodologia estruturada em quatro fases, a saber:

- **Fase I** – Elicitação de Requisitos: Fase inicial com objetivo de identificar os *stakeholders*, requisitos funcionais e requisitos de qualidade do sistema por meio de análise de processos existentes e revisão bibliográfica de ferramentas similares.
- **Fase II** – Análise e Design de Projeto: Fase em que os requisitos catalogados são investigados e convertidos em visões de dados e arquiteturas (estrutural e comportamental).
- **Fase III** – Implementação: Fase na qual as estruturas de dados e a arquitetura de software são implementadas, aplicando ferramentas, tecnologias e padrões consolidados.
- **Fase IV** – Estudo de Caso: Etapa final que consiste em implantar o sistema em produção, avaliá-lo sob diferentes perspectivas qualitativas, como usabilidade, experiência do usuário e satisfação dos envolvidos, e coletar feedbacks para identificar áreas de melhoria.

1.4. Declarações e contribuições

Este trabalho tem como principal contribuição o desenvolvimento de um sistema que inclui a disponibilização de uma Interface de Programação de Aplicativos (API). Esse sistema é capaz de integrar-se aos serviços de hospedagem

Git, utilizando webhooks, e às Redes Sociais Corporativas, por meio de chatbots. O objetivo é automatizar a comunicação no processo de revisão de código moderna, proporcionando maior adaptabilidade e eficiência ao fluxo de trabalho.

1.5. Organização do trabalho

Este documento divide-se em seis capítulos. **Capítulo 1** (em circunstâncias): contextualiza a linha de pesquisa, apresenta os problemas e objetivos, descreve a metodologia e a contribuição do estudo, além de detalhar a organização do trabalho; **Capítulo 2**: descreve a fundamentação teórica, com intuito de apresentar técnicas, processos, conceitos, ferramentas e tecnologias; **Capítulo 3**: explora estudos correlatos, destacando soluções de terceiros que, de alguma forma, buscam resolver os problemas abordados neste estudo; **Capítulo 4**: apresenta a proposta de solução, descrevendo o processo de desenvolvimento e as limitações identificadas; **Capítulo 5**: detalha um estudo de caso realizado com a solução proposta, abordando o escopo, planejamento, operação e resultados obtidos; **Capítulo 6**: traz as considerações finais sobre o trabalho desenvolvido, destacando as contribuições e possíveis desdobramentos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

A prática de revisão de código cada dia mais está imersa na cultura de desenvolvimento de software, com foco na inspeção do código antes de sua integração na base de código e implantação (BADAMPUDI; UNTERKALMSTEINER; BRITTO, 2023, p. 1). No entanto, o processo de comunicação durante revisões frequentemente abarca desafios relacionados à eficiência, clareza e agilidade. Assim, a automação surge como uma solução promissora para otimizar a troca de informações e reduzir o tempo gasto em revisões manuais.

Este capítulo busca apresentar os fundamentos que sustentam o estudo sobre a automação da comunicação no processo de revisão de código. Serão abordados conceitos de revisão de código, comunicação em equipes de desenvolvimento, controle de repositórios Git, Arquitetura Producer-Worker e tecnologias para implementação da automação.

2.1. Revisão de Código

A revisão de código por pares, uma inspeção manual do código-fonte realizada por desenvolvedores diferentes do autor, caracteriza-se como uma prática valiosa para reduzir defeitos de software e melhorar a qualidade dos projetos (ACKERMAN; FOWLER; EBENAU, 1984). Além disso, trata-se de uma prática orientada por tecnologia que complementa a integração e implantação contínuas (CI/CD), permitindo o lançamento de novos recursos com frequência e confiabilidade (BADAMPUDI; UNTERKALMSTEINER; BRITTO, 2023, p. 1).

Em 1976, Fagan (2002) estabeleceu um método rigorosamente estruturado para a avaliação de código, fundamentado em revisões de grupo, abordando detalhadamente cada linha, durante reuniões extensas conhecidas como inspeções de código.

Com o passar dos anos, pesquisadores têm apresentado indícios dos benefícios proporcionados pela inspeção de código, destacando em particular sua eficácia na detecção de defeitos. No entanto, a complexidade, o tempo demandado e a natureza síncrona dessa abordagem têm representado desafios significativos, dificultando sua adoção generalizada na prática (SHULL; SEAMAN, 2008).

Atualmente, diversas organizações estão incorporando abordagens de revisão de código mais ágeis como uma estratégia para mitigar as ineficiências associadas às inspeções. Desse modo, as revisões de código de software evoluíram de rigorosas, colocadas e síncronas para leves, distribuídas, baseadas em ferramentas e assíncronas (SADOWSKI *et al.*, 2018).

2.1.1. Revisão de Código Moderna

A revisão de código moderna (*Modern Code Review* — MCR), surge como uma prática informal, baseada em ferramentas, assíncrono e focado na inspeção de novas alterações propostas no código (RIGBY; BIRD, 2013). Uma alternativa leve às inspeções de códigos tradicionais (FAGAN, 2002) que se concentra em alterações de código e permite que os desenvolvedores de software melhorem a qualidade de código e reduzam os defeitos pós-entrega (BAVOTA; RUSSO, 2015).

Dentre as características da MCR, destacam-se as seguintes práticas:

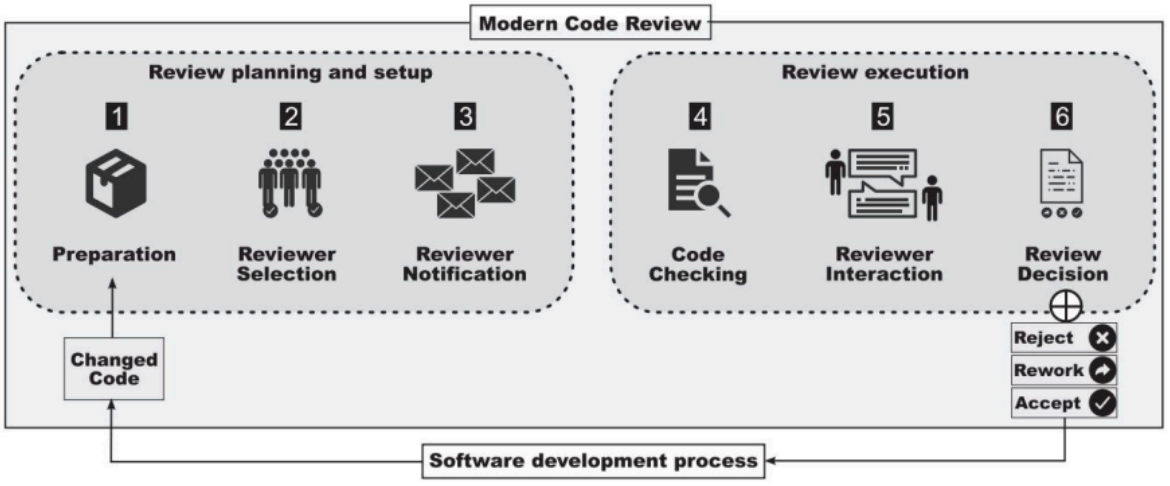
- **Revisões assíncronas**: favorecem discussões em equipe sobre soluções para defeitos, identificando a mesma quantidade de problemas em menos tempo que reuniões presenciais. Além disso, permitem que desenvolvedores e ouvintes passivos aprendam com essas discussões (RIGBY *et al*., 2012);
- **Revisões frequentes**: quanto mais tempo um defeito persiste em um artefato, mais dispendiosa torna-se sua correção (FAGAN, 2002). Em projetos de código aberto, revisões assíncronas contínuas são comuns, aproximando-se de uma forma assíncrona de programação em pares (RIGBY *et al*., 2012);
- **Revisões incrementais**: mudanças pequenas, independentes e completas facilitam o trabalho dos revisores, reduzindo o tempo de preparação e permitindo uma visão clara de como a alteração se integra ao sistema (RIGBY *et al*., 2012);
- **Revisões por especialistas**: são essenciais devido à compreensão prévia do contexto da mudança. Revisores sem conhecimento detalhado podem ter dificuldades, especialmente com artefatos complexos (RIGBY *et al*., 2012);

- **Revisões por ferramentas**: com as rápidas mudanças nos projetos, sistemas automatizados, como análise estática e aprendizado profundo, tornam a revisão mais eficiente e prática (SIOW *et al*., 2020).

O processo formal de inspeção de código requer entradas bem definidas (um produto de software pronto para revisão), planejamento e organização de recursos de revisão (tempo, local, experiência), execução da revisão seguindo diretrizes que facilitam a detecção de defeitos e uma síntese das descobertas, que são usadas para melhoria (AURUM; PETERSSON; WOHLIN, 2002).

A Figura 1 ilustra as etapas do processo de revisão de código moderna, normalmente suportadas por ferramentas que se integram com sistemas de controle de versão, como Gerrit, GitHub e GitLab (BADAMPUDI; UNTERKALMSTEINER; BRITTO, 2023, p. 3). Os principais envolvidos são o(s) autor(es) do código e o(s) revisor(es).

Figura 1 – Processo de revisão de código moderno



Fonte: Badampudi, Unterkalmsteiner e Britto (2023, p. 3).

Segundo Badampudi, Unterkalmsteiner e Britto (2023, p. 3), o processo aplica-se independente do contexto organizacional, seja desenvolvimento de software de código aberto ou comercial. A diferença percebe-se quanto ao propósito, enquanto um busca-se construir relacionamentos entre os envolvidos no processo, no outro almeja-se disseminar conhecimento.

As etapas do MCR incluem:

1) **Preparar solicitação**: O autor do código prepara a alteração para revisão, incluindo uma descrição detalhada da modificação proposta e,

frequentemente, uma referência ao problema registrado em um sistema de rastreamento de bugs. Em ferramentas como Gerrit e GitHub, o autor cria uma solicitação de mesclagem, com nomenclaturas específicas, como *pull request* no GitHub e *merge request* no GitLab.

2) **Selecionar revisor(es)**: O proprietário do projeto seleciona um ou mais revisores, considerando fatores como expertise no código afetado e disponibilidade de tempo.
3) **Notificar revisor(es)**: Os revisores são informados de sua tarefa, concluindo assim a fase de planejamento do MCR.
4) **Verificar código**: Os revisores analisam as alterações em busca de defeitos ou sugerem melhorias. O procedimento desta etapa varia conforme o suporte da ferramenta, a cultura organizacional e a experiência dos revisores. Normalmente, inspecionam-se as alterações no contexto do código inalterado, fornecendo *feedback* para linhas específicas e para a modificação geral.
5) **Interagir**: Revisores e autores discutem as alterações e o *feedback* por meio de ferramentas que suportam comunicação assíncrona e permitem referenciar tanto o código quanto os participantes. Essa interação gera um registro permanente das discussões técnicas realizadas durante a revisão.
6) **Decidir revisão**: Ao final, a modificação é rejeitada, aceita ou devolvida ao autor para refinamento. O processo decisório pode ser realizado por votação ou ficar a cargo do proprietário do projeto.

2.2. Comunicação em Equipes de Desenvolvimento

A comunicação é, antes de mais, uma experiência antropológica fundamental. Intuitivamente, comunicar consiste em trocar algo com alguém. Além disso, envolve um conjunto de técnicas e tornou-se uma necessidade social funcional para economias interdependentes (WOLTON, 2004).

No contexto da produção de software, a comunicação desempenha um papel essencial no processo de desenvolvimento. Embora diversos fatores influenciem o sucesso de um projeto, frequentemente o fracasso não está relacionado à falta de conhecimento técnico da equipe. O aspecto mais crucial para

a equipe é garantir que cada membro esteja ciente de seus deveres, poderes, responsabilidades e interesses (DEFRANCO; LAPLANTE, 2017).

2.2.1. Plataforma de comunicação de equipe baseada em nuvem

Nos últimos anos, as empresas têm observado um aumento na demanda por maior flexibilidade e autonomia no trabalho, especialmente em relação a modelos que não dependem de local ou horário fixos (MAJUMDAR; KRISHNA; BJORN, 2013). Essa transformação impulsionou a adoção de tecnologias que permitem a colaboração distribuída e o fortalecimento da comunicação organizacional.

Nesse contexto, Behrendt, Richter e Trier (2014) destacam a relevância das Redes Sociais Corporativas (*Enterprise Social Networks* — ESN) para apoiar práticas colaborativas em ambientes corporativos. De acordo com os autores, as ESN são plataformas baseadas na web que permitem aos usuários compartilhar objetos persistentes em um repositório comum.

Além disso, essas plataformas oferecem funcionalidades como respostas públicas a objetos compartilhados, exibição de informações de perfil e conexões entre usuários por meio de interações, como seguir ou enviar solicitações de amizade. Exemplos incluem *weblogs*, *wikis*, *microblogs* e redes sociais integrados em sistemas unificados de ESN.

Entre as ESN mais amplamente utilizadas, o Slack se destaca como um espaço de trabalho digital baseado em nuvem e sistema de gerenciamento de informações. A plataforma permite a colaboração em tempo real entre diversos usuários em projetos de grupo. Por adotar uma estrutura semelhante à de sistemas de mensagens instantâneas, promove uma interação mais fluida e intuitiva, diferenciando-se de outros sistemas colaborativos (MONTRIEF *et al*., 2020).

> O Slack é um poderoso sistema operacional de trabalho que reúne todos os componentes essenciais do trabalho em uma única plataforma de conversação. Ao integrar-se com as principais aplicações empresariais, o Slack simplifica o acesso aos dados e fluxos de trabalho de uma organização em um local unificado. O Slack pode ser dividido nos três componentes que compõem qualquer organização: pessoas, conhecimento e ferramentas (SALESFORCE, 2024, p. 6).

1) **Sua equipe***:* O Slack centraliza equipes em um único espaço de trabalho, onde podem colaborar em canais públicos/privados organizados por projetos ou tópicos, seja em tempo real ou assíncrono (SALESFORCE, 2024, p. 6).
2) **Seu conhecimento**: O Slack rompe barreiras criadas por silos de informação, centralizando o conhecimento da organização em um recurso único e acessível (SALESFORCE, 2024, p. 9).
3) **Suas ferramentas***:* O Slack conecta ferramentas empresariais, desde sistemas legados até soluções modernas, através do diretório de aplicativos, chatbots e serviços (SALESFORCE, 2024, p. 10).

Nesse cenário, as Slack apps destacam-se como um recurso central da plataforma, integrando-se a sistemas externos para atender às diversas necessidades organizacionais. Com APIs avançadas, essas aplicações podem ler, escrever e atualizar dados na plataforma, oferecendo experiências dinâmicas e personalizadas para os usuários (SALESFORCE, 2024).

As Slack apps fornecem interações automatizadas, incluindo chatbots, que impulsionam a automação de tarefas e a otimização da comunicação interna. Esses recursos simplificam processos e o gerenciamento de informações, tornando os fluxos de trabalho mais eficientes e promovendo maior produtividade e colaboração nas equipes (SALESFORCE, 2024).

2.3. Chatbots

Conforme Lokman e Ameedeen (2019, p. 1012), *chatbot* (também conhecido como *Chatting Robot*) consiste em um sistema computacional que possibilita a comunicação entre humanos e computadores utilizando a linguagem natural. O conceito surgiu na década de 60, com a criação de ELIZA, um programa de computador para o estudo da comunicação em linguagem natural entre o homem e a máquina (WEIZENBAUM, 1966).

Para Lokman & Ameedeen (2019, p. 1012), observa-se um crescimento dos chatbots com o advento da Inteligência Artificial (*Artificial Intelligence* — AI). Sendo sua relevância aumentada por causa da mudança na formação de comunicação, de longa interações direta (chamadas telefónicas) para trocas de mensagens curtas.

Os chatbots modernos podem ser projetados considerando quatro elementos principais: o tipo de conhecimento (se abrangem um domínio aberto ou fechado), a forma de geração de respostas (se utilizam recuperação ou geração), o processamento de texto e o modelo de aprendizado de máquina empregado, como redes neurais (LOKMAN; AMEEDEEN, 2019).

2.3.1. Domínio do conhecimento

Segundo Lokman e Ameedeen (2019, p. 1013), chatbots podem ser classificados em duas categorias: domínio aberto ou fechado, com base no conhecimento que abrangem. No domínio aberto, o *chatbot* deve possuir conhecimento geral, incluindo tópicos atuais, entretenimento e aspectos do conhecimento humano. Já no domínio fechado, o *chatbot* se restringe a um campo específico de conhecimento, como atendimento ao cliente ou psicologia.

2.3.2. Geração de Resposta

De acordo com Lokman e Ameedeen (2019, p. 1013–1014), os chatbots podem gerar respostas por meio de dois métodos principais: recuperação e geração. Embora existam várias variações desses métodos, o processo fundamental continua o mesmo. A recuperação envolve selecionar a melhor resposta entre as opções pré-determinadas, enquanto a geração permite a criação flexível de respostas com base na sequência de entrada e em classificadores treinados.

2.3.3. Processamento de Texto

Para Lokman e Ameedeen (2019, p. 1014–1015), o processamento de texto em chatbots envolve a manipulação e transformação de dados de entrada (como palavras ou caracteres) para gerar respostas apropriadas. Esse processo pode incluir várias abordagens, como o uso de *Word Embedding* (WE), onde palavras são representadas como números reais em um espaço vetorial, permitindo que o *chatbot* entenda as relações semânticas entre elas.

2.3.4. Modelo de Aprendizado de Máquina

Lokman e Ameedeen (2019, p. 1015–1016) afirmam que o Modelo de Aprendizado de Máquina (*Machine Learning* – ML) desempenha um papel crucial nas arquiteturas modernas de chatbots, sendo aplicado em várias fases do processo, incluindo o processamento de entradas e a geração de saídas. As Redes Neurais (*Neural Network* – NN), em particular, são fundamentais, pois ajudam a transformar os dados de entrada e a criar respostas de maneira eficiente, permitindo que o sistema aprenda e se adapte ao longo do tempo.

Uma aplicação específica de ML no processamento de texto é o uso de *Word Embedding* (WE), que gera representações vetoriais das palavras. Esse método, baseado em abordagens estatísticas como o preditivo, permite que o *chatbot* compreenda relações semânticas entre as palavras e situe melhor as interações.

2.4. Controle de Repositórios Git

Um Sistema de Controle de Versão (*Version Control System* – VCS) registra alterações em arquivos ou conjuntos de arquivos armazenados no sistema. Por exemplo, pode ser código-fonte, ativos e outros documentos que podem fazer parte de um projeto de desenvolvimento de software (DAVIS; DANIELS, 2016).

Ele permite a aceleração e simplificação do processo de desenvolvimento de software, possibilita novos fluxos de trabalho, permite monitorar os arquivos e seu histórico e possui um modelo para acesso simultâneo (STEFAN, 2009, p. 11). Além disso, VCS melhora a cooperação e colaboração dentro e entre equipes através do fornecimento da capacidade de confirmar, comparar, mesclar e restaurar versões anteriores de objetos no repositório (DAVIS; DANIELS, 2016).

Os sistemas de controle de versão podem ser centralizados (*Centralized Version Control System*, CVCS) ou distribuídos (*Distributed Version Control System*, DVCS). No modelo centralizado, um repositório único organiza os artefatos e é acessado via LAN ou WAN. Já no modelo distribuído, cada usuário possui um repositório completo localmente. A rede é necessária apenas para sincronizações, como mesclagens e compartilhamento de repositórios (STEFAN, 2009, p. 11–12).

2.4.1. Git

O Git, exemplo de DVCS, surgiu da necessidade na comunidade de desenvolvimento do kernel do Linux. Inicialmente adotando o BitKeeper em 2002, a comunidade viu-se confrontada em 2005 pela revogação do status gratuito dessa ferramenta (CHACON; STRAUB, 2014).

Desenvolvido sob a liderança de Linus Torvalds, o Git surgiu como uma alternativa distribuída, rápida e eficiente, com suporte robusto ao desenvolvimento não linear e capacidade de gerenciar projetos complexos, como o kernel do Linux (CHACON; STRAUB, 2014). Trata-se de um sistema distribuído, seguro, leve, econômico e de código aberto (GHODKE; CHAVAN, 2024).

O fluxo de trabalho no Git é composto por quatro fases que promovem um controle de versão eficiente e uma colaboração contínua, garantindo que as alterações sejam realizadas, revisadas e integradas de forma eficaz em um sistema de controle de versão ao longo do processo de desenvolvimento (GHODKE; CHAVAN, 2024).

1) **Diretório de Trabalho** (*Working Directory*): O diretório de trabalho corresponde ao local onde o projeto se encontra e o código recebe alterações. Nesse ambiente, os arquivos passam por modificações ou são criados, permanecendo fora do gerenciamento do sistema de versionamento até o momento de sua inclusão em etapas posteriores.
2) **Área de Preparação** (*Staging Area*): A área de preparação, também conhecida como *index*, permite organizar e revisar arquivos modificados ou novos antes de sua inclusão em um *commit*. Esse espaço funciona como um intermediário entre o diretório de trabalho e o repositório local, facilitando o gerenciamento das mudanças. Equipes utilizam essa área para revisões de código, garantindo uma preparação adequada antes do registro no histórico do projeto.
3) **Repositório Local** (*Local Repository*): O repositório local armazena os *commits* realizados na máquina do desenvolvedor, antes do envio ao repositório central. Localizado na pasta *.git* do projeto, esse ambiente mantém o histórico completo das alterações realizadas, identificando cada *commit* por meio de hashes exclusivos que incluem informações como autor,

data e mensagem. Essa estrutura permite acesso e gerenciamento direto do histórico em sistemas de controle de versão distribuídos.

4) **Repositório Central** (*Central Repository*): O repositório central, também chamado de repositório remoto, serve como ponto de sincronização entre os membros da equipe. Geralmente hospedado em plataformas como GitHub, GitLab ou Bitbucket, esse ambiente centraliza o projeto, permitindo que desenvolvedores enviem (*push*) suas alterações locais e recuperem (*pull*) contribuições de outros integrantes, facilitando a colaboração em equipe e o gerenciamento integrado de mudanças.

2.4.2. Serviços de Hospedagem Git

Os serviços de hospedagem Git são plataformas projetadas para centralizar a gestão e o armazenamento de repositórios Git. Utilizando o sistema de controle de versão distribuído Git, esses serviços oferecem diversas ferramentas que facilitam o controle de versão, a colaboração entre equipes e o gerenciamento de projetos (GHODKE; CHAVAN, 2024).

GitLab, por exemplo, é uma plataforma DevOps que integra diversas ferramentas para gerenciar todo o ciclo de vida do desenvolvimento de software. Fundada em 2011, oferece versões tanto auto-hospedadas quanto na nuvem, atendendo desde pequenos projetos até soluções empresariais de grande escala (GHODKE; CHAVAN, 2024).

Com pipelines CI/CD integrados, o GitLab automatiza processos como construção, teste e implantação, permitindo a entrega contínua com configurações personalizáveis. Além disso, a plataforma fornece ferramentas para o gerenciamento de marcos, rastreamento de problemas e quadros Kanban, otimizando o gerenciamento de projetos (GHODKE; CHAVAN, 2024).

O uso de *merge requests* no GitLab facilita a colaboração nas revisões de código, permitindo que membros da equipe sugiram melhorias e aprovem alterações. A plataforma também oferece suporte ao gerenciamento e à implantação de aplicativos conteinerizados, com integração com o Kubernetes. GitLab é altamente configurável e extensível, permitindo inúmeras integrações de terceiros e personalizações (GHODKE; CHAVAN, 2024).

2.5. Webhooks

Um *webhook* representa uma comunicação baseada em eventos e de baixo custo, que transmite dados automaticamente entre aplicativos por meio de Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol* – HTTP). Acionado por eventos específicos, o *webhook* automatiza a interação entre APIs – conjunto de definições e protocolos para construir e integrar software de aplicativo – e pode ser utilizado para iniciar fluxos de trabalho, como em ambientes GitOps (REDHAT, 2024).

Hooks no Git consistem em programas que são acionados quando o Git executa uma ação específica, como ao tentar adicionar um *commit* ao histórico. Eles podem ser escritos em qualquer linguagem de programação interpretada pelo servidor, como bash, Ruby ou PHP (HETHEY, 2013).

2.6. Arquitetura Producer-Worker

Arquitetura Producer-Worker constitui um modelo amplamente adotado em sistemas de filas de tarefas distribuídas. Nesse modelo, o produtor coloca as solicitações de trabalho (tarefas) em filas específicas, chamadas de *task queues*. Após inserir as tarefas, o produtor recupera os resultados das tarefas a partir do *result backend*, área que armazena os resultados das operações executadas pelos trabalhadores (PIERFEDERICI, 2016).

Os processos trabalhadores (*workers*) se inscrevem em algumas ou todas as filas de tarefas, identificando quais trabalhos realizar. Eles executam as tarefas e armazenam os resultados novamente no *result backend*. Esse modelo permite que os trabalhadores e produtores operem sem precisar conhecer a quantidade ou a localização dos outros (PIERFEDERICI, 2016).

2.7. Tecnologias

A engenharia de software envolve todas as etapas do desenvolvimento de sistemas, desde a definição inicial dos requisitos até a manutenção e o gerenciamento do sistema em operação. A implementação, reconhecida como a

etapa mais crítica, corresponde ao momento em que uma versão funcional do software é desenvolvida (SOMMERVILLE, 2018).

A implementação pode incluir o desenvolvimento de programas utilizando linguagens de programação de alto ou baixo nível, além da personalização e adaptação de sistemas prontos, ajustando-os para atender às necessidades específicas de uma organização (SOMMERVILLE, 2018).

2.7.1. Django

O Django, *framework* Python, baseia-se no princípio “não se repita” (*Don't Repeat Yourself* – DRY), que reduz redundâncias no software e incentiva a reutilização de componentes. Para isso, organiza funcionalidades em módulos chamados "apps", pacotes independentes reutilizáveis em diversos projetos (WETHERBY, 2024).

Adotando a arquitetura Model-View-Template (MVT), uma variação da Model-View-Controller (MVC), o *framework* gerencia interações com bancos de dados na camada *Models*, processa a lógica de negócios nas *Views* e apresenta os dados ao usuário por meio dos *Templates* (WETHERBY, 2024).

O Django integra diversas ferramentas, incluindo um *Mapeamento Objeto-Relacional* (ORM) compatível com múltiplos bancos de dados, um despachante de URLs intuitivo, suporte a *middleware* e um sistema completo de autenticação. Um diferencial importante envolve a interface administrativa, que oferece uma solução pronta para uso (WETHERBY, 2024).

Além disso, segundo Wetherby (2024), o *framework* oferece escalabilidade para atender projetos de diferentes portes, garantindo alta disponibilidade e adaptação às demandas de desempenho. Sua robustez em segurança inclui proteções contra ameaças como *SQL injection*, *Cross-Site Scripting (XSS)*, *Cross-Site Request Forgery (CSRF)* e *Clickjacking*, além de sistemas avançados de autenticação e controle de acesso.

2.7.2. Django REST Framework (DRF)

O *Django REST Framework*, conhecido como DRF, trata-se de um pacote auxiliar para construir *APIs Web* que integra-se ao Django. Oferece uma variedade de componentes prontos para uso, como: *Class-Based REST Views*, *Viewsets* e *Serializers* (GAGLIARDI, 2021).

C*lass-Based REST Views* fundamenta-se em *Class-Based Views* do Django, de modo a evitar a repetição de padrões comuns no tratamento de dados (exibição de formulários, validação de entradas e salvamento no banco de dados) em APIs REST (GAGLIARDI, 2021).

Viewsets oferecem uma abstração inteligente sobre as visualizações baseadas em classes, permitindo que o Django se torne ainda mais orientado a recursos. Eles facilitam a exposição de operações *Create/Read/Update/Delete* (CRUD) de maneira mais eficiente, tornando o desenvolvimento de APIs REST mais intuitivo e simplificado (GAGLIARDI, 2021).

Os *Serializers* são responsáveis por converter objetos Python em formatos como JSON, permitindo que esses dados sejam facilmente enviados em uma API REST. Eles atuam como uma ponte entre os modelos do Django e a comunicação com o usuário final por meio da API, facilitando a serialização (transformação de objetos em dados de troca) e a desserialização (transformação de dados recebidos de volta em objetos utilizáveis) (GAGLIARDI, 2021).

2.7.3. Django Simple History

O pacote *Django Simple History* armazena o estado do modelo Django a cada criação, atualização e exclusão. Para isso, cria um modelo que inclui todos os campos da instância do modelo base, assim como incorpora vários metadados, como: identificador da história (*history_id*), tipo da ação (*history_type*), usuário que fez a ação (*history_user*), data e horário da ação (*history_date*) e motivo da ação (*history_change_reason*) (COREY BERTRAM, 2024).

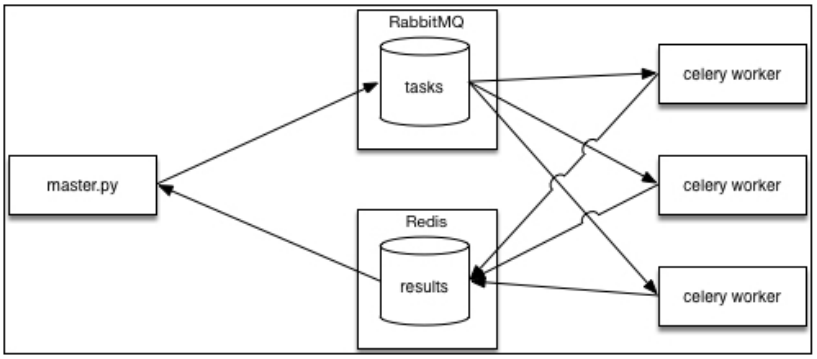
2.7.4. Celery

O Celery caracteriza-se como uma biblioteca usada para gerenciar filas de tarefas distribuídas. Ela permite que sistemas baseados em filas sejam executados

de forma distribuída, ou seja, os processos de trabalho e as filas que armazenam os resultados e as solicitações podem ser executados em máquinas diferentes (PIERFEDERICI, 2023).

A Figura 2 apresenta a arquitetura de um aplicativo típico com Celery, utilizando RabbitMQ e Redis. Cada processo nas caixas retangulares (RabbitMQ, Redis, celery worker e master.py) executa-se em máquinas distintas. Em configurações mais simples, o RabbitMQ e o Redis costumam ser hospedados no mesmo servidor, com um ou dois nós para os trabalhadores. Em instalações maiores, distribui-se os processos por várias máquinas e, eventualmente, utilizam-se servidores dedicados (PIERFEDERICI, 2023).

Figura 2 – Exemplo de uso do Celery



Fonte: Pierfederici (2023).

2.7.5. Redis

O Redis (*REmote DIctionary Server*), desenvolvido em C por Salvatore Sanfilippo em 2006, classifica-se como um banco de dados NoSQL avançado baseado no modelo chave-valor. Também recebe a denominação de servidor de estruturas de dados devido à capacidade de manipular tipos de dados poderosos, como *Strings*, *Hashes*, *Lists*, *Sets*, *Sorted Sets*, *Bitmaps* e *HyperLogLogs*. Além disso, oferece funcionalidades como expiração configurável de chaves, transações e recursos de *publish*/*subscribe* (SILVA; TAVARES, 2015).

Por padrão, o Redis armazena dados na memória, garantindo operações de leitura e gravação extremamente rápidas. No entanto, permite também a persistência no disco. A persistência pode ocorrer por meio da criação de um *snapshot* binário dos dados ou de um arquivo legível contendo a sequência de

comandos executados ao longo do tempo. Esses processos recebem os nomes de *snapshotting* e *journaling*, respectivamente (SILVA; TAVARES, 2015).

2.7.6. PostgreSQL

PostgreSQL trata-se de um banco de dados relacional de código aberto (licença *Berkeley Software Distribution*, BSD), amplamente reconhecido por sua estabilidade, escalabilidade e segurança. Inicialmente desenvolvido como "POSTGRES" (*POST-Ingres*) por Michael Stonebraker na Universidade da Califórnia, Berkeley, o projeto evoluiu para oferecer um sistema de gerenciamento de banco de dados robusto e flexível, capaz de suportar grandes volumes de dados e operações complexas (FERRARI; PIROZZI, 2023).

Desde 1996, o projeto passou a se chamar PostgreSQL, refletindo seu foco no suporte a SQL e suas capacidades avançadas. Atualmente o *PostgreSQL Global Development Group* (PGDG) o mantém, garantindo atualizações constantes e recursos poderosos. Projetado para ser uma plataforma extensível, permitindo a adição de novos comportamentos e funcionalidades, como linguagens embutidas e extensões para necessidades específicas (FERRARI; PIROZZI, 2023).

3. TRABALHOS RELACIONADOS

Este capítulo apresenta o estado da arte associada à revisão de código moderna. Buscou-se identificar contribuições relevantes e lacunas em ferramentas para gerenciar MCR. Os trabalhos analisados foram selecionados com base em três critérios, a saber: estudos publicados nos últimos 15 anos, estudos que apresentam resultados práticos aplicáveis ao contexto de revisão de código e publicações em periódicos e conferências reconhecidas na área.

3.1. Ferramentas de Revisão de Código Moderna

A revisão de código moderna, conforme visto no capítulo anterior, fundamenta-se na prática informal, assíncrona, focada em inspeções de novas alterações propostas e com uso de ferramentas (RIGBY; BIRD, 2013). Nesta seção, serão apresentados estudos relevantes nessa área, com descrição de ferramentas utilizadas, como: Critique, CodeFlow, Gerrit, Review Board, Rietveld, Caesar, SCRUB e Please Review.

3.1.1. Critique

Sadowski *et al.* (2018) realizaram um estudo de caso no Google para investigar as motivações por trás da revisão de código, as práticas adotadas pelos desenvolvedores e os desafios enfrentados nesse processo. O estudo incluiu 12 entrevistas, uma pesquisa aplicada a 44 participantes e a análise de logs de revisão que abrangeram 9 milhões de alterações.

No estudo, os autores também descrevem o processo de revisão de código no Google, que ocorre em um repositório de código-fonte monolítico acessado por meio de um sistema interno de controle de versão. A revisão de código é obrigatória na empresa, e todo *commit* deve passar pela ferramenta Critique, uma solução interna, centralizada e baseada na *web*, desenvolvida especificamente para gerenciar revisões de código.

Segundo Sadowski *et al.* (2018), a ferramenta permite que os revisores analisem as diferenças (*diffs*) nas alterações propostas no código e iniciem

discussões encadeadas diretamente em linhas específicas, envolvendo tanto os autores das alterações quanto outros revisores.

Além disso, Sadowski *et al.* (2018) destacam que o Critique registra todas as interações dos desenvolvedores com a ferramenta, como abrir o sistema, visualizar *diffs*, fazer comentários e aprovar alterações. Esse registro detalhado fornece um histórico completo de atividades relacionadas à revisão de código, permitindo tanto a análise de processos quanto a identificação de padrões de uso.

O Critique moldou-se pela cultura de desenvolvimento do Google, que inclui a revisão de código como parte central do fluxo de trabalho (WINTERS; WRIGHT; MANSHREK, 2020). Essa influência cultural se traduz em um conjunto de princípios, como:

- Simplificada: A interface do Critique é rápida, intuitiva e eficiente, com navegação fácil, suporte a teclas de atalho e indicadores visuais claros para o status da revisão;
- Confiança: A revisão de código é baseada na confiança mútua, permitindo mudanças abertas e promovendo agilidade sem exigir verificações adicionais para ajustes menores;
- Comunicação Genérica: O Critique incentiva comentários claros e objetivos, evitando complexidades desnecessárias e focando na comunicação direta entre os desenvolvedores;
- Integração de Fluxo de Trabalho: A ferramenta se conecta a outras plataformas de desenvolvimento, facilitando a navegação, edição de código e validação de testes diretamente no fluxo de trabalho.

3.1.2. CodeFlow

Bacchelli e Bird (2013) realizaram um estudo envolvendo desenvolvedores, testadores e gerentes da Microsoft, que atuam em diversas áreas, como soluções de gerenciamento de dados empresariais, aplicativos para smartphones e mecanismos de busca. Cada equipe na empresa possui sua própria cultura de desenvolvimento e políticas de revisão de código. O foco do estudo foi na ferramenta CodeFlow, amplamente adotada na Microsoft e utilizada por mais de 40.000 desenvolvedores à época da pesquisa.

Segundo os autores, o CodeFlow é uma ferramenta colaborativa de revisão de código que permite aos usuários anotar diretamente o código-fonte e interagir com outros participantes por meio de um modelo de chat ao vivo. O processo começa com o autor criando um pacote contendo os arquivos alterados (novos, excluídos e modificados), selecionando os revisores, escrevendo uma mensagem descritiva e submetendo o pacote ao serviço da ferramenta. Em seguida, a CodeFlow notifica os revisores por e-mail sobre a nova tarefa.

Os revisores interagem com a ferramenta por meio de uma única janela *desktop*, onde visualizam a lista de arquivos alterados, a descrição textual da mudança e o status de cada revisor. O painel principal exibe as diferenças (*diffs*) nos arquivos revisados. Tanto autores quanto revisores podem destacar trechos do código e adicionar comentários em linha, que geram discussões. Essas interações podem ser síncronas, funcionando como mensagens instantâneas, ou assíncronas, com os comentários armazenados para revisões posteriores.

3.1.3. Gerrit

McIntosh *et al.* (2014) conduziram um estudo sobre a relação entre qualidade de software e dois aspectos chave do processo moderno de revisão de código: a cobertura de revisão de código, que representa a proporção de alterações que foram devidamente revisadas, e a participação na revisão de código, que reflete o grau de envolvimento dos revisores no processo. O estudo foi realizado por meio de um estudo de caso nos projetos de código aberto Qt, VTK e ITK.

McIntosh *et al*. (2014) descrevem o Gerrit[1] como uma ferramenta moderna de revisão de código que facilita um processo rastreável para projetos baseados em Git. O Gerrit integra-se estreitamente a ferramentas de automação de testes e integração de código. No processo, os autores enviam *patches* (conjuntos de alterações propostas) para o servidor Gerrit, e os revisores podem ser selecionados de três formas: convidados pelos autores, designados automaticamente com base em sua expertise ou auto-selecionados por meio de listas de discussão.

Os revisores avaliam as alterações propostas, deixam comentários, atribuem pontuações e podem solicitar revisões adicionais. Paralelamente, verificadores, que

[1] https://www.gerritcodereview.com

podem ser membros da equipe ou ferramentas de Integração Contínua (CI), testam os patches para garantir que corrigem defeitos ou adicionam funcionalidades sem causar regressões. Tanto revisores quanto verificadores podem atribuir pontuações que influenciam a decisão de integração das alterações.

O Gerrit permite automatizar critérios para integrar alterações aos repositórios principais, como a exigência de aprovações positivas de pelo menos um revisor e um verificador. Após atender aos critérios, os patches são automaticamente integrados ao sistema.

3.1.4. Review Board

O Review Board[2], conforme Rawat (2014), trata-se de uma ferramenta de código aberto baseada na web utilizada para otimizar o processo de revisão de código em equipes de desenvolvimento. As principais características incluem integração com sistemas de controle de versão, como Git, Subversion e Mercurial, suporte a publicação automática de revisões via API REST ou *scripts* Python e uma interface intuitiva para visualização de diferenças no código.

Segundo Rawat (2014), ela engloba:

- Publicação de Revisões: Permite criar e compartilhar solicitações de revisão por meio de uma interface web ou linha de comando com facilidade;
- Revisão de Código Facilitada: Oferece um diff viewer intuitivo que destaca alterações no código com visualização lado a lado;
- Comentários e Colaboração: Permite comentários em linhas, grupos ou textos, e discussões sobre eles;
- Rastreamento de Solicitações: Acompanha solicitações, notifica revisores e exibe pendências em um painel;
- Integração com Sistemas de Controle de Versão: Compatível com diversas ferramentas de controle de versão, garantindo flexibilidade e adaptabilidade;
- Revisão de Arquivos Não-Codificados: Suporte a diferentes tipos de arquivos além do código, permitindo revisões abrangentes;
- Controle Administrativo: Fornece interface web para gerenciar usuários, repositórios e sistema, facilitando o trabalho dos administradores.

[2] https://www.reviewboard.org

3.1.5. Rietveld

Biase, Bruntink e Bacchelli (2016), exploraram o processo de revisão de código moderna no Chromium, um projeto de código aberto, em busca de analisar como a MCR aborda problemas de segurança e identificar os fatores que influenciam sua capacidade de detectar vulnerabilidades no código.

No estudo, Biase, Bruntink e Bacchelli (2016) descrevem a ferramenta utilizada no Chromium: o Rietveld[3]. Segundo os autores, essa ferramenta permite que as revisões sejam emitidas diretamente a partir do repositório de código, facilitando o processo de revisão de alterações no software.

O processo de revisão começa quando o desenvolvedor cria uma *change list* após realizar as modificações no código e as envia para iniciar a revisão. O autor da alteração seleciona pelo menos um revisor, geralmente baseado nas últimas modificações feitas no código, e define um responsável (*owner*) pelo subsistema alterado, que tem a tarefa de garantir a qualidade do código.

Durante a revisão, os revisores podem visualizar as diferenças entre a versão original e a proposta (*diff*) e inserir comentários em linha para iniciar discussões. Se forem sugeridas alterações, o desenvolvedor pode enviar uma nova versão do *patch*, reiniciando o ciclo de feedback. Para que o patch seja integrado, o responsável pelo subsistema deve aprová-lo com um "Looks Good to Me" (LGTM).

3.1.6. Caesar

Tang (2011) apresenta Caesar, uma ferramenta de revisão de código distribuída e social, projetada para atender às necessidades e restrições de um curso de programação. Caesar possui a capacidade de escalar para uma grande quantidade e diversificada base de revisores, oferecendo ferramentas automatizadas que aumentam a eficiência na revisão e implementando uma *interface web* que estimula a discussão e a participação dos usuários.

Conforme Tang (2011), Caesar compõem-se em três componentes principais e fracamente acoplados: um pré-processador de código específico para a linguagem, que divide o código em partes menores, filtra as partes irrelevantes,

[3] https://github.com/rietveld-codereview/rietveld

realiza análise estática e identifica clusters de código semelhantes; um roteador de tarefas incremental, que atribui dinamicamente os revisores às tarefas; e uma *interface web* independente de linguagem para facilitar a revisão do código.

3.1.7. SCRUB

Holzmann (2010) descreve a ferramenta *Source Code Review User Browser* (SCRUB[4]), projetada para melhorar o processo de revisão de código por meio de ferramentas. Ela visa apoiar o desenvolvimento de software crítico em equipe no *Jet Propulsion Laboratory* (JPL, NASA), mas também pode ser aplicada em projetos menores e de desenvolvimento individual.

A ferramenta combina revisão clássica de código por pares com análises automáticas, utilizando uma variedade personalizável de analisadores de código-fonte. Todos os relatórios gerados, seja por humanos ou ferramentas automáticas, são acessados através de uma interface única e uniforme fornecida pelo SCRUB.

Segundo Holzmann (2010), o processo de revisão de código suportado pela ferramenta SCRUB consiste em três fases:

I. Revisão: Os revisores pares estudam o código e registram seus comentários e observações na ferramenta SCRUB;
II. Resposta: O desenvolvedor responde a todos os relatórios, indicando concordância, discordância ou a necessidade de mais informações.
III. Resolução: Resolve as pendências, incluindo discordâncias entre comentários de revisores e relatórios da ferramenta.

3.1.8. Please Review

Filho (2017) propôs o Please Review, uma ferramenta integrada ao GitHub que centraliza e organiza as solicitações de revisão de código em um único canal de informações. Ela cria um feed de solicitações, proporcionando maior visibilidade aos pedidos de revisão e, assim, otimizando o processo de revisão de código dentro de uma organização.

[4] https://github.com/nasa/scrub

De acordo com Filho (2017), o Please Review oferece um conjunto de seis funcionalidades para otimizar o fluxo moderno de revisão de código, a saber:

- Autenticar-se no GitHub: Permite que a ferramenta acesse dados de usuários, repositórios e *pull requests* com as permissões necessárias;
- Cadastrar solicitação de revisão: O usuário autenticado seleciona o repositório, escolhe o *pull request* e registra a solicitação;
- Visualizar solicitação de revisão: Exibe no feed título, revisões, commits, linhas alteradas, linguagens, solicitante e repositório;
- Abrir ferramenta de revisão de código do GitHub: Cada solicitação possui um link que direciona o revisor para a ferramenta de revisão de código;
- Encerrar solicitação de revisão: O solicitador pode encerrar a solicitação a qualquer momento, removendo-a do feed, se satisfeito com as revisões;
- Fechar a sessão: Os dados da sessão do usuário estão no canto superior direito, e ele pode encerrar a sessão clicando em "Logout".

3.2. Análise comparativa

Nesta seção, apresenta-se uma análise comparativa das ferramentas Critique, CodeFlow, Gerrit, Review Board, Rietveld, Caesar, SCRUB e Please Review, discutidas neste trabalho. Busca-se avaliar suas características, vantagens e limitações.

A análise concentra-se nos seguintes critérios: presença de notificações, tipos de interface, integração com sistemas de controle de versão, estilo de processo de MCR, modo de comunicação (síncrono, assíncrono ou híbrido), existência de prazos (*deadlines*) para o processo MCR e o tipo de licença.

- **Critique**: Desenvolvido como ferramenta interna do Google para atuar com Piper (repositório monolítico de código-fonte), ele proporciona uma interface de usuário fundamentada na simplificada de (fácil navegação, carregamento rápido, teclas de atalho, marcadores visuais), comunicação assíncrona e prover notificações por e-mail e pelo navegador, através da instalação de plugin. Caracteriza-se por contemplar um processo MCR em seis etapas: criar uma alteração, solicitar revisão, adicionar comentários, modificar alteração/responder aos comentários, aprovar alterações e commit de

alteração. Como limitação, observa-se a ausência de suporte a prazos. (SADOWSKI; KURNIA; ROHLFS, 2020).

- **CodeFlow**: Criado como ferramenta interna da Microsoft para auxiliar no processo de revisão, ela caracteriza-se por ser uma aplicação com interface *desktop*, possuir notificação através de e-mail e integra-se com o Git. Adota um processo de quatro etapas: criar pacote contendo os arquivos alterados, selecionar os revisores, submeter solicitação e interagir com solicitação. A comunicação, um aspecto central, pode ser conduzida de forma síncrona ou assíncrona. Como limitação, destaca-se a ausência de *deadlines*. (BACCHELLI; BIRD, 2013).
- **Gerrit**: Ferramenta de código aberto, amplamente utilizada em projetos como Qt, VTK e ITK, diferencia-se por oferecer uma interface web, aliada à flexibilidade de uso via linha de comando (CLI), permitindo maior adaptação às preferências dos usuários. Integra-se ao Git, conta com um sistema de notificação por e-mail e suporta comunicação assíncrona. Adota-se um processo com três etapas: enviar *patches*, selecionar revisores e avaliar as alterações propostas. O Gerrit permite configurar a seleção de revisores e esquema de pontuação para determinar o estado da solicitação. Como limitação, evidencia-se a falta de suporte para prazos. (MCINTOSH *et al*., 2014).
- **Review Board**: Ferramenta de código aberto com foco na revisão não apenas de coleções de código-fonte, mas também de capturas de tela, documentação, esquemas, diagramas e outros artefatos. Integra-se a diversos sistemas de controle de versão, como Git, Mercurial, Perforce, Subversion e Bazaar/Breezy. Além disso, oferece suporte a integrações com Redes Sociais Corporativas, como Discord, Slack e Microsoft Teams, entre outras. A ferramenta disponibiliza uma interface web e suporte por linha de comando. A comunicação, um ponto de destaque, pode ser síncrona ou assíncrona, e o processo de revisão pode ser adaptado às regras específicas de cada projeto. Como limitação, percebe-se a ausência de suporte para *deadlines*. (RAWAT, 2014).
- **Rietveld**: Ferramenta de código aberto utilizada no Chromium que integra-se exclusivamente ao Subversion. Disponibiliza uma interface web

com comunicação assíncrona. Oferece um mecanismo de notificações por intermédio de e-mail. Contempla um processo bem definido para revisão: cria uma lista de alterações, definir revisores, escolher responsável pelo subsistema alterado e analisar lista de alterações. Como limitação, ressalta-se a falta de suporte para prazos. (BIASE; BRUNTINK; BACCHELLI, 2016).

- **Caesar**: Ferramenta de código aberto, com foco na revisão de código social distribuída, projetada para atender às restrições e objetivos específicos de um curso de programação. Embora não possua integração com VCS, permite comunicação síncrona e assíncrona, por meio de uma interface web. Destaca-se por realizar o processo em duas etapas: pré-processamento das alterações e roteamento de tarefas. Sua limitação está na falta de um mecanismo de notificação, assim como no suporte para *deadlines*. (TANG, 2011).
- **SCRUB**: Ferramenta de código aberto desenvolvida para apoiar o desenvolvimento de software crítico na equipe do Jet Propulsion Laboratory. Caracteriza-se por oferecer uma interface *desktop* com comunicação assíncrona. Embora não tenha integração com VCS nem mecanismo de notificação, destaca-se por permitir o controle do tempo no processo de MCR, que é estruturado em três fases: revisão, resposta e resolução. (HOLZMANN, 2010).
- **Please Review**: Ferramenta proprietária que se integra ao GitHub. Disponibiliza uma interface web com comunicação assíncrona para centralizar e organizar as solicitações de revisão de código. Apresenta um processo bem definido: cadastrar solicitação de revisão, visualizar solicitação de revisão e encerrar solicitação de revisão. Sua limitação está na ausência de um mecanismo de notificação e na falta de definição de prazo para consolidar a solicitação de revisão. (FILHO, 2017).

Assim, busca-se desenvolver um sistema de código aberto, denominado Smart Review, que se integre às redes sociais empresariais por meio de chatbot, incorporando mecanismos de notificação. O objetivo é fornecer uma interface API que simplifique a integração tanto entre os serviços de hospedagem de repositórios com função de revisão de código quanto com os sistemas de comunicação.

Conforme observado em ferramentas como CodeFlow, Gerrit, Review Board e Please Review, o VCS mais utilizado é o Git. Dessa forma, almeja-se que a ferramenta se integre com serviços de hospedagem de repositórios Git. Assim como o Review Board, a ferramenta deve permitir a configuração do processo de revisão, com definição de etapas e regras para acionamento. De forma parecida ao SCRUB, ela também deve possibilitar a definição de *deadlines*.

O Quadro 1 resume as características das ferramentas para revisão de código. Ele está organizado em oito colunas: ferramenta, notificação, interface, VCS, processo, comunicação, prazo e licença. Cada linha corresponde a uma ferramenta, detalhando suas características associadas.

Quadro 1 – Comparativo de ferramentas para revisão de código

Ferramenta	**Notificação**	**Interface**	**VCS**	**Processo**	**Comunicação**	**Deadline**	**Licença**
Critique	Navegador E-mail	Web	Piper	Fixo	Assíncrona	N/S	Proprietária
CodeFlow	E-mail	Desktop	Git	Fixo	Híbrida	N/S	Proprietária
Gerrit	E-mail	Web CLI	Git	Fixo	Assíncrona	N/S	Código Aberto
Review Board	ESN	Web CLI	Git Mercurial Perforce Subversion Bazaar Breezy	Adaptável	Híbrida	N/S	Código Aberto
Rietveld	E-mail	Web	Subversion	Fixo	Assíncrona	N/S	Código Aberto
Caesar	N/S	Web	N/S	Fixo	Híbrida	N/S	Código Aberto
SCRUB	N/S	Desktop	N/S	Fixo	Assíncrona	Controlável	Código Aberto
Please Review	N/S	Web	Git	Fixo	Assíncrona	N/S	Proprietária
Smart Review	ESN	API	Git	Adaptável	Híbrida	Controlável	Código Aberto

Fonte: Elaborado pelo autor (2024).

4. SMART REVIEW: COMUNICAÇÃO DE REVISÃO DE CÓDIGO

No capítulo anterior, os trabalhos correlatos foram analisados, destacando lacunas importantes, como a ausência de um mecanismo de notificação eficiente, a customização do processo de revisão de código moderna e a inclusão de *deadlines*. Com base nesses desafios, a solução proposta, intitulada **Smart Review**, foi desenvolvida com o objetivo de automatizar a comunicação das etapas presentes na MCR através do uso de *webhook* e *chatbot*.

Neste capítulo, o desenvolvimento da solução será aprofundado. Em primeiro momento, será abordado o processo de criação do software, com detalhamento das etapas de elicitação de requisitos, *design* de projeto e implementação. Por fim, as limitações da solução serão discutidas, destacando-se os pontos que podem ser aprimorados.

4.1. Smart Review: Processo de Criação

O desenvolvimento de software é uma atividade complexa e multifacetada que envolve uma série de etapas bem definidas para transformar uma ideia em uma solução funcional. A construção do Smart Review segue uma abordagem estruturada, conforme descrito por Sommerville (2018), com foco nas melhores práticas para garantir a entrega de um produto de alta qualidade.

O processo escolhido considera três etapas, a saber: elicitação de requisitos, *design* de projeto e implementação. Além disso, utiliza-se uma abordagem iterativa para garantir flexibilidade e adaptação contínua ao longo do ciclo de vida do software.

4.1.1. Etapa I: Elicitação de Requisitos

A elicitação de requisitos constitui a etapa inicial do processo de desenvolvimento do Smart Review. Nessa fase, busca-se identificar, compreender e documentar as necessidades e expectativas dos *stakeholders*, considerando tanto os aspectos funcionais quanto os de qualidade do sistema.

A começar pelos *stakeholders* — pessoas ou organizações que influenciam diretamente os requisitos do sistema (POHL; RUPP, 2012) —, sua participação é fundamental para o desenvolvimento bem-sucedido de um projeto, pois são eles que fornecem as informações necessárias para a definição precisa dos requisitos.

A cultura organizacional do projeto de pesquisa SmartSuite influenciou a definição dos requisitos, com a participação de desenvolvedores front-end, profissionais com perfil DevSecOps, squad leader e coordenador. Como parte dessa cultura, adota-se a revisão de código moderna como prática interna, realizada por desenvolvedores com perfis semelhantes (front-ends ou back-ends), que assumem os papéis de autor e revisor.

Esse processo é apoiado por ferramentas como o sistema de controle de versão GitLab e a plataforma de comunicação Slack. Ambas são mantidas e otimizadas por administradores com perfil de DevSecOps, responsáveis pela gestão e aprimoramento contínuo dessas ferramentas.

Com base nesse contexto, destacam-se três stakeholders principais: autor, revisor e administrador. O Quadro 2 apresenta uma descrição detalhada de cada um desses perfis, destacando suas responsabilidades no processo de revisão de código moderna.

Quadro 2 – Stakeholders

Perfil	Descrição
Autor	Desenvolvedor que pretende criar uma solicitação de revisão de código em busca de averiguar se o artefato proposto atende os padrões pré definidos pela organização de atuação e não possui inconsistências. Aprovação efetuada por um terceiro com papel de revisor.
Revisor	Pessoa, entidade ou equipe, geralmente um desenvolvedor, que assume a responsabilidade por analisar uma solicitação de revisão de código com objetivo de verificar os padrões pré definidos pela organização, além de identificar eventuais problemas.
Administrador	Pessoa, entidade ou equipe, de preferência portadora de conhecimentos de DevOps, que possui autoridade e responsabilidade sobre o projeto em execução. Ele(a) tem acesso privilegiado para realizar ações administrativas, gerenciar recursos e garantir que os processos estejam alinhados com os objetivos e normas estabelecidos. No contexto atual, possui a missão de configurar e manter o sistema de controle de versão, sistema de comunicação e automação de comunicação associada ao processo de revisão de código.

Fonte: Elaborado pelo autor (2024).

Cada stakeholder impacta, de alguma forma, os requisitos do sistema, especialmente os funcionais. Esses requisitos descrevem os resultados esperados de comportamentos específicos que devem ser fornecidos pelas funções do sistema,

alinhando-se às necessidades e objetivos definidos pelos envolvidos (POHL; RUPP, 2012).

Por exemplo, os autores e revisores, conforme visto na seção de Revisão de Código Moderna, estão constantemente se comunicando conforme certo fluxo de revisão de código predefinido pela organização, empresa ou instituição do projeto. Muitas vezes isso ocorre várias vezes durante suas jornadas de trabalho e com troca de informações repetitivas.

Esse comportamento poderia ser convertido em uma necessidade de automatizar o processo de comunicação, na qual o sistema de controle de versão poderia publicar mensagens padronizadas sobre eventos associados a revisão diretamente no sistema de comunicação.

Os requisitos nessa situação, assim como outros, estão detalhados no Quadro 3, que reúne as funcionalidades de alta prioridade relacionadas aos *stakeholders*. O quadro está organizado em três colunas: identificação, descrição e *stakeholders*. A identificação segue o padrão “RF” mais número do requisito funcional. A descrição obedece ao padrão de escrita de requisito definido por Pohl e Rupp (2012). Por fim, os *stakeholders* são aqueles que possuem algum nível de envolvimento com o requisito.

Quadro 3 – Requisitos Funcionais

Identificador	Descrição	Stakeholders
RF01	O sistema deverá fornecer para o administrador capacidade de integrar um gerenciador de repositório baseado em GIT e com webhooks para atuar como consumidor.	Administrador
RF02	O sistema deverá fornecer para o administrador capacidade de utilizar uma ferramenta de comunicação baseada em canais e com suporte para robot.	Administrador
RF03	O sistema deverá fornecer para o administrador a capacidade de configurar etapas, com suas respectivas regras, do processo de revisão de código.	Administrador
RF04	O sistema deverá fornecer para o administrador a capacidade de gerenciar os desenvolvedores com funções de autor e/ou revisor no processo de revisão de código.	Administrador
RF05	O sistema deverá fornecer para o gerenciador de repositório baseado em GIT a capacidade de publicar uma atualização de revisão de código.	Autor, Revisor
RF06	O sistema deverá controlar solicitações de revisão de código sem conclusões e com tempo expirado.	Autor, Revisor
RF07	O sistema deverá verificar as solicitações de revisão a cada uma hora.	Autor, Revisor

Identificador	Descrição	Stakeholders
RF08	O sistema deveria fornecer para o administrador a capacidade de utilizar o slack como ferramenta de comunicação.	Administrador
RF09	O sistema deveria fornecer para o administrador a capacidade de utilizar um modelo padrão para o processo de revisão de código.	Administrador
RF10	O sistema deveria ser capaz de manter o histórico de eventos associados a revisão de código.	Administrador
RF11	O sistema deveria ser capaz de mencionar os envolvidos na notificação de um evento.	Autor, Revisor

Fonte: Elaborado pelo autor (2024).

A definição dos onze requisitos funcionais baseou-se na percepção das necessidades dos envolvidos, obtida por meio da análise do processo existente (mapeado no canal de revisão) e de reuniões, alinhada à cultura organizacional do SmartSuite. Considerou-se as especificidades do ambiente de desenvolvimento e as práticas internas das equipes. O Quadro 4 apresenta as origens dos requisitos funcionais, organizados em duas colunas: identificador, que atribui um número único a cada requisito, e origem, que descreve sua motivação.

Quadro 4 – Origens dos RFs

Identificador	Origem
RF01	Desenvolvedor front-end. Origina-se da necessidade de configurar o serviço de hospedagem de repositórios Git com webhooks, permitindo o envio de informações de revisão para um sistema externo responsável pela notificação de eventos.
RF02	Desenvolvedor front-end. Surge da necessidade de centralizar e automatizar a comunicação durante o processo de revisão de código, visando maior eficiência e fluidez na troca de informações.
RF03	Coordenador. Resulta de uma percepção interna das equipes do SmartSuite, cada uma com suas políticas e etapas de revisão de código específicas, destacando a importância de se adequar às abordagens de cada equipe para garantir um processo de revisão eficiente.
RF04	Desenvolvedor front-end. Surge da necessidade de mapear os usuários da ferramenta de controle de versão com os usuários da ferramenta de comunicação, para garantir maior precisão nas notificações e assegurar que as informações sejam enviadas às pessoas corretas no momento certo.
RF05	Desenvolvedor front-end. Originou-se da necessidade de automatizar a comunicação no processo de revisão de código, visando reduzir a carga manual e melhorar a agilidade da comunicação entre os envolvidos.
RF06	Desenvolvedor front-end. Baseia-se em uma prática interna do SmartSuite, que estabelece um prazo de 24 horas para a conclusão da revisão, para evitar que um desenvolvedor fique impedido de avançar e comprometa entregas.
RF07	Desenvolvedor front-end. Surge da necessidade de monitorar as revisões que ultrapassam o prazo de 24 horas, sendo uma consequência direta do requisito anterior, para evitar impactos negativos no andamento do projeto.

Identificador	Origem
RF08	Desenvolvedor front-end. Origina-se da comunicação interna no SmartSuite, que é baseada na ferramenta Slack, visando melhorar a interação entre as equipes durante o processo de revisão de código.
RF09	Desenvolvedor front-end. Surge da necessidade de estabelecer um mapeamento padrão de revisão de código, fundamentado nas práticas adotadas pela equipe de front-end responsável pelo produto SmartRetail, para garantir consistência nas revisões e facilitar a integração entre as equipes.
RF10	Desenvolvedor front-end. Resulta da necessidade de rastrear eventos durante o processo de revisão de código, permitindo uma análise detalhada das atividades realizadas e a identificação de pontos de melhoria para aprimorar o processo de revisão.
RF11	Squad leader. Surge como consequência do RF05, com o objetivo de intensificar o mecanismo de aviso, complementando a automação da comunicação no processo de revisão de código.

Fonte: Elaborado pelo autor (2025).

Além dos requisitos funcionais, os requisitos de qualidade (também conhecidos como requisitos não funcionais) desempenham um papel essencial na construção do *software*, uma vez que definem restrições e características de qualidade que o mesmo deve atender. A citar, aspectos como desempenho, usabilidade, segurança e confiabilidade. Isso, por muitas vezes, influencia em questões arquitetônicas mais do que requisitos funcionais (POHL; RUPP, 2012).

Os requisitos não funcionais da solução em circunstância são especificados no Quadro 5. Esse quadro é composto por três colunas: identificador, descrição e categoria. O identificador segue o padrão “RNF” seguido do número do requisito de qualidade, a descrição adota a estrutura proposta por Pohl e Rupp (2012), e a categoria indica o grupo ao qual o requisito de qualidade pertence.

Quadro 5 – Requisitos de Qualidade

Identificador	Descrição	Categoria
RNF01	O sistema deverá obedecer a Lei Geral de Proteção de Dados - Brasil (LGPD).	Conformidade
RNF02	O sistema deverá garantir que apenas usuários autenticados possam acessar as funcionalidades.	Segurança
RNF03	O sistema deverá requerer autenticação simples para acessar a interface de gerenciamento.	Segurança
RNF04	O sistema deveria permitir uso de autenticação através de token.	Segurança
RNF05	O sistema deverá processar atualização de revisão de código em duas etapas, persistência e comunicação, considerando a atomicidade.	Confiabilidade
RNF06	O sistema deverá operar 24/7.	Disponibilidade

Identificador	Descrição	Categoria
RNF07	O sistema deverá processar uma atualização de solicitação de revisão de código em até 1 segundo.	Desempenho
RNF08	O sistema deverá ter simplicidade na identificação e correção de falhas.	Manutenibilidade
RNF09	O sistema deverá utilizar mecanismo de exclusão lógica de dados.	Manutenibilidade
RNF10	O sistema deverá funcionar em diferentes sistemas operacionais e dispositivos.	Portabilidade
RNF11	O sistema deverá funcionar com virtualização Docker.	Portabilidade
RNF12	O sistema deverá possuir Interfaces com padrões uniformes.	Usabilidade
RNF13	O sistema deverá possuir padrões de mensagens relacionados ao processo de revisão de código.	Usabilidade
RNF14	O sistema deverá funcionar com padrão de API REST.	Interoperabilidade

Fonte: Elaborado pelo autor (2024).

O estabelecimento dos quatorze requisitos de qualidade fundamentou-se na análise das expectativas e demandas dos envolvidos, levando em conta a cultura organizacional do SmartSuite, assim como em boas práticas de engenharia de software. O Quadro 6 detalha as origens dos requisitos não funcionais, estruturados em duas colunas: identificador, que designa um número único a cada requisito, e origem, que explica sua fundamentação.

Quadro 6 – Origens dos RNFs

Identificador	Origem
RNF01	Desenvolvedor front-end. Exigência de legalidade brasileira para atuar com tratamento de dados.
RNF02	Profissional DevSecOps. Prática de segurança recomendada para proteger o acesso ao sistema.
RNF03	Profissional DevSecOps. Exigência operacional de controle de acesso simplificado para administradores.
RNF04	Profissional DevSecOps. Flexibilidade nos métodos de autenticação para efetuar requisições.
RNF05	Desenvolvedor front-end. Garantir consistência no processamento de revisões.
RNF06	Desenvolvedor front-end. Expectativa de operação ininterrupta esperada pelos desenvolvedores.
RNF07	Desenvolvedor front-end. Expectativa de desempenho para processar um evento de revisão.
RNF08	Profissional DevSecOps. Facilidade no diagnóstico e solução de problemas pelos administradores.
RNF09	Profissional DevSecOps. Preservação de dados, boas práticas de gestão de informações.

Identificador	Origem
RNF10	Profissional DevSecOps. Garantir acessibilidade em diversos ambientes operacionais.
RNF11	Profissional DevSecOps. Reduzir recursos em virtualização.
RNF12	Desenvolvedor front-end. Melhorar a experiência do usuário.
RNF13	Desenvolvedor front-end. Obter consistência na comunicação de eventos.
RNF14	Desenvolvedor front-end. Melhorar a interoperabilidade com outros sistemas.

Fonte: Elaborado pelo autor (2025).

Portanto, o sistema proposto busca realizar uma automação do processo de comunicação de revisão de código, permitindo o administrador configurar o sistema de gerenciamento de controle de versão (RF1), o sistema de comunicação (RF2) e as etapas do processo de revisão de código moderna (RF03). Com isso, os envolvidos na revisão de código, autor e revisor, poderão visualizar os acontecimentos do processo (RF05). Além disso, o mesmo contempla inspeções periódicas (RF07) para verificar solicitações de revisões expiradas (RF06).

Concluída a fase de elicitação, os requisitos funcionais e de qualidade identificados fornecem uma base sólida para a idealização da solução. Agora, a proposta avança para a etapa de análise e *design*, na qual essas especificações serão refinadas e convertidas em uma arquitetura que guiará a construção do sistema, garantindo alinhamento com as expectativas dos *stakeholders*.

4.1.2. Etapa II: Análise e Design de Projeto

Na etapa de análise e design de projeto, conexão entre a especificação de requisitos e implementação, busca-se compreender e organizar as necessidades dos envolvidos, além de propor soluções viáveis para estruturar a solução fundamentada em aspectos de qualidade desejados. Nesse sentido, os próximos tópicos abordarão, de forma detalhada, as visões de banco de dados e arquitetura do sistema.

4.1.2.1.Banco de Dados

A estruturação da proposta de solução baseia-se inicialmente na construção de uma abstração de dados, representado pelo modelo entidade-relacionamento

estendido (*Extended Entity-Relationship*, EER), que fornece uma visão conceitual ao mapear entidades e seus relacionamentos. Em seguida, esse modelo é transformado no modelo relacional, no qual os dados são organizados em tabelas com relações. Por fim, para garantir a integridade e a consistência dos dados, aplica-se o processo de normalização.

Para a construção do ERR, foi realizada uma análise dos requisitos funcionais e de qualidade previamente definidos. A primeira decisão envolveu a concepção de uma entidade genérica, denominada **GENERIC_MODEL**, da qual diversas entidades herdam. Essa entidade contém atributos padrões, como identificador único (RNF07), data de criação (RF10), data de atualização (RF10) e flag de exclusão lógica (RNF09).

O identificador único garante a unicidade dos registros, facilitando consultas rápidas (escalabilidade) e a criação de relacionamentos. A data de criação permite rastrear quando o registro foi inserido, enquanto a data de atualização registra a última modificação, viabilizando auditorias e ordenações temporais. Por fim, a flag de exclusão lógica possibilita a desativação de registros sem removê-los do banco de dados. Isso preserva o histórico e facilita a recuperação dos dados.

Na sequência, observou-se as características envolvidas no processo de revisão de código (RF05 e RF06). Nesse sentido, houve a criação da entidade para representar a revisão, chamada **REQUEST**, que contempla os informações de identificador da solicitação de mesclagem (*RequestId*), identificador da *thread* de comunicação (*MessageId*), autor da solicitação (*Author*) e status da fase corrente no processo de revisão de código (*Status*).

Para permitir a construção do histórico de eventos que ocorreram no processo de revisão de código (RF10), optou-se por criar uma entidade intitulada **HISTORICAL_REQUEST** cujo atributos incluem campo composto da solicitação (*Request*, uma cópia em dado momento das informações de REQUEST), identificador (*HistoryId*), usuário (*HistoryUser*), data (*HistoryDate*) e motivo da alteração (*HistoryChangeReason*) do item de histórico.

Para estabelecer um padrão de mensagem para cada evento no processo de revisão de código (RNF13) e relacionar os usuários envolvidos (RF11), decidiu-se criar a entidade **USER**. Essa entidade mapeia os usuários do sistema de controle de versão para o sistema de comunicação, com dois atributos principais: *GitUsername*,

que representa o nome de usuário no Git, e *ChatUsername*, correspondente ao nome de usuário no sistema de comunicação.

Com foco em proporcionar uma solução genérica suficiente para comunicar eventos de quaisquer processos de revisão de código (RF03), estabeleceu-se uma entidade **SETTING** com atributos de nome da configuração (*Name*), token de acesso (*AccessToken*) — hash para identificar aquela configuração —, endereço de projeto (*ProjectURL*), canal de comunicação (*Channel*) e estado de ativação da configuração (*Enable*).

Cada configuração possui uma associação com **DATA_PATH**, entidade com propósito de gerir informações sobre a localização de dados dispostos em formato JSON, como localização do identificador da solicitação (*JPRequestID*), autor (*JPAuthor*), título da solicitação (*JPTitle*) e revisores (*JPReviewers*).

Além disso, pressupõe-se que toda configuração contempla um grupo de estágios específicos, a depender do processo de revisão de código adotado. Desse modo, foi criada a entidade **STAGE** na qual possui o atributo de status do estágio (*Status*) e relaciona-se com um conjunto de regras. Para representar a regra, houve a concepção de **RULE**, entidade formada pelos campos de tipo de condição (*TypeCondition*), valor esperado (*Value*), localização do dado (*JsonPath*) e negar condição (*DenyCondition*).

A Figura 3 ilustra a abstração de dados idealizada, destacando as entidades GENERIC_MODEL, REQUEST, HISTORICAL_REQUEST, USER, SETTING, DATA_PATH, STAGE e RULE. A figura também apresenta os relacionamentos entre essas entidades, assim como as respectivas cardinalidades.

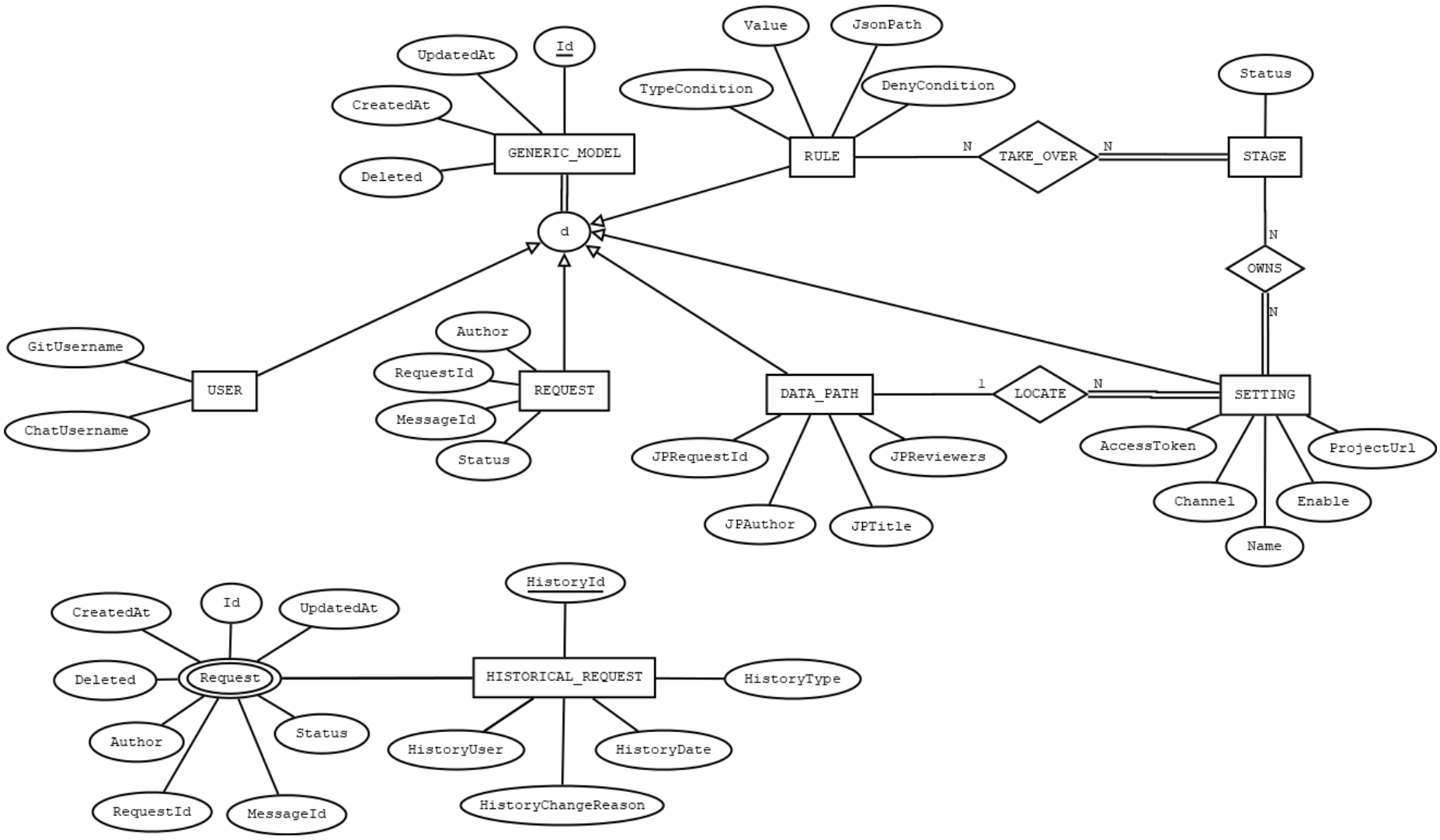
Com isso, foi criado o modelo relacional, utilizando um conjunto de regras de conversão, como transformação de entidades em tabelas, atributos (composto ou simples) em colunas, atributos identificadores em chaves primárias e inclusão de relacionamentos (tabelas intermediárias, chave estrangeira). Além disso, os nomes no formato CamelCase (exemplo: DataPath) foram convertidos para o formato SnakeCase (exemplo: data_path). O resultado pode ser visualizado na Figura 4.

Por fim, aplicou-se o processo de normalização, verificando a Primeira (1FN), Segunda (2FN) e Terceira Forma Normal (3FN), além da Forma de Normal de Boyce-Codd. Esse processo garantiu que a solução proposta elimina,

respectivamente, valores repetitivos e multivalorados, dependências parciais da chave primária e dependências transitivas.

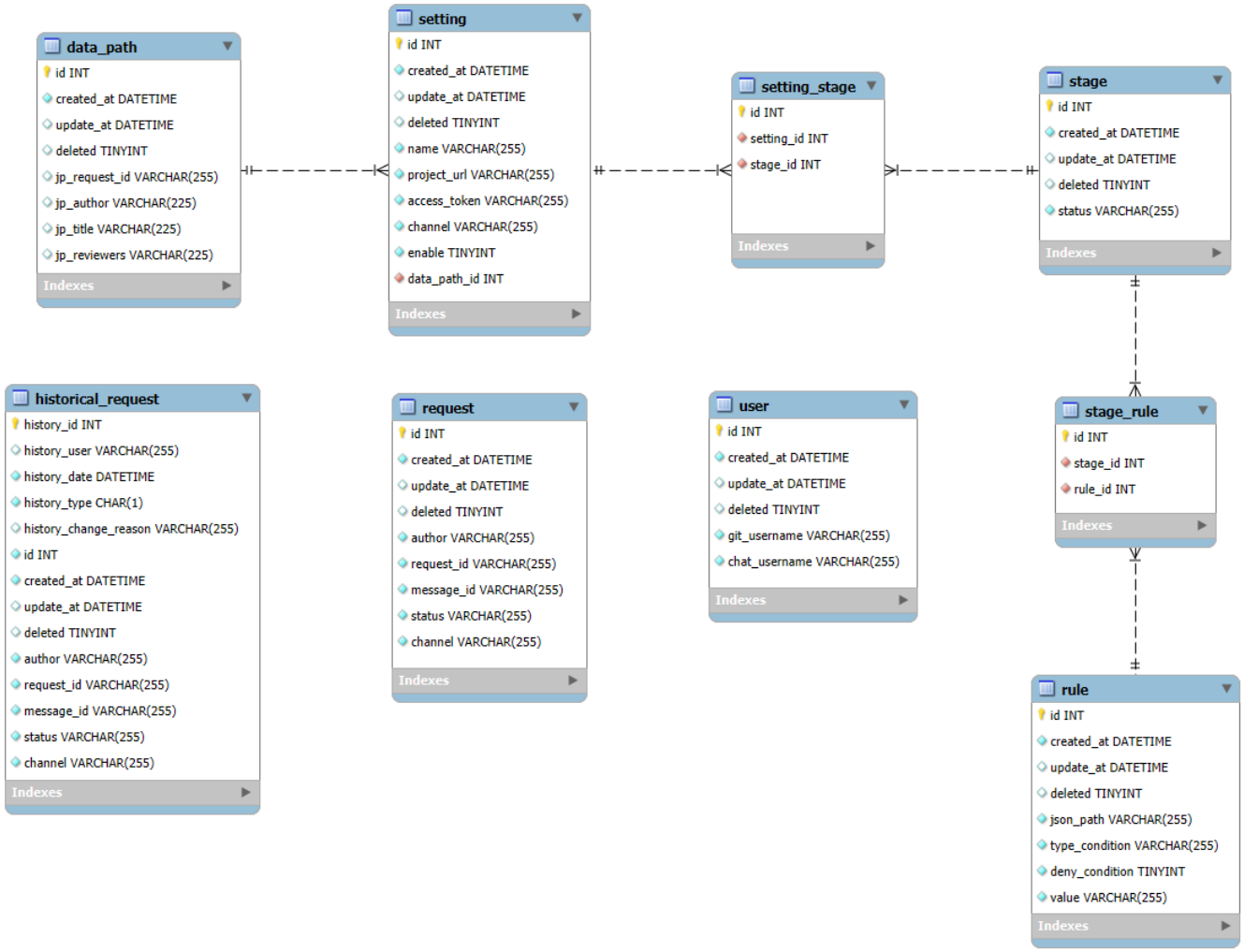
Para isso, utilizou-se a ferramenta Normalization Tool da *Griffith University*. Os resultados encontram-se no Apêndice A. Nele, cada tabela exibe-se a verificação de 2NF, 3NF e BCNF, com identificação de possíveis chaves candidatas para serem primárias e análise de dependências funcionais.

Figura 3 – Modelo entidade-relacionamento estendido (Smart Review)



Fonte: Elaborado pelo autor (2025).

Figura 4 – Modelo relacional (Smart Review)



Fonte: Elaborado pelo autor (2025).

4.1.2.2.Arquitetura

A arquitetura de um sistema representa os conceitos fundamentais que definem sua estrutura, abrangendo elementos, relacionamentos e os princípios que guiam seu design e evolução (ISO/IEC/IEEE 42010:2022). Nesta seção, será apresentada a arquitetura da solução proposta para a automação do processo de comunicação sobre revisão de código, com foco em como suas características estruturais e comportamentais foram planejadas para atender às necessidades dos *stakeholders*.

A primeira decisão arquitetural baseia-se nos protocolos de comunicação. A solução proposta visa integrar dois sistemas: a plataforma de gerenciamento de repositórios Git (exemplo: GitLab) e a plataforma de comunicação (exemplo: Slack), ambos com suporte, no mínimo, para *webhook* e *chatbot*, respectivamente.

Webhooks possibilitam o envio automático de informações para um serviço externo após a ocorrência de um evento específico, como modificações em uma solicitação de mesclagem (*merge request*, *pull request*). No contexto de plataformas de gerenciamento de repositórios Git, utilizam-se callbacks HTTP personalizados, os quais enviam dados em formato JSON sobre eventos para um endereço previamente configurado.

Chatbots no contexto da comunicação viabilizam a automação de interações, como a publicação de mensagens, curtidas em postagens e até respostas a comentários. Ferramentas de comunicação, como Slack, oferecem espaços para o desenvolvimento de aplicações automatizadas, disponibilizando APIs que facilitam a integração e personalização.

Assim, a solução proposta deve fornecer uma API para receber informações no formato JSON da plataforma de gerenciamento de repositórios Git, bem como disponibilizar recursos para permitir configurações de processo de revisão de código. Além disso, deve ser capaz de se comunicar com a API da ferramenta de comunicação. Nesse sentido, deve-se utilizar o estilo de arquitetura cliente-servidor.

Conforme exposto na seção anterior, a solução proposta exige o controle de informações de forma estruturada, o que demanda a adoção de um banco de dados relacional. Assim, a aplicação deve se comunicar com um Sistema Gerenciador de

Banco de Dados, como PostgreSQL ou MySQL, utilizando o protocolo TCP/IP para assegurar uma troca eficiente de informações.

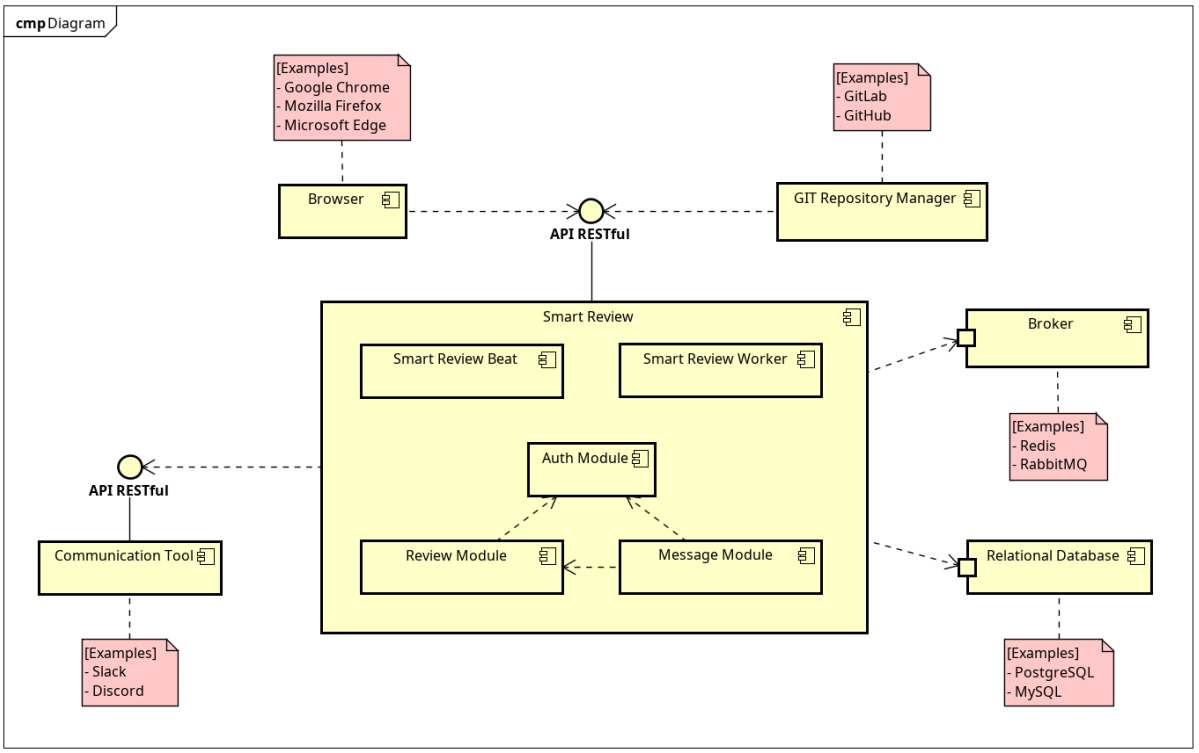
Para atender aos requisitos RF06 e RF07, que envolvem o controle de solicitações de revisão expiradas e verificações a cada uma hora, adotou-se o estilo de arquitetura broker para permitir o desacoplamento (componentes), escalabilidade (notificações) e flexibilidade (integração com diferentes tecnologias). Nesse modelo, os módulos internos da solução, **Smart Review Beat** e **Smart Review Worker**, atuam como produtor e consumidor, respectivamente.

Na visão interna da solução proposta, além dos módulos já mencionados, propõe-se a criação de um módulo central, denominado **Review Module**, responsável por estruturar a aplicação ao definir os modelos e fornecer uma interface para configurações, como mapeamento de usuários, processos de revisão de código e solicitações de revisão. Adicionalmente, o Review Module disponibiliza ao **Message Module** os recursos essenciais (interfaces, modelos, funções) para gerenciar eventos relacionados à revisão de código.

Para garantir os requisitos associados à categoria de segurança, RNF02, RNF03 e RNF02, decidiu-se utilizar um módulo interno de autenticação e autorização, chamado de **Auth Module**. Ele fornece para os demais módulos, Review Module e Message Module, funções para autenticar usuários e gerenciar permissões.

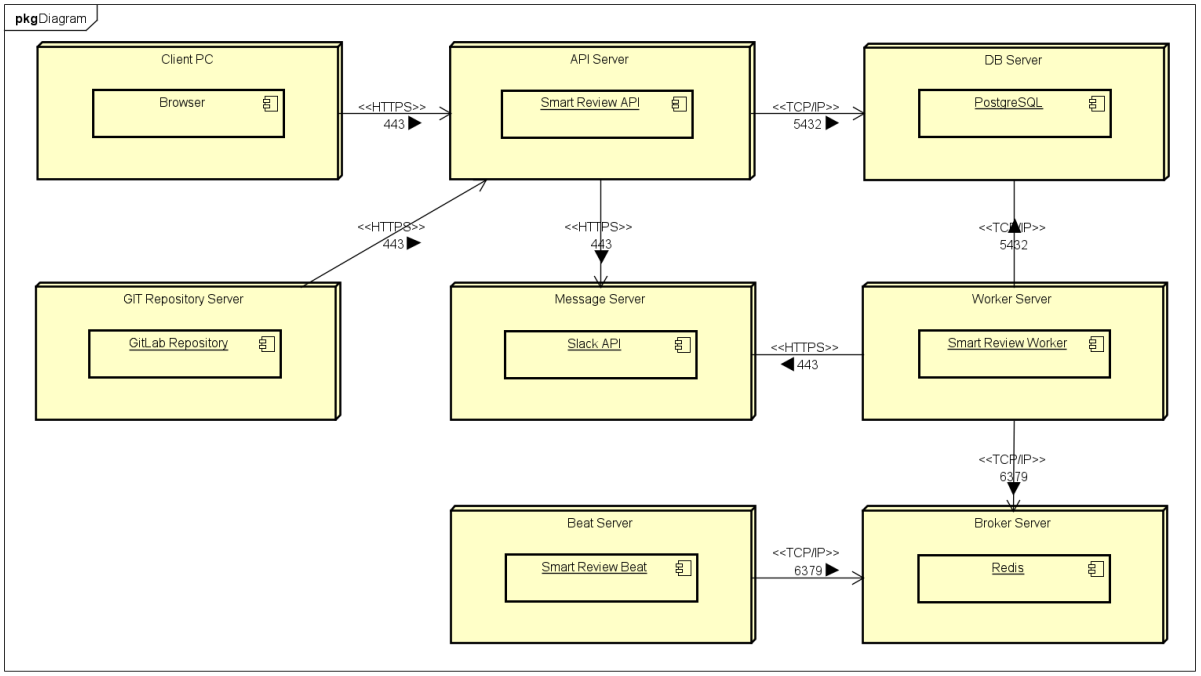
A Figura 5 ilustra, por meio do diagrama de componentes, um panorama geral da solução proposta, destacando os componentes envolvidos (internos e externos) e suas comunicações. A Figura 6, por sua vez, apresenta uma possível instanciação da solução, utilizando o GitLab, Slack, Redis e PostgreSQL como ferramentas de gerenciamento de repositórios Git, comunicação, broker e banco de dados relacional, respectivamente.

Figura 5 – Visão da arquitetura



Fonte: Elaborado pelo autor (2024).

Figura 6 – Implantação com GitLab, Slack, Redis e PostgreSQL



Fonte: Elaborado pelo autor (2024).

Para a estruturação do Review Module, optou-se por adotar uma arquitetura em camadas, com o objetivo de separar responsabilidades e organizar as funcionalidades de forma eficiente. A arquitetura abrange os níveis de apresentação (*View*), transformação (*Serializer*) e dados (*Model*). Visando a reutilização de código, cada nível é implementado com uma estrutura genérica que encapsula as propriedades e métodos comuns.

A Figura 7 ilustra, por meio de um diagrama de classes, a organização das camadas do sistema. O diagrama evidencia os relacionamentos entre as classes, as multiplicidades associadas e outros elementos relevantes. Dessa forma, proporciona uma visão detalhada e estruturada das interconexões entre os componentes.

Assim como no Review Module, optou-se por adotar o estilo arquitetural em camadas no Message Module, com o objetivo de manter uma padronização estrutural. Dessa forma, são definidos os níveis de visualização, serviço e dados. A camada de dados do Message Module é um subconjunto da camada de dados do Review Module. A Figura 8 ilustra essa organização.

Uma vez definida a estrutura dos módulos, chega-se o momento de definir as características comportamentais do sistema. A começar pela funcionalidade de publicar uma atualização de revisão de código com propõe-se atender o requisito RF05, a fim de aumentar a eficiência do processo e padronização de comunicação.

Definiu-se que essa funcionalidade envolve três componentes principais: a ferramenta de gerenciamento de repositórios Git, o Smart Review e a ferramenta de comunicação. Quando uma solicitação de mesclagem é criada, o gerenciador de repositórios envia os dados do evento ocorrido para o Smart Review.

Com base nesses dados, o Smart Review busca a configuração do projeto referência e avalia se o evento atende ao conjunto de regras de um estágio específico. Caso positivo, ele prepara uma mensagem mencionando os envolvidos, cria ou atualiza a solicitação de revisão, e solicita à ferramenta de comunicação a publicação da mensagem. Caso contrário, encerra o processamento.

A Figura 10 expõe, mediante ao diagrama de atividades, o fluxo descrito anteriormente. Nela, observa-se três partições: GIT Repository, Smart Review e Communication Tool, cada qual com um conjunto de atividades. Complementando, a Figura 9 exibe uma visão particular da solução para atender ao comportamento em evidência.

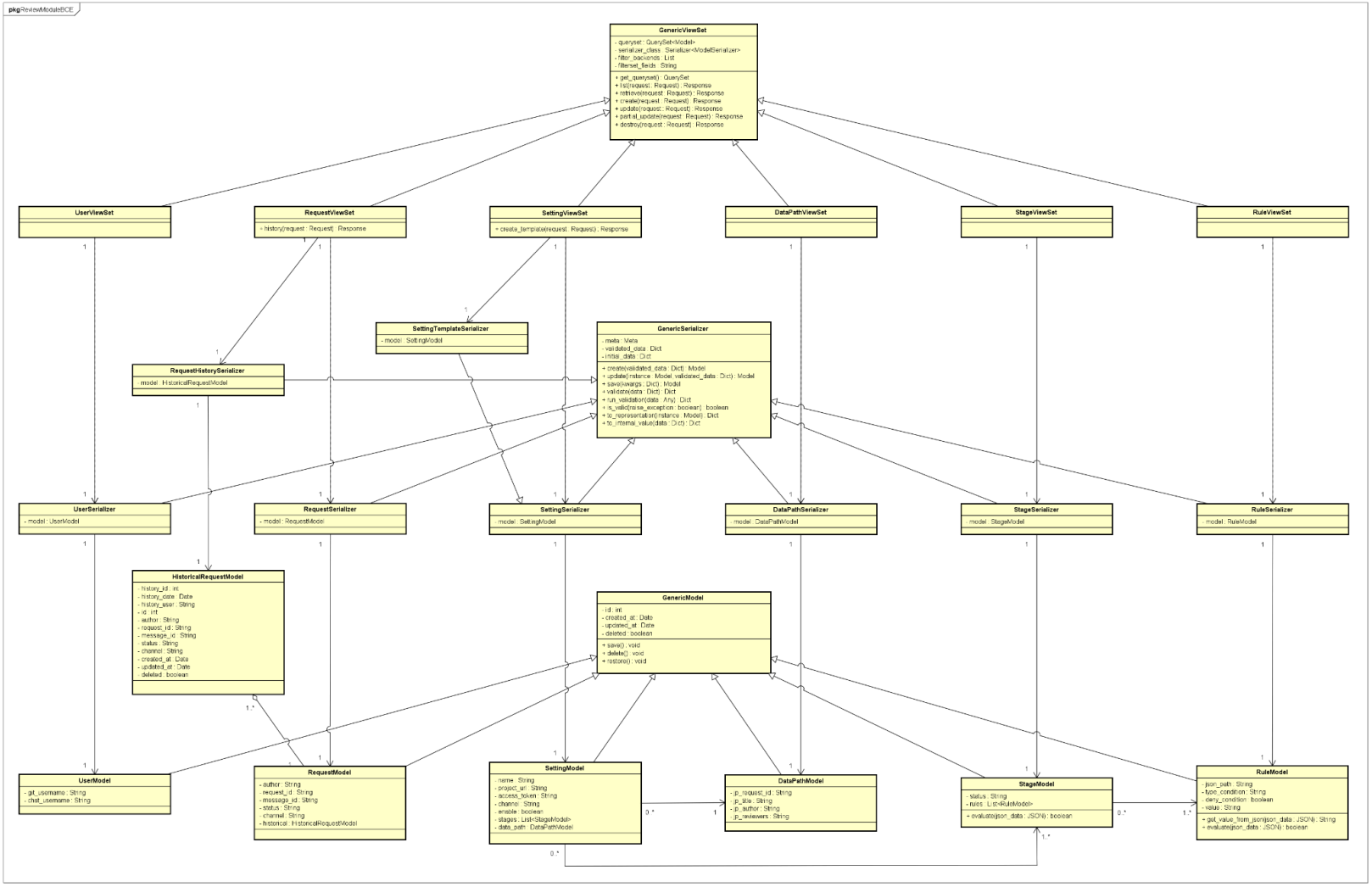
Outra funcionalidade importante é o controle da expiração das solicitações de revisão de código, com o objetivo de atender aos requisitos RF06 e RF07. Essa funcionalidade visa evitar solicitações de revisão com longa duração, que podem prejudicar o desempenho da equipe, além de reduzir comunicações pessoais que, de certa forma, comprometem o envolvimento entre autor e revisor.

Em termos comportamentais, definiu-se que essa funcionalidade envolve a participação de diversos componentes: Smart Review Beat, Smart Review Worker, Broker e Communication Tool. O fluxo inicia-se em intervalos predefinidos, como, por exemplo, a cada uma hora.

Nesse intervalo, o Smart Review Beat adiciona ao Broker a rotina para verificar a existência de solicitações expiradas, com duração de 24 horas e sem conclusão. Após isso, o Smart Review Worker é acionado para analisar as solicitações expiradas. Caso algum item seja encontrado, ele é atualizado e o sistema solicita à ferramenta de comunicação que publique uma mensagem referente à expiração.

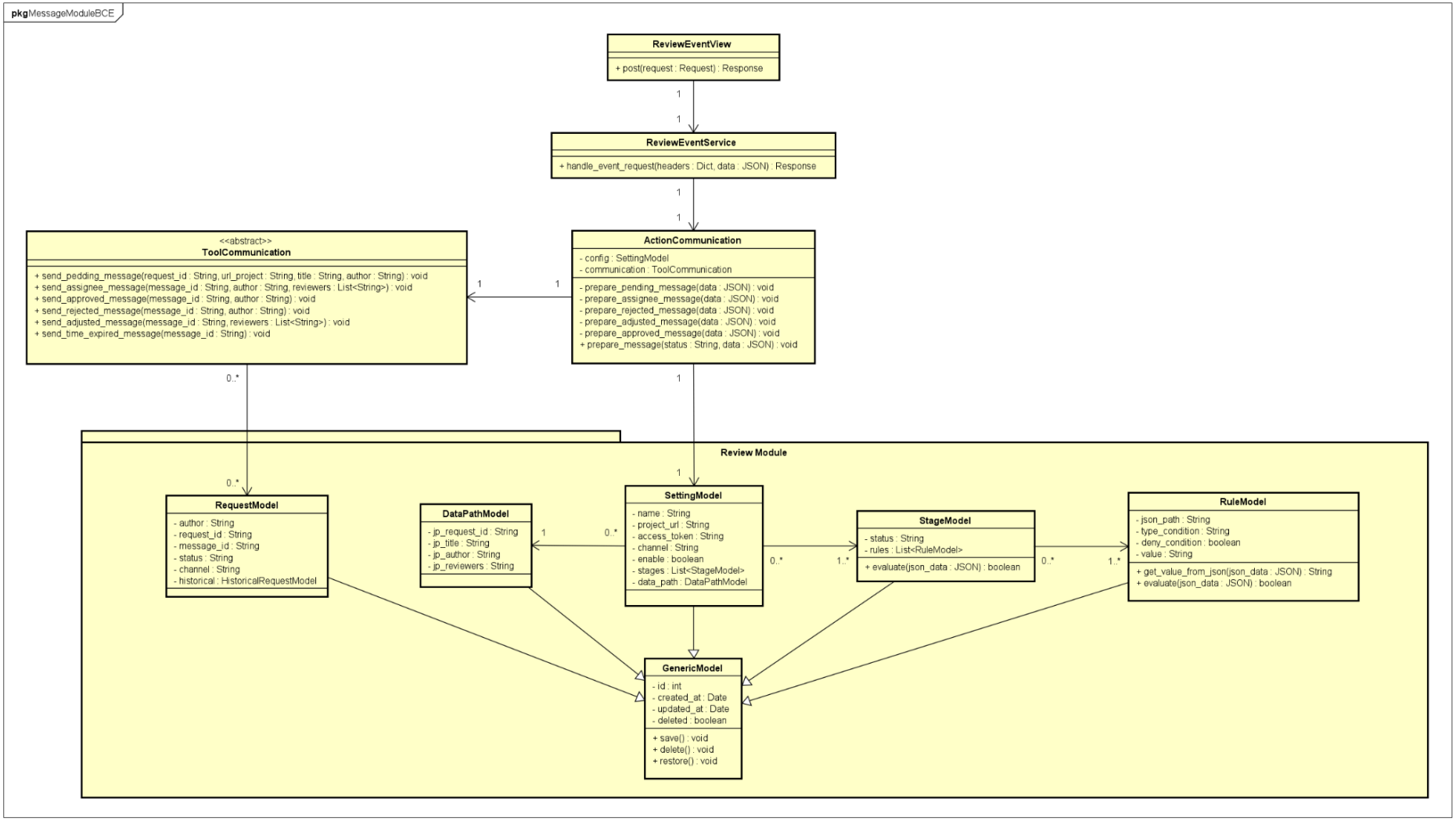
A Figura 11 ilustra, por meio do diagrama de atividades, esse conjunto de ações. O diagrama está organizado em quatro partições: Smart Review Beat, Smart Review Worker, Broker e Communication Tool, sendo que cada partição contém suas respectivas atividades.

Figura 7 – Estrutura de Review Module



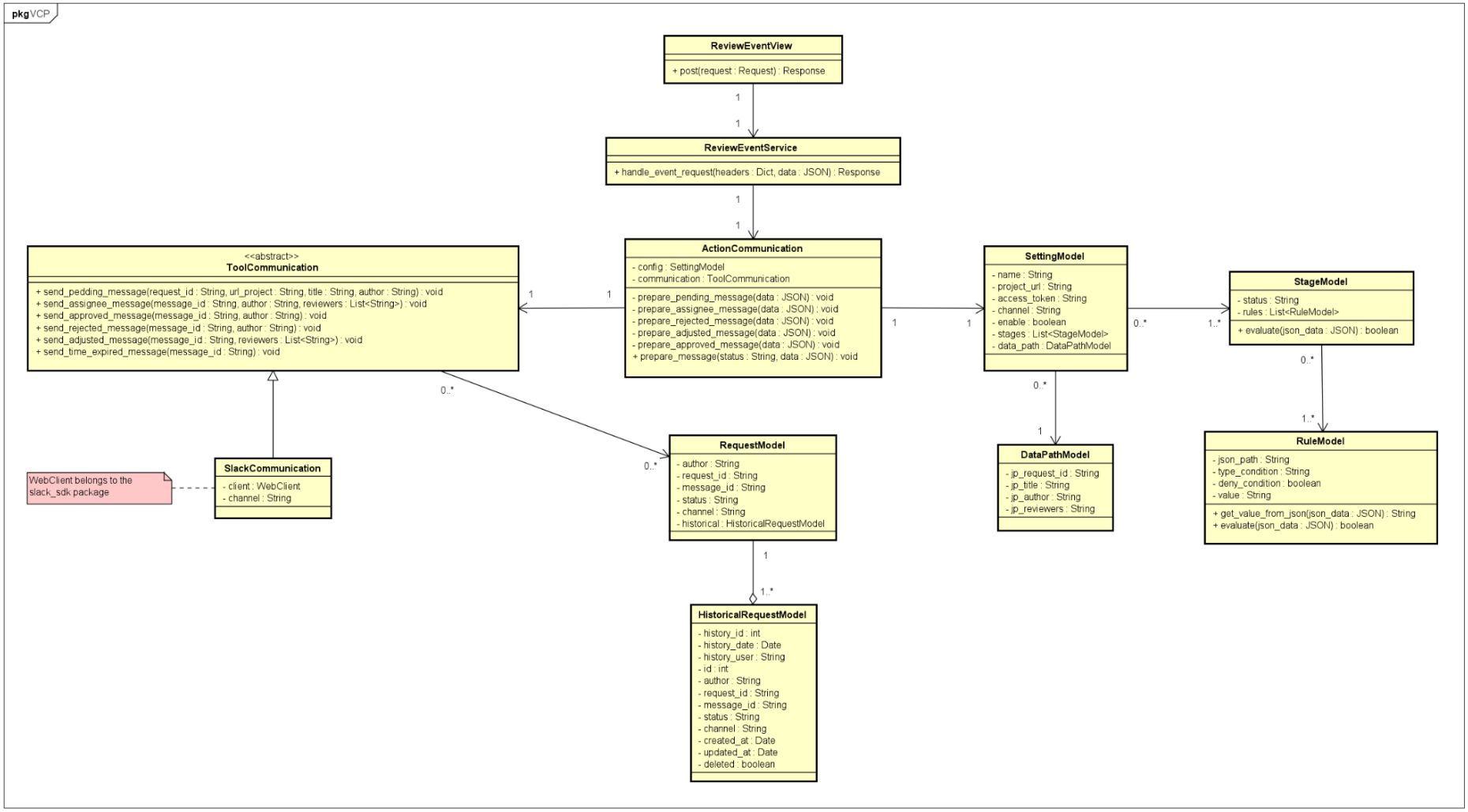
Fonte: Elaborado pelo autor (2024).

Figura 8 – Estrutura de Message Module



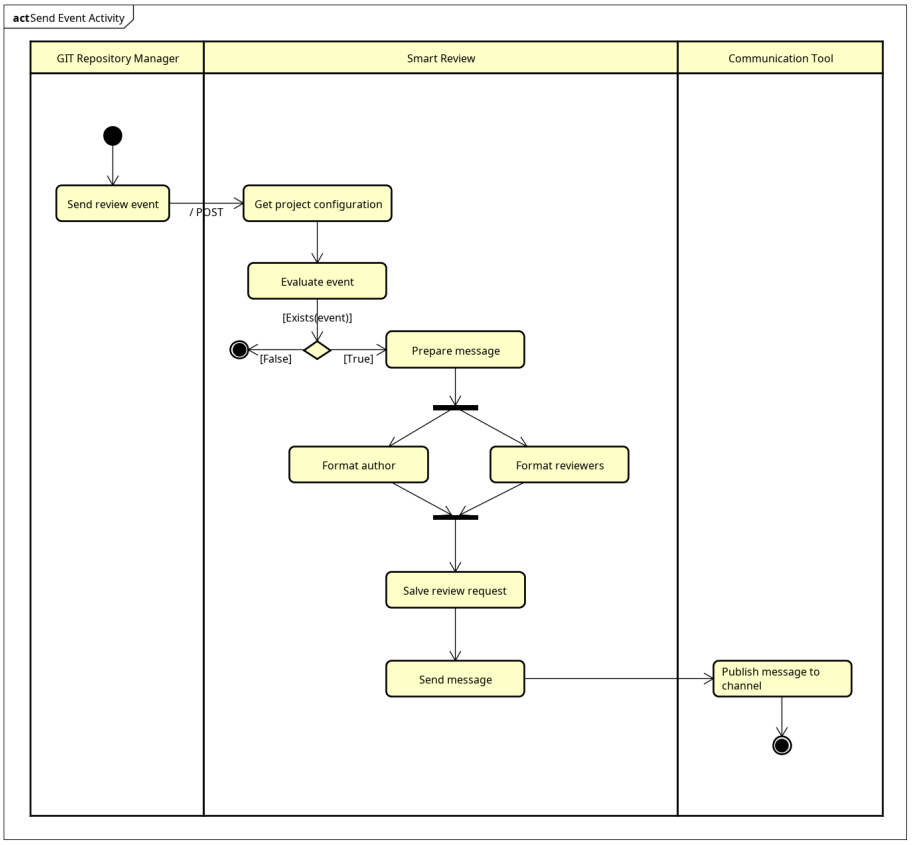
Fonte: Elaborado pelo autor (2024).

Figura 9 – Estrutura para processar eventos de revisão de código



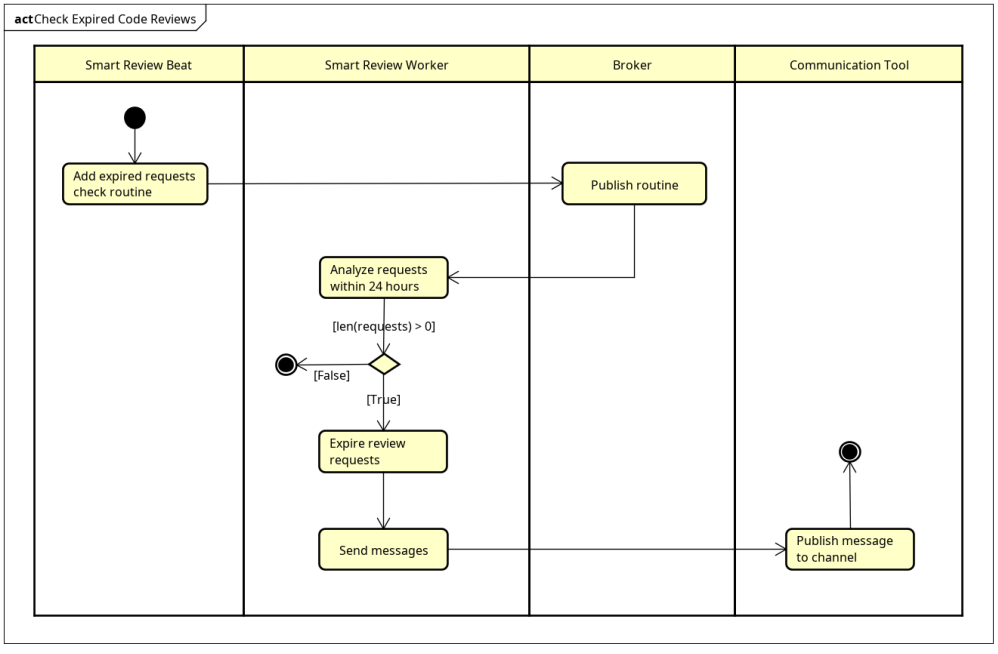
Fonte: Elaborado pelo autor (2024).

Figura 10 – Comportamento de enviar evento de revisão de código



Fonte: Elaborado pelo autor (2024).

Figura 11 – Comportamento de verificar solicitações de revisão expiradas



Fonte: Elaborado pelo autor (2024).

4.1.3. Etapa III: Implementação

Na etapa de implementação, as concepções de estrutura de dados e arquitetura de software são consolidadas por meio de ferramentas, tecnologias e padrões. Essa fase envolve a tradução dos requisitos em soluções práticas, assegurando a integração e o funcionamento adequado de todos os componentes.

De acordo com a arquitetura idealizada, propõe-se o desenvolvimento de uma API para representar o Smart Review, utilizando o *framework* Django, devido aos seguintes aspectos:

- **Desenvolvimento rápido**: Django segue o princípio 'não se repita', oferecendo uma estrutura robusta para criar aplicações ágeis e eficientes;
- **Segurança**: Django foi projetado para segurança, protegendo contra ataques como *SQL Injection*, *XSS*, *CSRF* e *Clickjacking*;
- **Escalabilidade**: Django facilita a construção de aplicações escaláveis, graças à sua arquitetura modular;
- **Administração**: Django inclui um painel de administração automático, que simplifica a gestão de dados;
- **Documentação**: Django oferece uma documentação detalhada, que facilita tanto o aprendizado quanto a implementação de novas funcionalidades;
- **Extensibilidade**: Django permite a adição de módulos específicos de forma simples e eficiente.

A ferramenta de gerenciamento de repositórios Git escolhida, por sua vez, foi GitLab, uma vez que possui suporte nativo para *webhook* (condição primordial) e oferece um série de benefícios para o contexto em circunstâncias, a saber:

- **Colaboração e Gestão de Projetos**: Colaboração é facilitada com revisão de código, comentários e sugestões em *merge requests*;
- **Integração e Compatibilidade**: Integração com ferramentas e serviços externos como Kubernetes, JIRA, Slack e outras plataformas DevOps.

No tocante da ferramenta de comunicação, preferiu-se o Slack, em virtude de disponibilizar *chatbot* (condição essencial) e proporcionar diversas vantagens, como:

- **Facilidade de Uso**: Com uma interface intuitiva e moderna, o Slack é fácil de usar, mesmo para equipes que não possuem experiência com ferramentas de colaboração;
- **Comunicação Assíncrona**: Além da comunicação em tempo real, o Slack suporta mensagens assíncronas, o que é ideal para equipes distribuídas ou que trabalham em diferentes fusos horários;
- **Organização por Canais**: Permite criar canais públicos ou privados para diferentes assuntos, garantindo que as conversas sejam mantidas organizadas e relevantes;
- **Automatização com Slack Bots e Workflows**: Slack permite criar automações simples para tarefas repetitivas por meio de bots e workflows, como lembretes, notificações automáticas e respostas rápidas.

Para SGDB optou-se pelo PostgreSQL devido ser gratuito, possuir uma estabilidade no mercado (mais de trinta anos em desenvolvimento), conformidade com padrões SQL, desempenho, escalabilidade, segurança e facilidade na integração com linguagens de programação.

A tecnologia de broker escolhida foi Redis, pelos seguintes motivos: alto desempenho e baixa latência, estrutura em memória, possui um sistema de publicação e assinatura nativo, versatilidade no suporte de estrutura de dados (listas, conjuntos, hashes), escalabilidade e simplicidade na configuração.

A primeira etapa no desenvolvimento do projeto consistiu em definir os módulos específicos que permitiriam a construção de uma API de forma rápida e robusta, além da implementação do histórico. Para atender a essas necessidades, optou-se pelos pacotes *Django Rest Framework* e *Django Simple History*, respectivamente.

A próxima definição na construção do projeto foi adotar variáveis de ambiente, em busca de atender ao requisito de RNF10 e RNF11. Para isso, utilizou-se a função de "*getenv"* da biblioteca "*os*" provida nativamente pelo Python. O Quadro 7 fornece um detalhamento das variáveis de ambiente criadas, informando a identificação, descrição e exemplo de uso.

Quadro 7 – Variáveis de Ambiente

Identificador	Descrição	Exemplo
DEBUG	Modo de depuração.	False

Identificador	Descrição	Exemplo
SECRET_KEY	Chave secreta do aplicativo.	django-insecure-pjzbnxclzdjlo)6yf5n^s^#p6ng+1o130=old^@sa3fia9p7zc
SLACK_API_TOKEN	Token para acesso à API do Slack.	xoxb-7862828220625-7977643400353-63aRRh72gBu301dQwpme3HUk
DB_NAME	Nome do banco de dados.	smart-review
DB_USER	Nome de usuário para acessar o SGDB.	admin
DB_PASSWORD	Senha de usuário para acessar o SGDB.	asdf123$
DB_HOST	Domínio do SGDB.	127.0.0.1
DB_PORT	Porta do SGDB.	5432
CELERY_BROKER_URL	Endereço do Redis.	redis://0.0.0.0:6379/0
DOMAIN_URL	Domínio da aplicação.	https://smart-review.br

Fonte: Elaborado pelo autor (2024).

Após realizar as configurações iniciais e comunicações entre as tecnologias, iniciou-se o processo de desenvolvimento pelo módulo central. A primeira camada implementada foi de modelos. Nesse sentido, o modelo relacional produzido na etapa de análise e design de projeto (Figura 4) foi transformado em modelos no contexto de Django ORM. A Figura B.1 (Apêndice B) ilustra o modelo de RequestModel que herda propriedades e métodos de GenericModel, além de categorizar os status (ReviewStatus).

Na sequência, desenvolve-se a camada de transformação de dados. Nesse sentido, foi criado um serializador para cada modelo, herdando atributos e características de um serializador genérico. A Figura B.2 (Apêndice B) contém essa relação ao apresentar RequestSerializer, a classe serializadora do modelo RequestModel.

Em seguida, implementou-se a camada de visualização. Semelhante à camada de transformação, cada classe de visão é responsável por um modelo específico e utiliza o serializador adequado para ele. Por exemplo, a visão associada ao modelo RequestModel é empregada em determinados contextos, como consultas de dados, onde é necessário transformar objetos complexos em formatos simples, como JSON. A Figura B.3 (Apêndice B) mostra a visão de RequestModel.

Finalizando o módulo central, foram desenvolvidas as classes **Tool Communication** e **ActionCommunication**, com o objetivo de subsidiar a construção de aplicações para comunicação com ferramentas de colaboração

externa. A classe ToolCommunication é uma classe abstrata que define métodos para o envio de mensagens, enquanto a ActionCommunication implementa métodos para a preparação dessas mensagens.

Uma vez finalizada a estrutura do módulo central, iniciou-se o desenvolvimento de funcionalidades de criar configuração a partir de padrão de processo de revisão de código e obter histórico de eventos de uma solicitação de revisão, para satisfazer os requisitos RF09 e RF10, respectivamente. Para isso, utilizou-se “@action”, função disponibilizada pelo pacote rest_framework para estender comportamentos de um recurso. As Figuras B.4 e B.5 (Apêndice B) retratam as construções das ações.

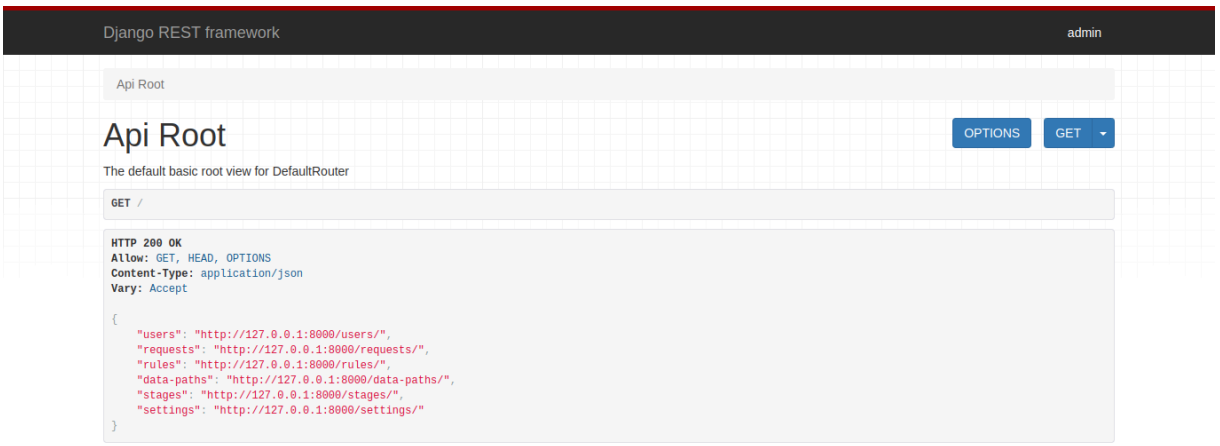
Com isso, começou-se o desenvolvimento do módulo de mensagem com foco em integrar a solução com o Slack, ferramenta de comunicação. Nesse sentido, necessitou-se incorporar o pacote slack_sdk e utilizar a classe WebClient para realizar chamadas à API do Slack. O WebClient permite enviar mensagens, consultar canais e executar outras operações. Após adicionar isso, implementou-se as classes que podem ser observadas na Figura 9, de modo a contemplar a funcionalidade de processar eventos de revisão de código.

Para concluir a implementação da solução proposta, realizou-se uma análise de ferramentas presentes no ecossistema Python para representar o Smart Review Beat e Smart Review Worker. Nesse sentido, encontrou-se o pacote Celery, uma biblioteca com foco no gerenciamento de tarefas assíncronas e agendadas.

Desse modo, criou-se uma função que é executada a cada intervalo de tempo pré-definido, inicialmente de uma hora. Essa captura as solicitações com 24 horas de existência que não possuem status “aprovada” ou “expirada”, ou seja, solicitações não concluídas. Ela, então, expira às solicitações encontradas e publica mensagens nas threads associadas. As Figuras B.6 e B.7 (Apêndice B) ilustram, respectivamente, a configuração e a definição da função.

A Figura 12 mostra a página inicial da API em execução com autenticação realizada (usuário admin). Estão disponíveis os recursos *users*, *requests*, *rules*, *data-paths*, *stages* e *settings*, cada um oferecendo funcionalidades como listagem, cadastro, atualização (parcial e completa), remoção e aplicação de filtros.

Figura 12 – Smart Review API



Fonte: Elaborado pelo autor (2025).

4.2. Limitações

Embora a solução proposta tenha sido projetada para atender aos objetivos estabelecidos e melhorar a eficiência no processo de comunicação de revisão de código, algumas limitações precisam ser reconhecidas. Essas limitações resultam de escolhas técnicas, restrições de recursos e desafios enfrentados durante a concepção, *design* e implementação.

Personalização das mensagens. Embora a automação permita a geração de mensagens de forma padronizada, as mensagens podem carecer de uma personalização dependendo do contexto de uso. Por exemplo, definir o estilo da linguagem (informal, formal) e incluir outras informações (data da operação, avisos).

Restrições de API. Dependendo da ferramenta de controle de repositórios ou de comunicação, a solução pode ser limitada pela taxa de requisições. Por exemplo, o Slack limita as postagens de mensagens de 1 por segundo[5]. Isso significa que se houver duas solicitações de revisões de código criadas no mesmo segundo, uma ficará sem processamento.

[5] https://api.slack.com/apis/rate-limits

Escalabilidade. A solução pode enfrentar limitações à medida que a carga de trabalho e o número de usuários aumentam. Isso ocorre devido à capacidade limitada de processamento de dados e requisições simultâneas, que podem ser impactadas por restrições de taxa das APIs integradas ou pela infraestrutura utilizada. Por exemplo, um grande volume de revisões de código geradas ao mesmo tempo pode resultar em atrasos ou falhas na entrega de notificações.

Tolerância a falhas. A solução não está preparada para se recuperar de falhas no envio de notificações relacionadas aos eventos de revisão de código. Por exemplo, se duas solicitações de revisão de código forem geradas no mesmo segundo para a ferramenta de comunicação Slack, apenas uma será notificada, enquanto a outra será perdida.

5. ESTUDO DE CASO

Neste capítulo, busca-se avaliar a proposta de solução, Smart Review, por meio de um estudo de caso exploratório com abordagem qualitativa, seguindo as diretrizes de Yin (2004) e Wohlin *et al*. (2012). Para isso, serão apresentados o escopo, o planejamento, a operação e a análise dos resultados.

5.1. Escopo

Pretende-se analisar a automação das comunicações durante a revisão de código com o propósito de avaliar a eficiência percebida do processo e o nível de colaboração do ponto de vista dos desenvolvedores, em seus diferentes papéis (autor e revisor), no contexto da equipe front-end referente ao projeto SmartRetail.

Objeto de estudo. O objeto de estudo é a automação das comunicações que ocorrem durante o processo de revisão de código-fonte. Isso inclui plataformas de comunicação e integração com sistemas de controle de versão, considerando a percepção dos envolvidos sobre seu impacto na experiência de trabalho.

Propósito. O propósito do estudo é compreender se a automação das comunicações contribui para a melhoria na eficiência do processo de revisão de código-fonte e para o fortalecimento da colaboração entre os desenvolvedores, a partir da perspectiva dos próprios participantes.

Perspectiva. A perspectiva se concentra nos desenvolvedores, considerando seus papéis como atores que submetem o código para revisão e revisores que avaliam as contribuições dos colegas. Essa abordagem permite entender como a automação impacta tanto a experiência de submeter código quanto a de revisar, analisando a comunicação e a colaboração entre eles.

Foco na Qualidade. O foco na qualidade envolve a análise qualitativa de como os desenvolvedores percebem o impacto da automação no processo de revisão de código, incluindo sua experiência geral, nível de colaboração e satisfação com as práticas atuais.

Contexto. O estudo será aplicado à equipe de desenvolvedores com atuação em front-end, do projeto SmartRetail – sistema para controle de vendas de produtos. A equipe é composta por cinco desenvolvedores com diferentes níveis de

experiência. A cultura organizacional do projeto prioriza a colaboração e o *feedback* contínuo, criando um ambiente de trabalho aberto e estimulante. As ferramentas utilizadas incluem o Git para controle de versão, gerenciado através do GitLab, e o Slack para comunicação, ambos elementos essenciais no fluxo de trabalho colaborativo e nos processos analisados.

5.2. Planejamento

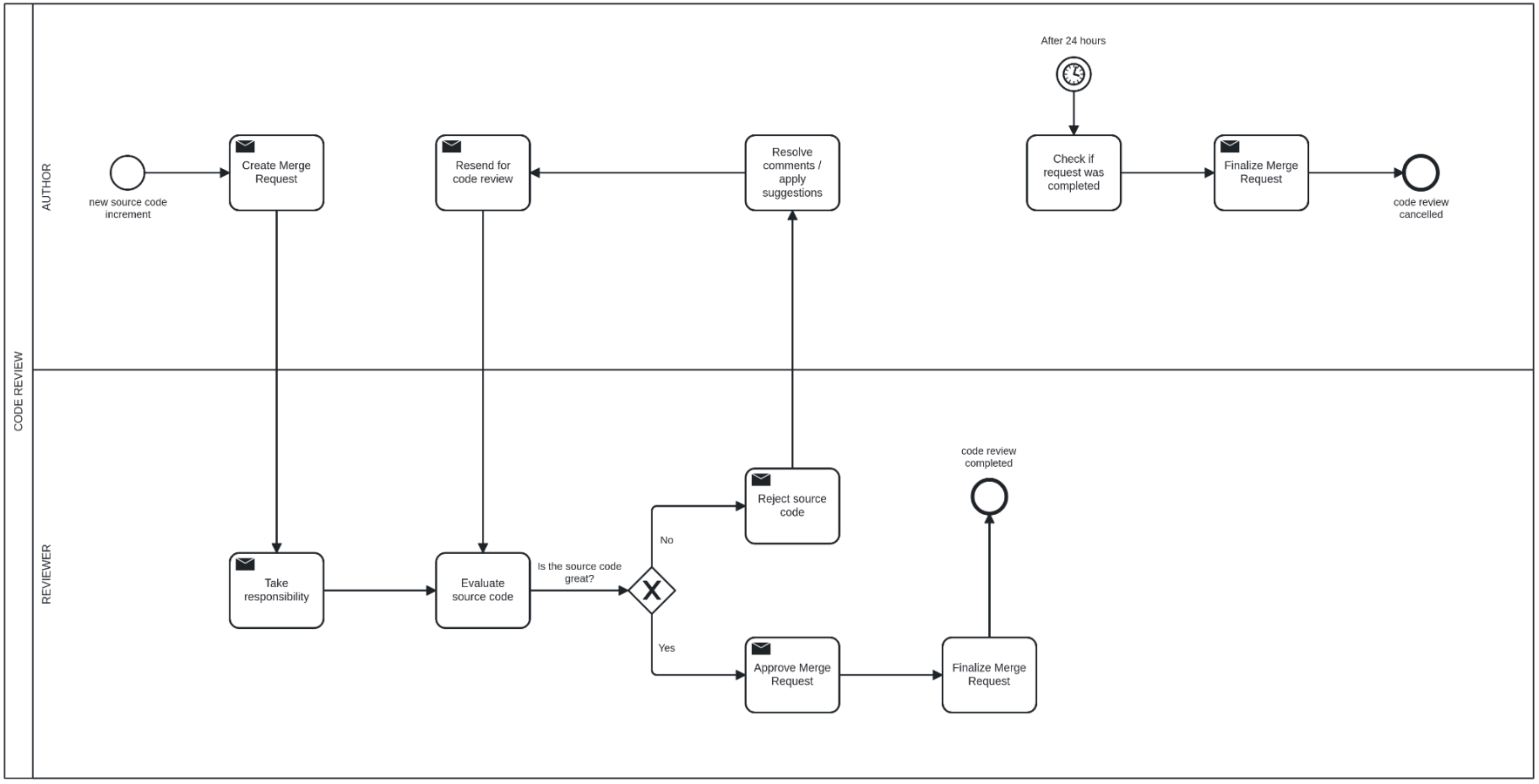
O planejamento de um estudo de caso estabelece como ele será estruturado, alinhando as ações aos objetivos definidos durante a fase de escopo (WOHLIN *et al*., 2012). Essa etapa envolve atividades como a seleção do contexto, escolha dos participantes, definição dos objetivos de avaliação e recursos necessários para a execução do estudo.

Neste estudo de caso, busca-se compreender os impactos da automação das comunicações durante a revisão de código no projeto SmartRetail. Inicialmente, a equipe seria composta por cinco desenvolvedores front-end. No entanto, devido a questões de disponibilidade, a amostra final foi composta por três participantes: dois estudantes de graduação em Tecnologia da Informação e um pesquisador convidado com formação em Ciência e Tecnologia.

O objetivo principal do estudo é investigar como a comunicação de eventos de revisão de código, seja manual ou automatizada, influencia a percepção dos desenvolvedores sobre a eficiência do processo e a colaboração durante as revisões. Adicionalmente, busca-se coletar e analisar os feedbacks dos participantes, considerando seus perfis, experiências anteriores com revisões de código e opiniões sobre as abordagens de comunicação empregadas.

Para conduzir o estudo de caso, é necessário utilizar o fluxo de revisão de código adotado pela equipe front-end do SmartRetail, ilustrado na Figura 13. Também são essenciais os seguintes recursos: um documento de implantação que guie a configuração da automação, a definição da plataforma de controle de versão (GitLab), a escolha da ferramenta de comunicação (Slack) e um instrumento para coleta de dados (Google Forms).

Figura 13 – BPMN do fluxo de revisão de código do SmartRetail (front-end)



Fonte: Elaborado pelo autor (2024).

5.3. Operação

Após o planejamento e o projeto de um estudo, inicia-se a execução, que envolve a coleta e análise dos dados, marcando o início da seção operacional do estudo (WOHLIN *et al*., 2012). A fase operacional abrange três etapas: preparação, execução e validação dos dados.

5.3.1. Preparação

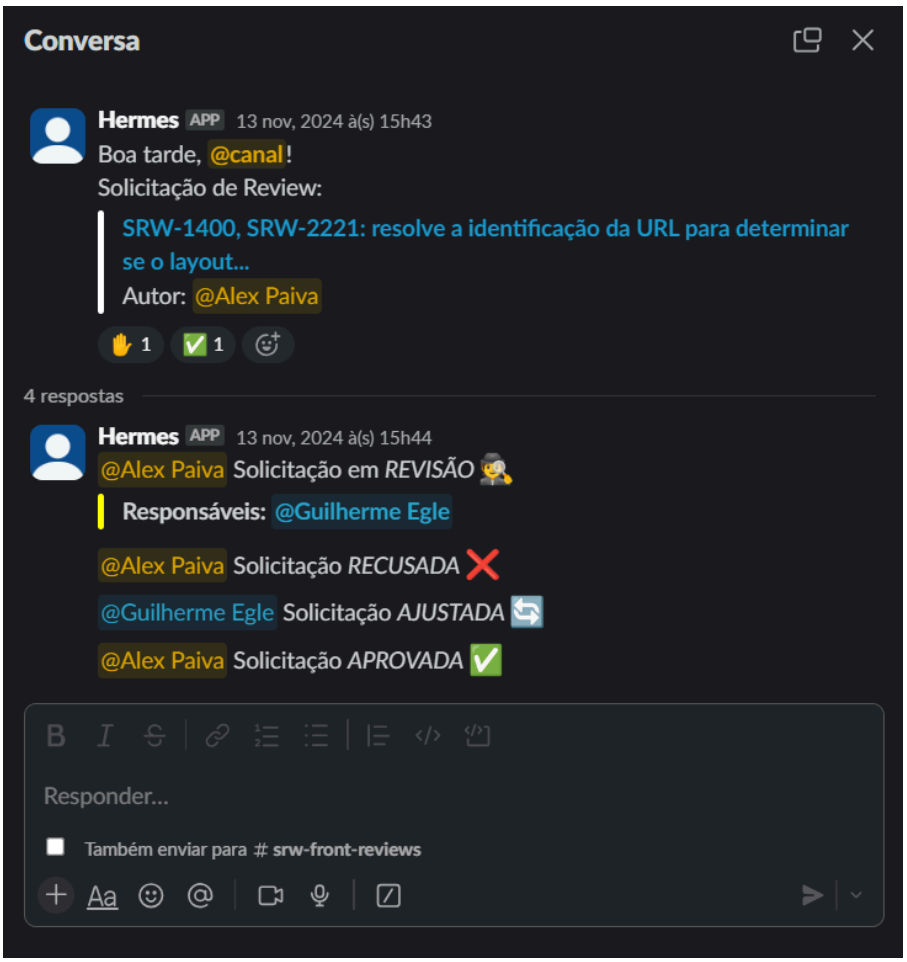
A preparação envolveu cinco etapas: elaboração do documento de implantação da solução proposta, configuração do ambiente remoto, aceitação dos participantes, criação de documento com descrição do processo de revisão de código e elaboração de questionário de avaliação.

Para confeccionar o documento de implantação, visto no Apêndice C, necessitou-se conhecer características organizacionais e de atuação da equipe de front-end do projeto SmartRetail. Por exemplo, foram observadas práticas e etapas de revisão de código (Figura 13), ferramenta de comunicação (Slack) e sistema para gerenciamento de código-fonte (GitLab). Com isso, produziu-se um manual adequado para o contexto, com introdução, visão geral da solução, etapas de implantação, testes de validação e controle de acesso/permissões.

Em seguida, ocorreu um encontro remoto com a equipe DevSecOps responsável pela manutenção do SmartRetail. Durante o encontro, foi apresentada a ferramenta Smart Review, realizado o levantamento de necessidades para a subida do ambiente (recursos) e definida a estratégia de implantação: configurar o ambiente e realizar testes por um período.

A Figura 14 exibe um teste realizado para verificar o estado de funcionamento do Smart Review. Para isso, houve a criação de um "*merge request*", atribuição de revisor, recusa solicitação (simular que existem melhorias ou correções a serem efetuadas), ajuste da solicitação e aprovação da solicitação. Observa-se que para cada evento existe um mensagem pré-definida seguido de um símbolo para criar uma representação de fácil compreensão.

Figura 14 – Revisão de código com uso do *chatbot* Hermes (Slack)

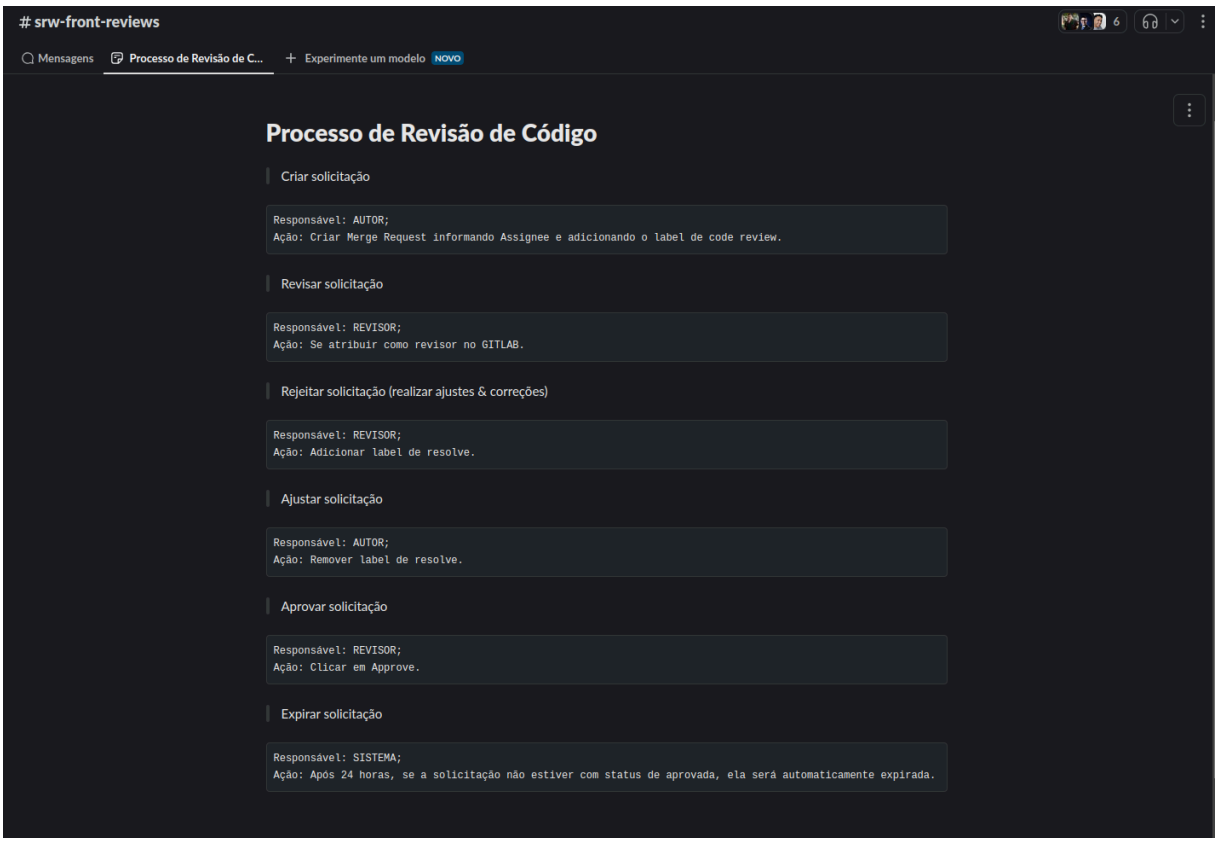


Fonte: Elaborado pelo autor (2024).

Após alinhar com a equipe DevSecOps sobre a disponibilização da solução, foi realizada uma reunião com os desenvolvedores front-end do SmartRetail. Durante a reunião, a ferramenta Smart Review foi demonstrada e a necessidade de avaliação foi explicada. Em seguida, os desenvolvedores foram convidados a manifestar interesse em participar do estudo.

Com a aceitação dos participantes, iniciou-se a preparação da documentação para orientá-los sobre em que momento há disparo de notificações de eventos de revisão de código. Para isso, criou-se uma aba contextual no Slack, que apresenta todas as etapas, os responsáveis pela condução e os passos necessários. A Figura 15 mostra o documento produzido.

Figura 15 – Aba de processo de revisão de código (Slack)



Fonte: Elaborado pelo autor (2024).

Por fim, foi elaborado um questionário que incluiu as seções de identificação, revisão de código e processo de comunicação, com o objetivo de conhecer o perfil dos participantes, sua experiência com revisão de código e a avaliação das abordagens de comunicação.

5.3.2. Execução

A execução do estudo foi conduzida em dois momentos: de 4 a 15 de novembro de 2024, com a comunicação das etapas realizada manualmente pelos envolvidos no processo de revisão; e de 18 a 29 de novembro de 2024, utilizando o Smart Review. Durante a realização houveram três feriados: Proclamação da República (15 de novembro), Consciência Negra (20 de novembro) e Padroeira de Natal – Nossa Senhora da Apresentação (21 de novembro).

Após o encerramento dos dois momentos, foi disponibilizado de 6 a 9 de dezembro de 2024 um questionário para coletar feedbacks, sugestões e comentários dos participantes. Paralelamente, foi construída uma base de dados

com informações de cada solicitação de revisão (identificação, autor, revisor, data de início, data de finalização, expirado), utilizando técnica de mineração de dados aplicada ao canal de comunicação de revisões.

5.3.3. Validação de Dados

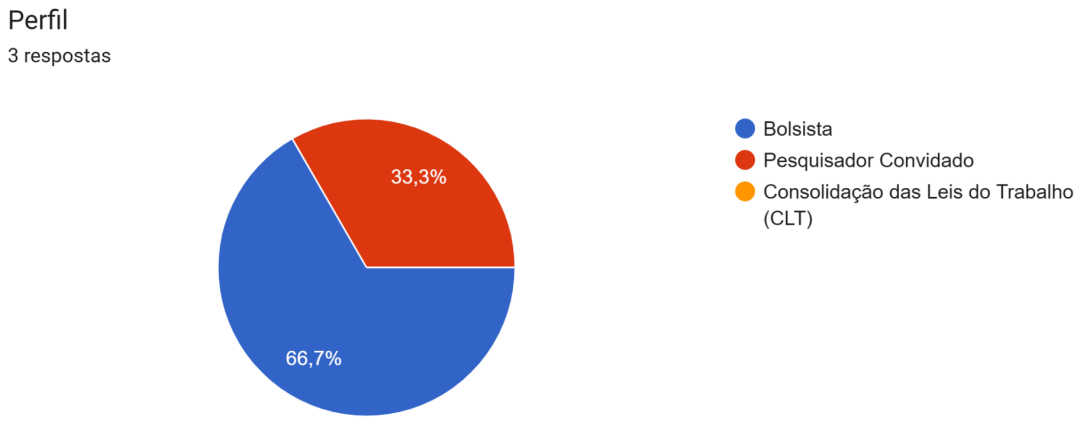
A validação dos dados foi realizada por meio da verificação das respostas no formulário, incluindo a análise da quantidade de submissões e do preenchimento completo dos campos. Três participantes responderam 17 das 19 perguntas, todas de múltipla escolha, e apenas os campos de comentários e sugestões ficaram sem respostas.

5.4. Resultados

Após a coleta dos dados na fase operacional, busca-se extrair conclusões fundamentadas a partir desses dados (WOHLIN *et al*., 2012). Nesta seção, serão analisados os resultados do questionário, com o objetivo de avaliar os feedbacks dos participantes.

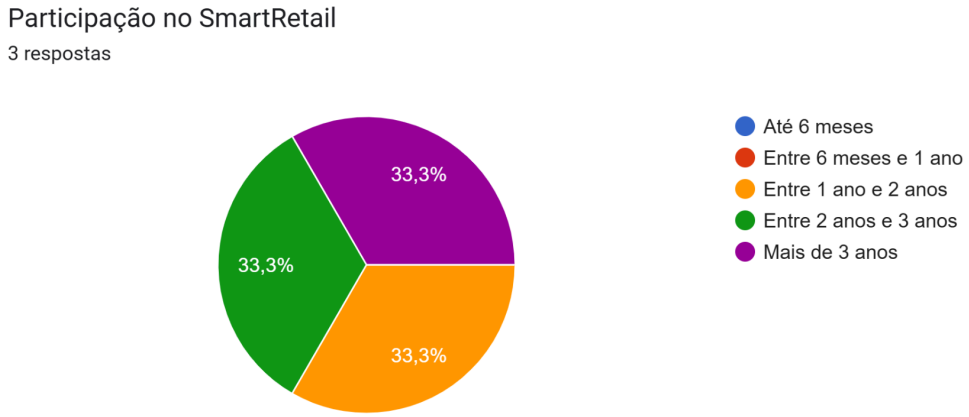
Conforme ilustrado no Gráfico 1, 66,7% dos respondentes eram bolsistas (2 participantes) e 33,3% eram pesquisadores convidados (1 participante). No Gráfico 2, observa-se que todos os participantes (desenvolvedores front-end) tinham mais de um ano de experiência no projeto SmartRetail, sendo a maioria (66,6%) com dois anos ou mais. Isso sugere um nível razoável de familiaridade com práticas recorrentes no desenvolvimento de software, como a revisão de código.

Gráfico 1 – Perfil dos envolvidos



Fonte: Elaborado pelo autor (2025).

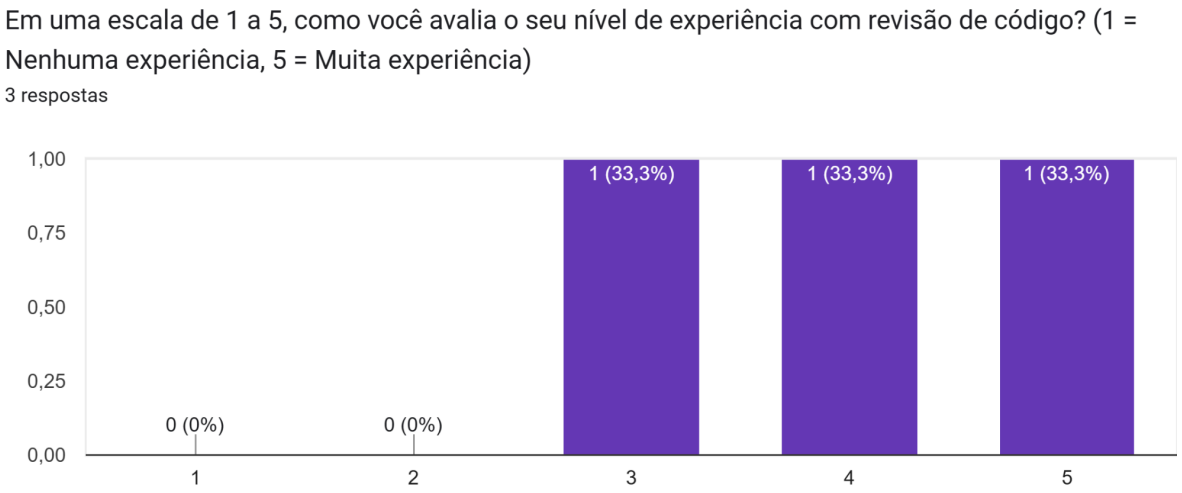
Gráfico 2 – Participação no SmartRetail



Fonte: Elaborado pelo autor (2025).

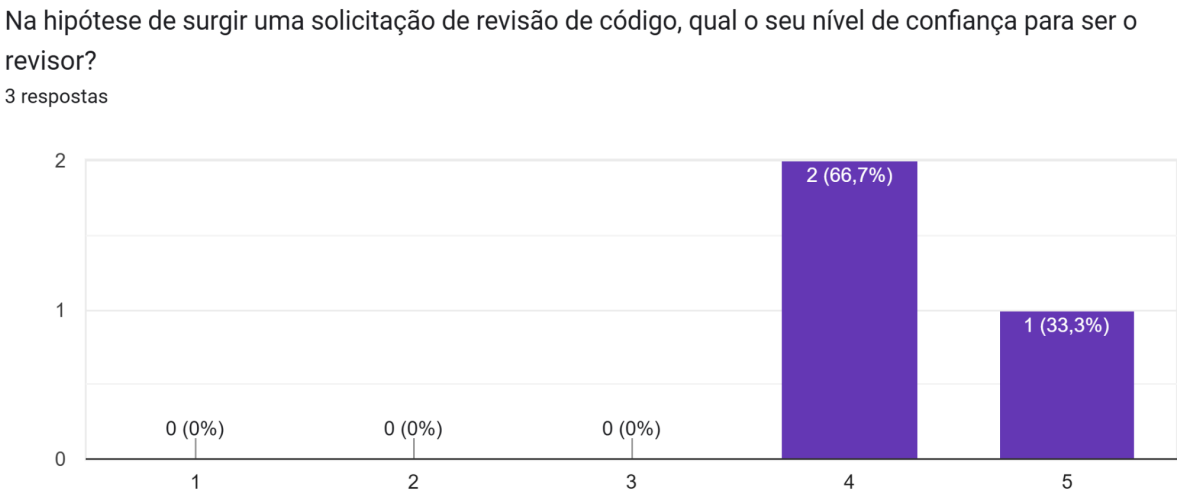
Quando questionados sobre seus níveis de experiência em revisão de código (Gráfico 3), utilizando uma escala de 1 a 5, um participante avaliou-se com "5" (muita experiência), enquanto os demais indicaram "3" e "4", apontando um nível médio a alto. Sobre a confiança em realizar revisões de código (Gráfico 4), dois participantes marcaram "4" e um indicou "5", evidenciando um alto grau de confiança, possivelmente associado ao domínio do processo empregado no projeto.

Gráfico 3 – Experiência com revisão de código



Fonte: Elaborado pelo autor (2025).

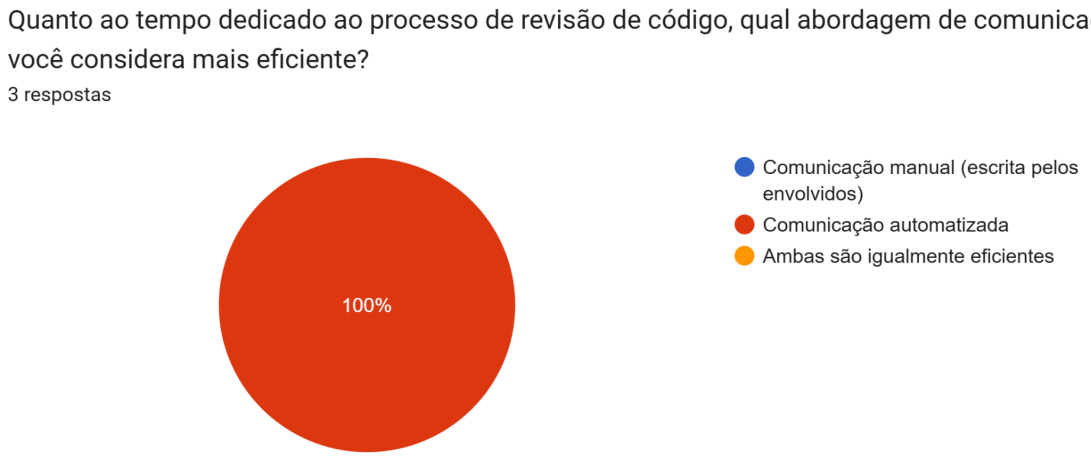
Gráfico 4 – Nível de confiança para ser revisor



Fonte: Elaborado pelo autor (2025).

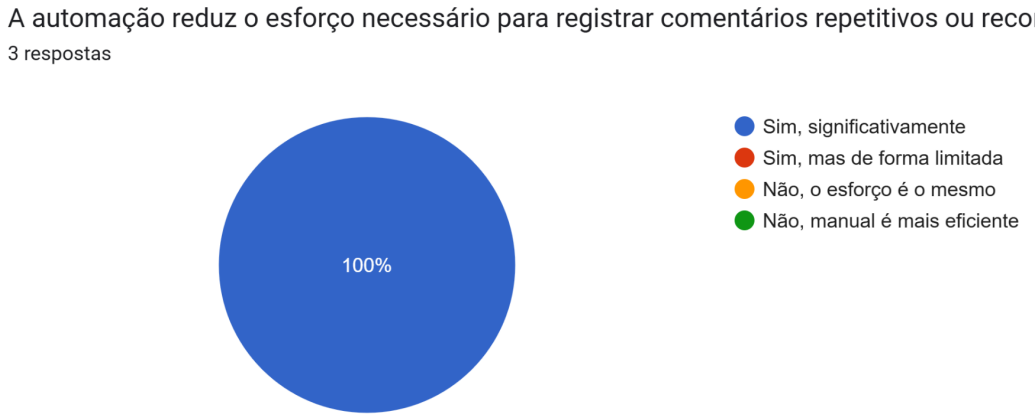
Com relação à eficiência das abordagens de comunicação (Gráfico 5), 100% dos participantes indicaram preferência pela comunicação automatizada, destacando o Smart Review como uma solução eficiente para economizar tempo durante o processo de revisão. Além disso, quanto ao impacto da automação na redução do esforço para registrar comentários repetitivos ou recorrentes (Gráfico 6), todos os participantes afirmaram que a automação contribui significativamente para essa redução.

Gráfico 5 – Forma de comunicação mais eficiente para revisão de código



Fonte: Elaborado pelo autor (2025).

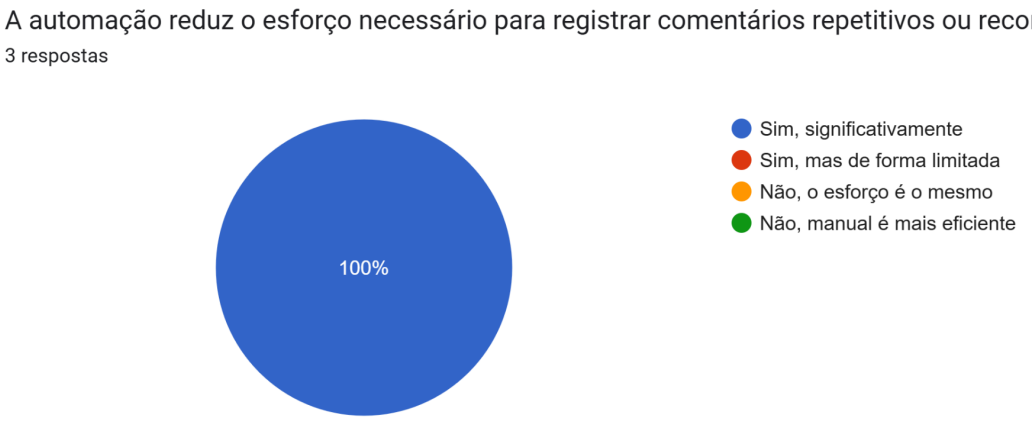
Gráfico 6 – Redução de comentários repetitivos/recorrentes



Fonte: Elaborado pelo autor (2025).

No Gráfico 7, todos os participantes concordaram que os feedbacks gerados pela automação ajudam a economizar tempo sem comprometer a qualidade. No entanto, ao avaliar se esses feedbacks cobrem todos os pontos necessários em uma revisão (Gráfico 8), as opiniões se dividiram: 33,3% afirmaram que sim (sempre), enquanto 66,7% indicaram que, ocasionalmente, é necessário complementá-los manualmente.

Gráfico 7 – Qualidade de feedback



Fonte: Elaborado pelo autor (2025).

Gráfico 8 – Completude de feedback

Fonte: Elaborado pelo autor (2025).

Quanto à clareza da comunicação (Gráfico 9), todos os participantes consideraram as mensagens geradas automaticamente mais claras e objetivas que as escritas manualmente, sendo 66,7% afirmando que isso ocorre na maioria das vezes e 33,3% indicando que sempre.

Gráfico 9 – Clareza de mensagens

Fonte: Elaborado pelo autor (2025).

Na perspectiva de consistência na comunicação (Gráfico 10), todos os participantes indicaram que a automação frequentemente oferece maior consistência nos comentários em comparação ao método manual. Esse resultado permite inferir que a automação contribui para a padronização do feedback, reduzindo ambiguidades e garantindo maior alinhamento com as diretrizes e padrões estabelecidos pela equipe.

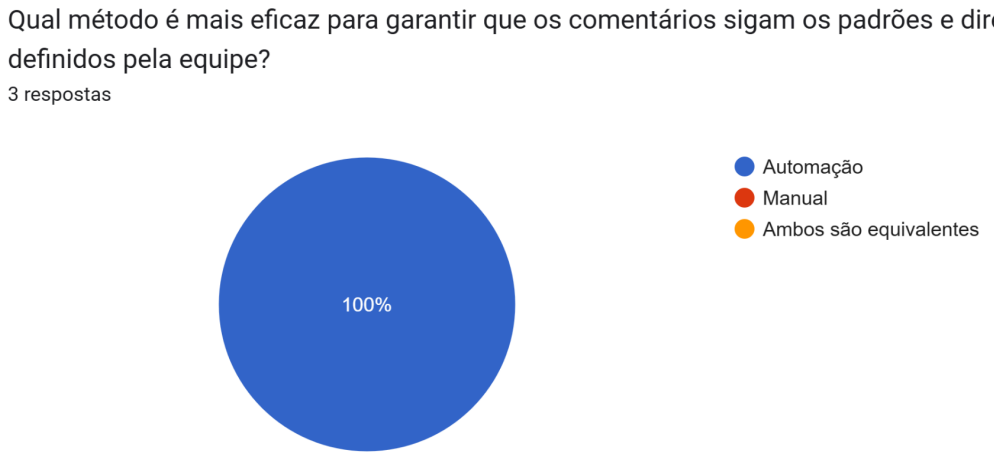
Gráfico 10 – Consistência nos comentários

Fonte: Elaborado pelo autor (2025).

De forma complementar, o Gráfico 11 reflete qual método de comunicação é mais eficaz para garantir que os comentários sigam os padrões e diretrizes da equipe: todos os participantes escolheram a automação. Além disso, todos os

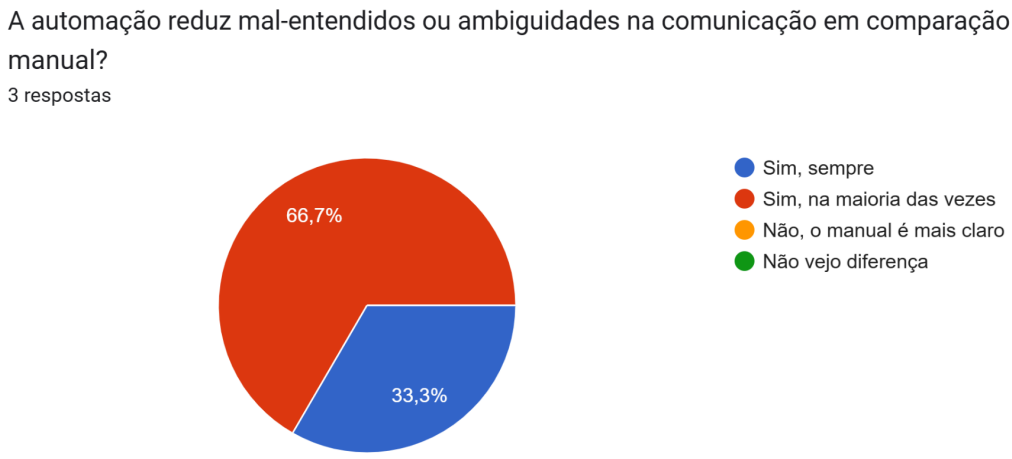
participantes afirmaram que a automação reduz mal-entendidos ou ambiguidades na comunicação em comparação ao método manual (Gráfico 12), com 66,7% indicando que isso ocorre na maioria das vezes e 33,3% afirmando que sempre.

Gráfico 11 – Padrões e diretrizes



Fonte: Elaborado pelo autor (2025).

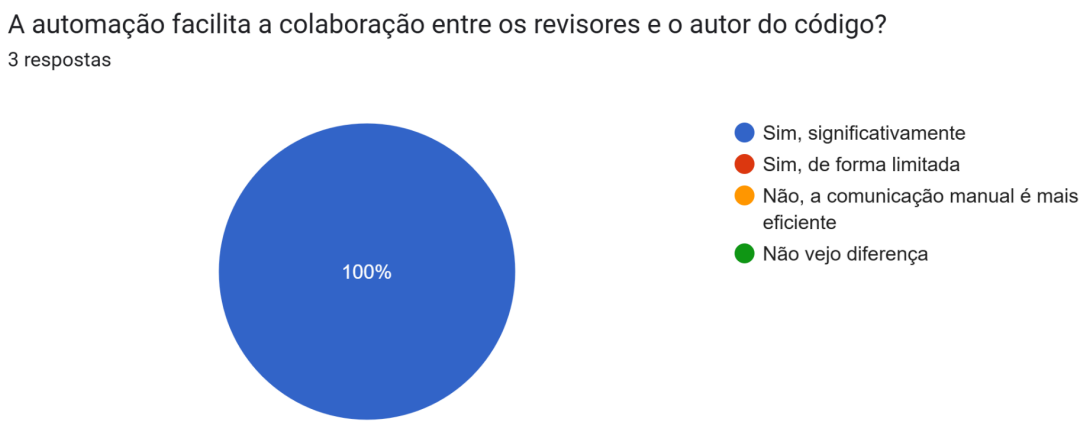
Gráfico 12 – Redução de ambiguidades na comunicação



Fonte: Elaborado pelo autor (2025).

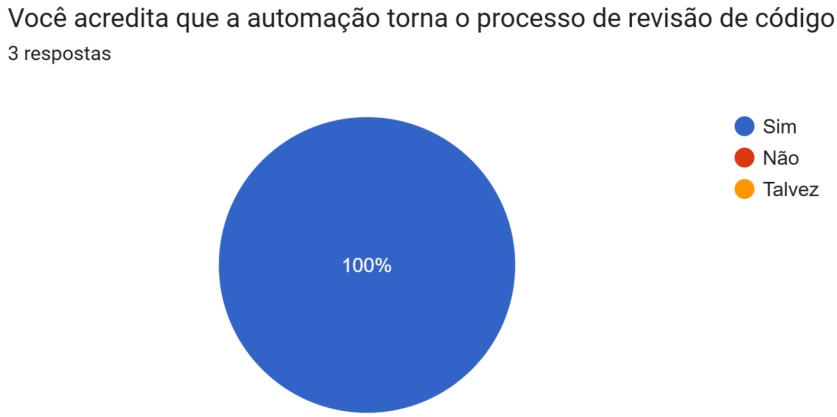
Em relação à colaboração e preferência na comunicação, todos os participantes indicaram que a automação facilita significativamente a colaboração entre revisores e o autor do código (Gráfico 13). Sobre a satisfação com o processo de revisão de código automatizado (Gráfico 14), 100% dos participantes responderam que sim.

Gráfico 13 – Facilitação na colaboração



Fonte: Elaborado pelo autor (2025).

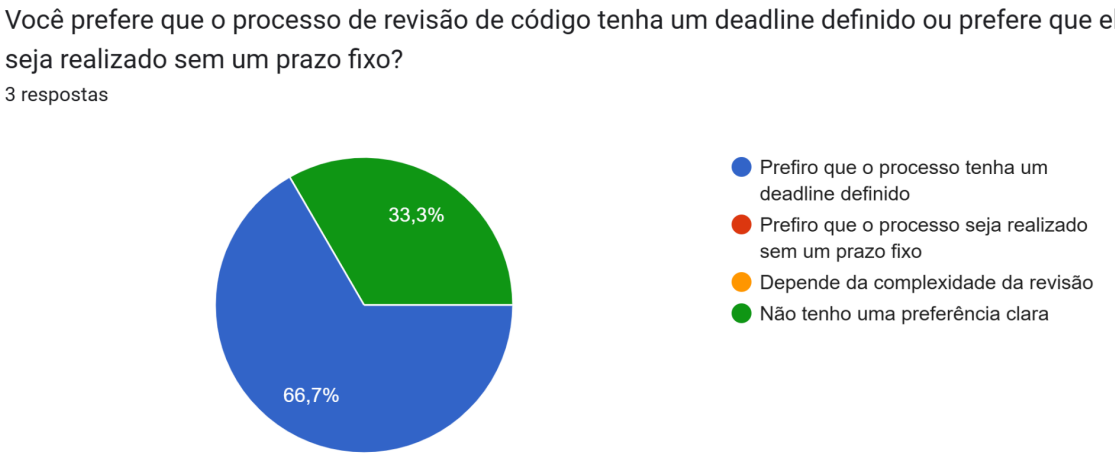
Gráfico 14 – Satisfação na revisão de código



Fonte: Elaborado pelo autor (2025).

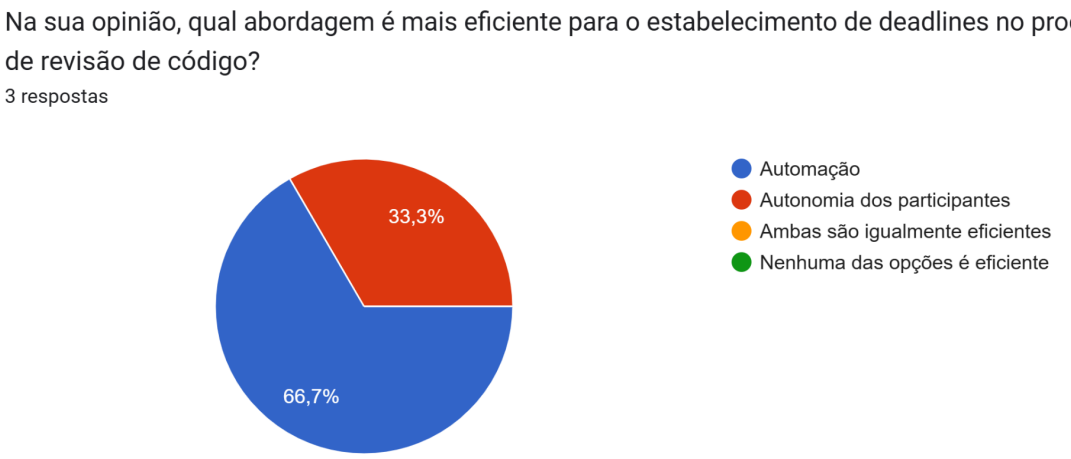
Sobre o aspecto de deadlines na revisão de código moderna (Gráfico 15), 66,7% dos participantes disseram preferir que o processo tenha um prazo definido, enquanto 33,3% não expressaram uma preferência clara. Em seguida, ao serem questionados sobre qual abordagem consideram mais eficiente para estabelecer deadlines de revisão (Gráfico 16), 66,7% indicaram a automação 33,3% preferiram a autonomia dos participantes.

Gráfico 15 – Deadline para revisão de código



Fonte: Elaborado pelo autor (2025).

Gráfico 16 – Abordagem mais eficiente para estabelecer deadlines



Fonte: Elaborado pelo autor (2025).

A análise dos dados coletados no questionário revela várias percepções valiosas sobre a eficácia da automação no processo de revisão de código. Todos os participantes demonstraram uma clara preferência pela comunicação automatizada, destacando a sua eficiência, clareza e capacidade de reduzir o esforço com comentários repetitivos.

Essa tendência está alinhada com a visão de que ferramentas automatizadas, como o Smart Review, podem otimizar o tempo de revisão sem

comprometer a qualidade, especialmente em contextos de desenvolvimento contínuo, como o projeto SmartRetail.

Os participantes também indicaram que a automação contribui significativamente para a consistência na comunicação. A padronização dos feedbacks gerados automaticamente foi considerada uma vantagem importante, já que assegura maior alinhamento com as diretrizes e padrões estabelecidos pela equipe, reduzindo ambiguidades e mal-entendidos. Isso sugere que a automação não só melhora a eficiência, mas também fortalece a qualidade e a confiança no processo de revisão de código.

A preferência por estabelecer deadlines claros para o processo de revisão também foi evidente, com a maioria dos participantes indicando que a automação seria a forma mais eficiente de gerenciar esses prazos. Esse dado aponta para uma tendência crescente de otimizar o tempo no desenvolvimento de software sem comprometer a qualidade do código, ressaltando a importância de ferramentas que integrem automação e gestão de prazos de maneira eficaz.

Em suma, os resultados indicam que a automação pode otimizar o processo de revisão de código, trazendo benefícios significativos em termos de eficiência, clareza, consistência e colaboração, alinhando-se bem à cultura de desenvolvimento ágil e contínuo do projeto SmartRetail.

6. CONSIDERAÇÕES FINAIS

Este trabalho visou automatizar a comunicação na revisão de código. Os resultados do estudo de caso com a equipe front-end do projeto SmartRetail mostram que a integração do MCR às plataformas Git, com notificações via ESN, melhora a eficiência, clareza e consistência do processo, reduzindo comentários repetitivos (feitos manualmente), padronizando feedbacks e facilitando o cumprimento de prazos.

A relevância deste estudo está na concepção de uma aplicação leve e simplificada (API), sem a necessidade de instalação no ambiente de cada desenvolvedor. Essa aplicação atua como uma ponte entre as ferramentas de controle de repositórios Git e os sistemas de comunicação, permitindo a definição de etapas-chave de notificação, desde o fluxo principal de revisão de código até a definição de prazos.

Apesar dos resultados alcançados, o sistema proposto apresenta algumas limitações, como a inexistência de personalização das mensagens predefinidas para cada estágio do processo de revisão de código, a necessidade de operar dentro das restrições das APIs de sistemas de terceiros (como o Slack), desafio de escalar carga de trabalho e a dificuldade em se recuperar de falhas no envio de notificações.

6.1. Trabalhos futuros

Os resultados obtidos neste estudo indicam possibilidades de aprimoramento e expansão do sistema proposto, levando em consideração as limitações identificadas. Investigações futuras podem explorar a integração de uma maior variedade de plataformas e ferramentas de controle de código, com o objetivo de ampliar a aplicabilidade do sistema em diferentes ambientes de desenvolvimento.

Além disso, a aplicação de tecnologias de inteligência artificial para a criação de feedbacks automáticos é uma área promissora para aumentar a eficiência do processo de revisão de código. A utilização de IA permitiria a análise semântica do código, identificando padrões e sugerindo melhorias, o que reduziria ainda mais o tempo gasto com revisões manuais e aumentaria a consistência dos feedbacks.

Outro aspecto importante a ser abordado é a tolerância a falhas. A solução atual ainda enfrenta desafios na recuperação de falhas no envio de notificações, como a perda de mensagens quando múltiplas solicitações de revisão ocorrem simultaneamente. Uma possível solução seria adotar a arquitetura idealizada com o modelo Producer-Worker, onde o produtor geraria as solicitações e os trabalhadores processariam essas solicitações de forma assíncrona, garantindo a entrega das notificações, mesmo em cenários de falhas temporárias.

Uma linha de trabalho futuro seria a implementação de uma integração de mão dupla entre a rede social corporativa e a ferramenta de repositórios. Isso permitiria que ações como aprovações e comentários fossem automaticamente refletidas em *pull requests* ou *merge requests*, promovendo maior sincronia entre as ferramentas e melhorando a colaboração da equipe.

Por fim, a realização de estudos de caso em diferentes contextos de desenvolvimento e equipes seria valiosa para validar e refinar as soluções propostas, ampliando sua aplicabilidade e robustez em diversos ambientes. Além disso, a análise de grandes volumes de dados, como em sistemas de revisão de código em larga escala, poderia fornecer *insights* valiosos para a evolução do sistema.

REFERÊNCIAS

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de Software**: Uma Abordagem Profissional. Porto Alegre: AMGH, 2016.

BADAMPUDI, D.; UNTERKALMSTEINER, M.; BRITTO, R. 2023. Modern Code Reviews—Survey of Literature and Practice. **ACM Transactions on Software Engineering and Methodology**, New York: Association for Computing Machinery, v. 32, n. 4, art. 107, p. 61, jul. 2023. Disponível em: https://doi.org/10.1145/3585004. Acesso em: 22 nov. 2024.

ACKERMAN, A. F.; FOWLER, P. J.; EBENAU, R. G. Software inspections and the industrial production of software. **Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives**, USA: Elsevier North-Holland, p. 13–40, 1984.

FAGAN, M. A History of Software Inspections. **Software Pioneers**, Heidelberg: Springer Berlin Heidelberg, p. 562–573, 2002. Disponível em: https://doi.org/10.1007/978-3-642-59412-0_34. Acesso em: 22 nov. 2024.

SHULL, F.; SEAMAN, C. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. **IEEE Software**, v. 25, n. 1, p. 88–90, 2008. Disponível em: https://doi.org/10.1109/MS.2008.7. Acesso em: 22 nov. 2024.

SADOWSKI, C.; SÖDERBERG, E.; CHURCH, L.; SIPKO, M.; BACCHELLI, A. Modern code review: A case study at google. **Proceedings of the 40th International Conference on Software Engineering**: Software Engineering in Practice, New York: Association for Computing Machinery, p. 181–190, 2018. Disponível em: https://doi.org/10.1145/3183519.3183525. Acesso em: 22 nov. 2024.

RIGBY, P. C.; BIRD, C. Convergent contemporary software peer review practices. **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**, New York: Association for Computing Machinery, p. 202–212, 2013. Disponível em: https://doi.org/10.1145/2491411.2491444. Acesso em: 22 nov. 2024.

BAVOTA, G.; RUSSO, B. Four eyes are better than two: On the impact of code reviews on software quality. **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, Bremen, p. 81–90, 2015. Disponível em: https://doi.org/10.1109/ICSM.2015.7332454. Acesso em: 22 nov. 2024.

AURUM, A.; PETERSSON, H.; WOHLIN, C. State-of-the-art: software inspections after 25 years. **Software Testing, Verification & Reliability**, v. 12, n. 3, p. 133–154, 2002. Disponível em: https://doi.org/10.1002/stvr.243. Acesso em: 22 nov. 2024.

RIGBY, P.; CLEARY, B.; PAINCHAUD, F.; STOREY, M.; GERMAN, D. Contemporary peer review in action: Lessons from open source development. **IEEE software**, v. 29, n. 6, p. 56–61, 2012. Disponível em: https://doi.org/10.1109/MS.2012.24. Acesso em: 22 nov. 2024.

SIOW, J. K.; GAO, C.; FAN, L.; CHEN, S.; LIU, Y. CORE: Automating review recommendation for code changes. **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**, p. 284–295, 2020. Disponível em: https://doi.org/10.1109/SANER48275.2020.9054794. Acesso em: 22 nov. 2024.

WOLTON, D. **Pensar a Comunicação**. Brasília (Brasil): Editora Universidade de Brasília, 2004.

DEFRANCO, J. F.; LAPLANTE, P. A. Review and analysis of software development team communication research. **IEEE transactions on professional communication**, v. 60, n. 2, p. 165–182, 2017. Disponível em: https://doi.org/10.1109/TPC.2017.2656626. Acesso: 22 nov. 2024.

MAJUMDAR, A.; KRISHNA, S.; BJORN, P. Managers' Perceptions of Social Software Use in the Workplace: Identifying the Benefits of Social Software and Emerging Patterns of its Use. **Proceedings of the 19th Americas Conference on Information Systems (AMCIS)**, Chicago, 2013.

BEHRENDT, S.; RICHTER, A.; TRIER, M. Mixed methods analysis of enterprise social networks. **Computer networks**, v. 75, p. 560–577, 2014. Disponível em: https://doi.org/10.1016/j.comnet.2014.08.025. Acesso em: 22 nov. 2024.

MONTRIEF, T.; HAAS, M. R. C.; ALVAREZ, A.; GOTTLIEB, M.; SIEGAL, D.; CHAN, T. Thinking outside the inbox: Use of slack in clinical groups as a collaborative team communication platform. **AEM education and training**, v. 5, n. 1, p. 121–129, 2021. Disponível em: https://doi.org/10.1002/aet2.10497. Acesso em: 22 nov. 2024.

SALESFORCE. **What is Slack? An enterprise guide**. EUA: Salesforce, 2024. *E-book*.

LOKMAN, A. S.; AMEEDEEN, M. A. Modern Chatbot Systems: A Technical Review. **Proceedings of the Future Technologies Conference (FTC) 2018**, Cham: Springer International Publishing, p. 1012–1023, 2019. Disponível em: https://doi.org/10.1007/978-3-030-02683-7_75. Acesso em: 23 nov. 2024.

WEIZENBAUM, J. ELIZA—a computer program for the study of natural language communication between man and machine. **Communications of the ACM**, v. 9, n. 1, p. 36–45, 1966.

DAVIS, J.; DANIELS, R. **Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale**. USA: O'Reilly Media, 2016.

STEFAN, O. Version control systems. **Computer Systems and Telematics pp**, p. 11–13, 2009.

CHACON, S.; STRAUB, B. **Pro Git**. Apress, 2014.

GHODKE, G. M.; CHAVAN, T. An overview of git. **International Journal of Scientific Research in Modern Science and Technology**, v. 3, n. 6, p. 17–23,

2024. Disponível em: https://doi.org/10.59828/ijsrmst.v3i6.216. Acesso em: 23 nov. 2024.

REDHAT. **What is a webhook?**. 1 fev. 2024. Disponível em: https://www.redhat.com/en/topics/automation/what-is-a-webhook. Acesso em: 23 nov. 2024.

HETHEY, J. M. **GitLab Repository Management**. Birmingham, England: Packt Publishing, 2013.

WINTERS, T.; WRIGHT, H.; MANSHREK, T. **Software engineering at Google**: Lessons learned from programming over time. Sebastopol, CA, USA: O'Reilly Media, 2020.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. **2013 35th International Conference on Software Engineering (ICSE)**, San Francisco, CA, USA, p. 712–721, 2013. Disponível em: https://doi.org/10.1109/ICSE.2013.6606617. Acesso em: 24 nov. 2024.

MCINTOSH, S.; KAMEI, Y.; ADAMS, B.; HASSAN, A. E. The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. **Proceedings of the 11th Working Conference on Mining Software Repositories**, New York: ACM, p. 192–201, 2014.

RAWAT, S. **Getting started with review board**. Birmingham, England: Packt Publishing, 2014.

BIASE, M. D.; BRUNTINK, M.; BACCHELLI, A. A security perspective on code review: The case of chromium. **2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)**, p. 21–30, 2016.

TANG, M. **Caesar**: A Social Code Review Tool for Programming Education. Orientador: Robert Miller. 2011. 65 f. Dissertação (Mestrado em Engenharia Elétrica e Ciência da Computação) – Massachusetts Institute of Technology, Estados Unidos, 2011.

HOLZMANN, G. J. SCRUB: a tool for code reviews. **Innovations in Systems and Software Engineering**, v. 6, n. 4, p. 311–318, 5 ago. 2010.

FILHO, F. J. B. G. **Please Review**: Um feed de solicitações de revisão de código integrado ao GitHub. Orientador: Fernando Filho. 2017. 44 f. Monografia (Engenharia de Software) – Universidade Federal do Rio Grande do Norte, Natal, 2017.

SADOWSKI, C.; KURNIA, I.; ROHLFS, B. Critique: Google's Code Review Tool. *In*: Winters, T.; Manshreck, T.; Wright, H. **Software Engineering at Google**. EUA: O'Reilly Media, 2020. p. 36–45.

SOMMERVILLE, I. **Software Engineering**. Munique, Germany: Pearson Studium ein Imprint von Pearson Deutschland, 2018.

WETHERBY, A. **Django Python for Web Development**: A Beginner's Guide to Mastering Django Through Practical Projects. Independently published, 2024. *E-book*.

GAGLIARDI, V. **Decoupled Django**: Understand and build decoupled Django architectures for JavaScript front-ends. Berlim, Germany: APress, 2021.

COREY BERTRAM. **django-simple-history**. Disponível em: https://django-simple-history.readthedocs.io/en/latest/index.html. Acesso em: 24 nov. 2024.

PIERFEDERICI, F. **Distributed computing with python**. Birmingham, England: Packt Publishing, 2023.

SILVA, M. D. D.; TAVARES, H. L. **Redis essentials**. Birmingham, England: Packt Publishing, 2015.

FERRARI, L.; PIROZZI, E. **Learn PostgreSQL**: Use, manage and build secure and scalable databases with PostgreSQL 16. Birmingham, England: Packt Publishing, 2023.

POHL, K.; RUPP, C. **Fundamentos da Engenharia de Requisitos**. T&M, 2012.

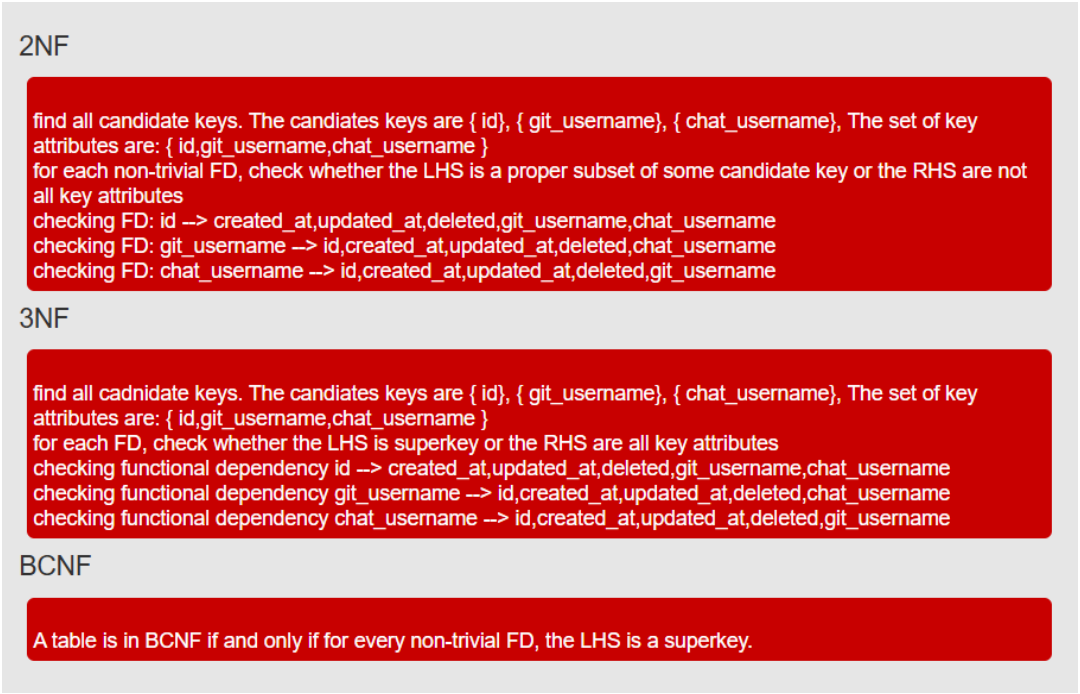
ISO/IEC/IEEE 42010:2022. **Software, systems and enterprise — Architecture description**. Geneva: International Organization for Standardization, 2022.

YIN, R. K. **Estudo de Caso**: Planejamento e Métodos. Porto Alegre: Bookman, 2001.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, G.; WESSLÉN, A. **Experimentation in Software Engineering**. Boston: Springer, 2012.

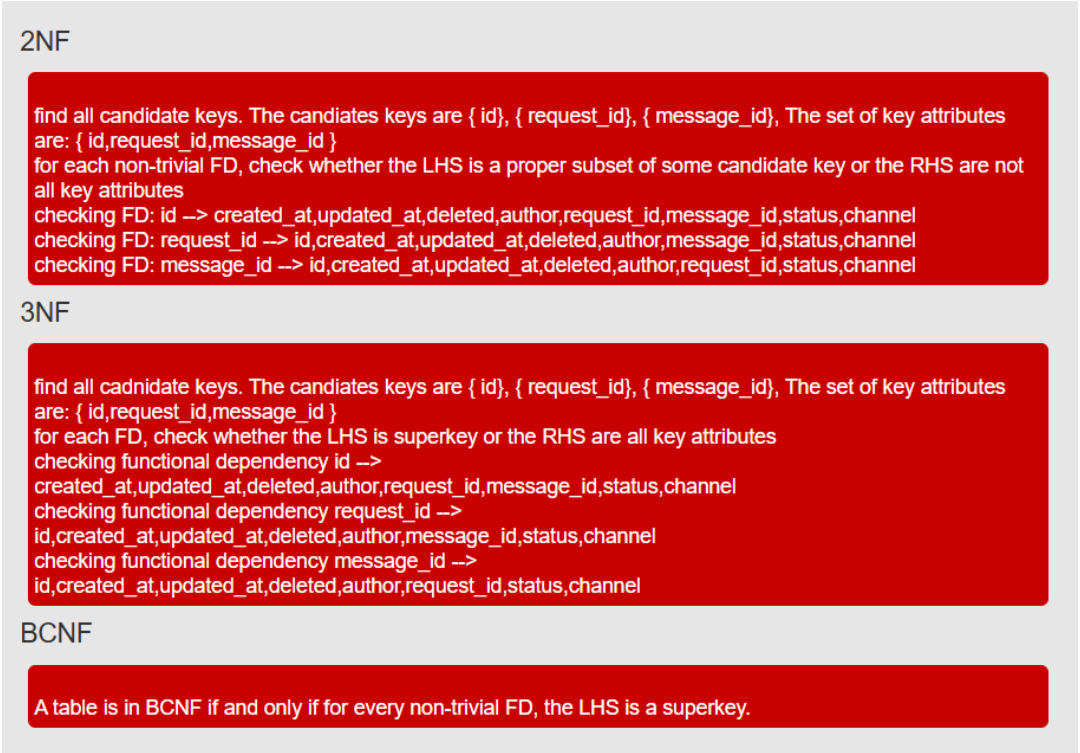
APÊNDICE A – NORMALIZAÇÃO

Figura A.1 – Verificação de forma normal de “user”



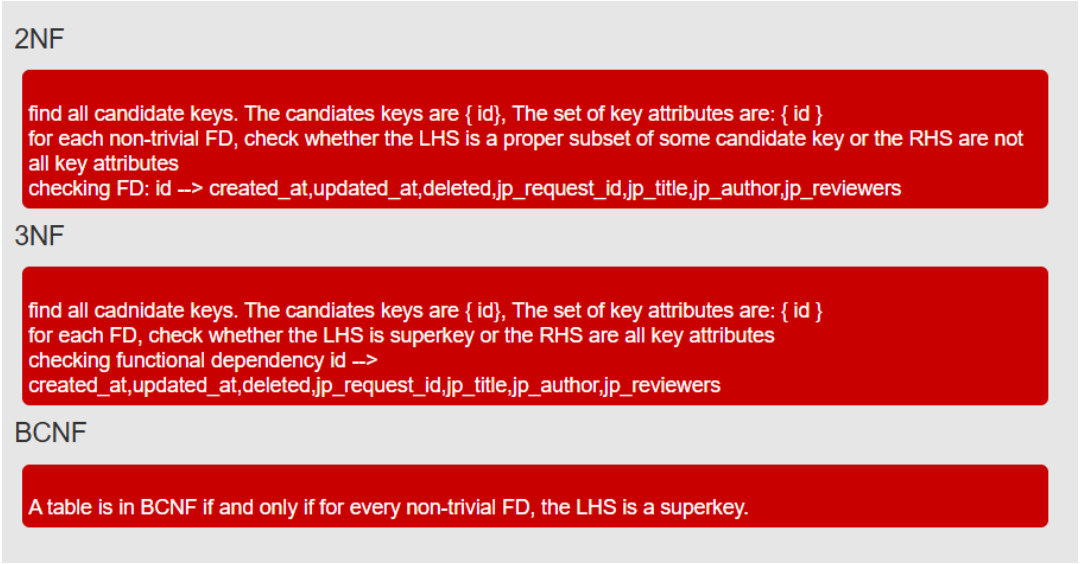
Fonte: Elaborado pelo autor (2024).

Figura A.2 – Verificação de forma normal de “request”



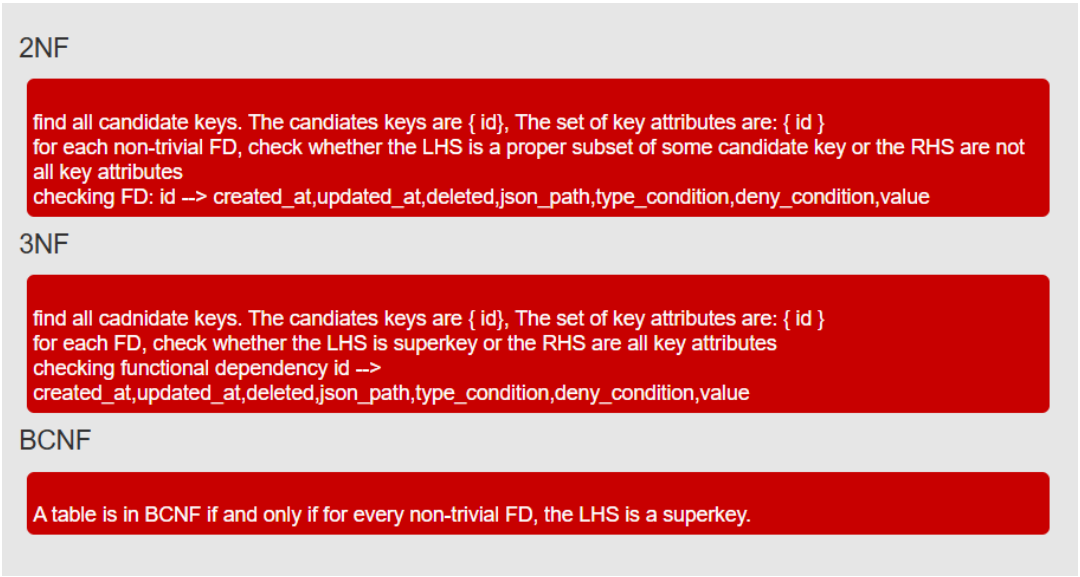
Fonte: Elaborado pelo autor (2024).

Figura A.3 – Verificação de forma normal de “data_path”



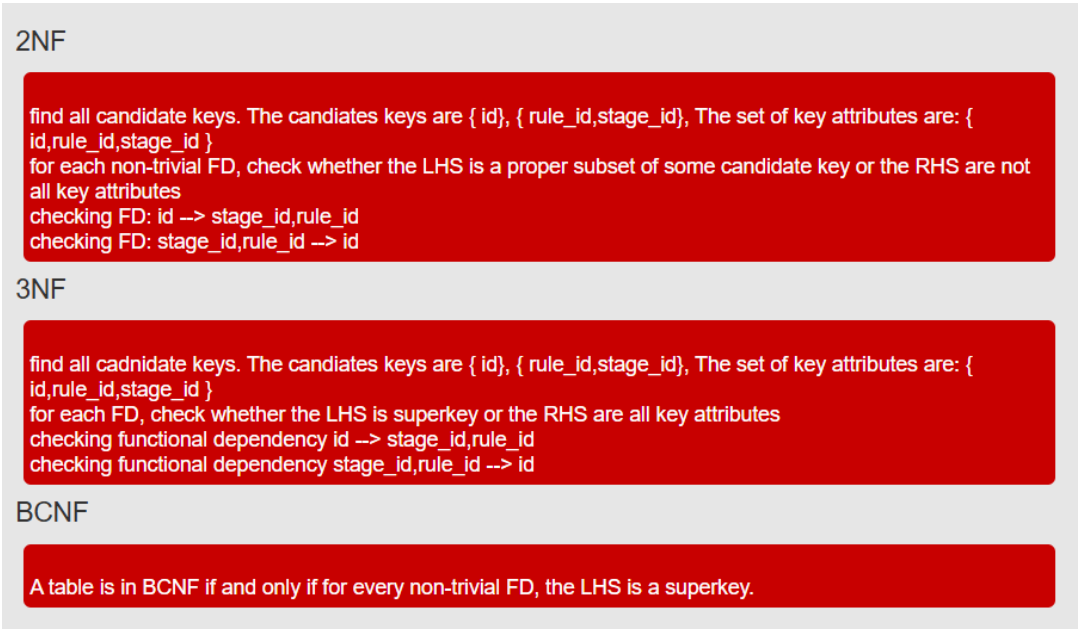
Fonte: Elaborado pelo autor (2024).

Figura A.4 – Verificação de forma normal de “rule”



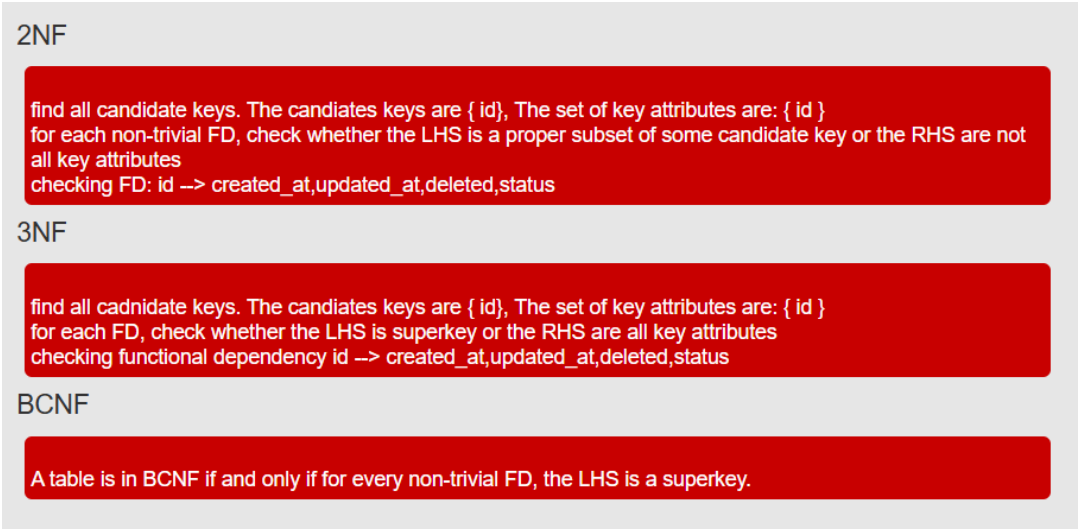
Fonte: Elaborado pelo autor (2024).

Figura A.5 – Verificação de forma normal de “stage_rule”



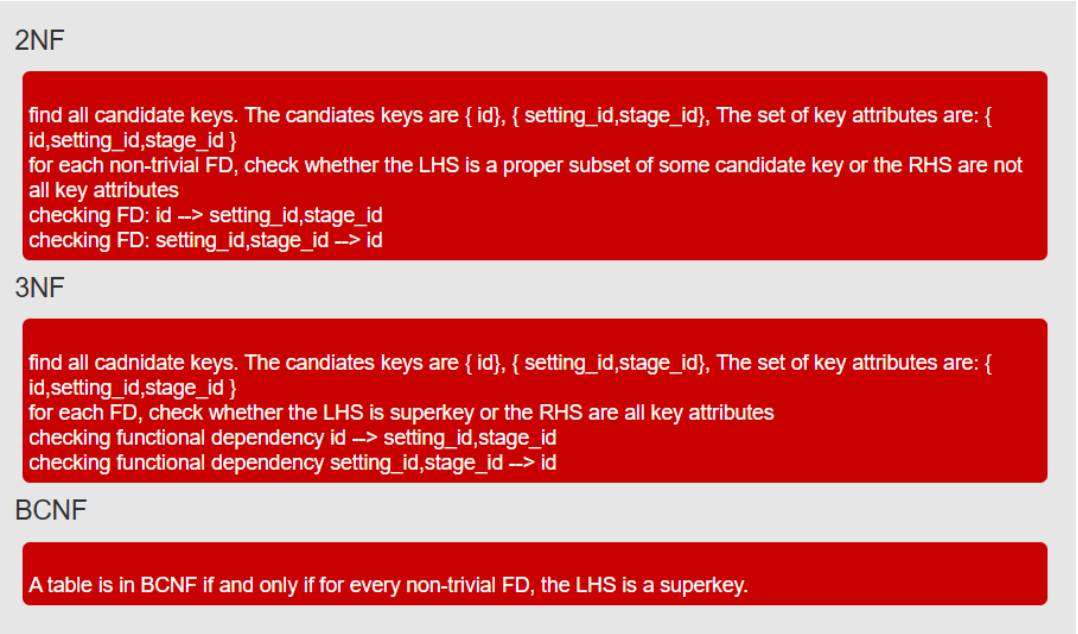
Fonte: Elaborado pelo autor (2024).

Figura A.6 – Verificação de forma normal de “stage”



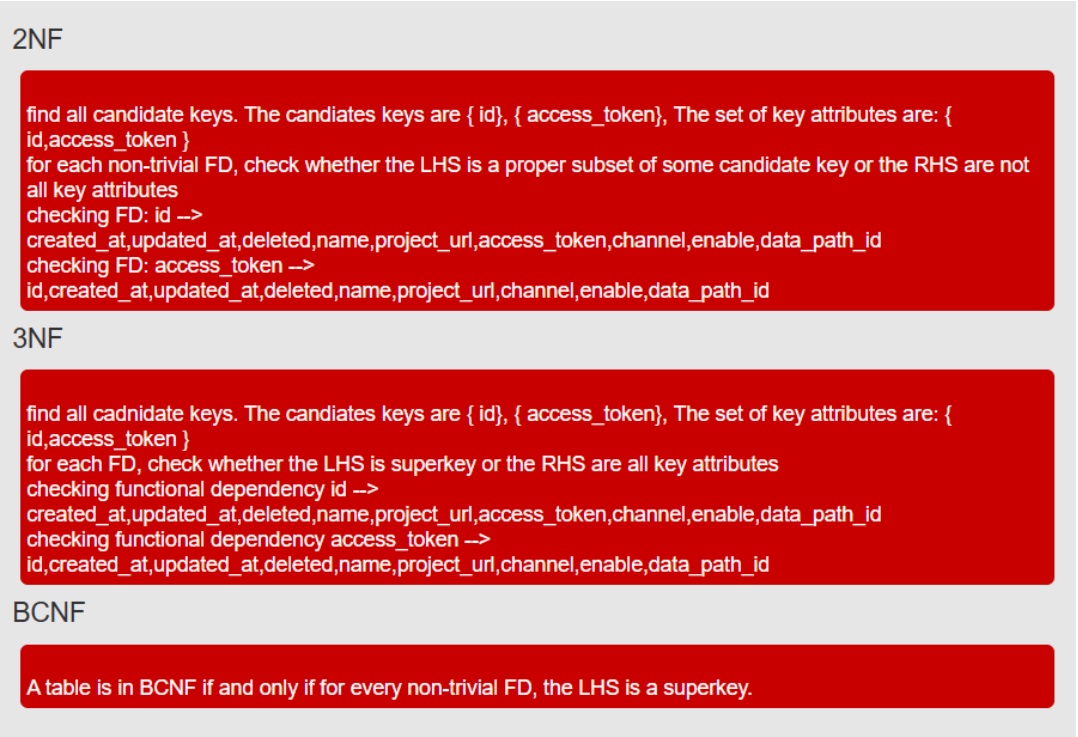
Fonte: Elaborado pelo autor (2024).

Figura A.7 – Verificação de forma normal de “setting_stage”



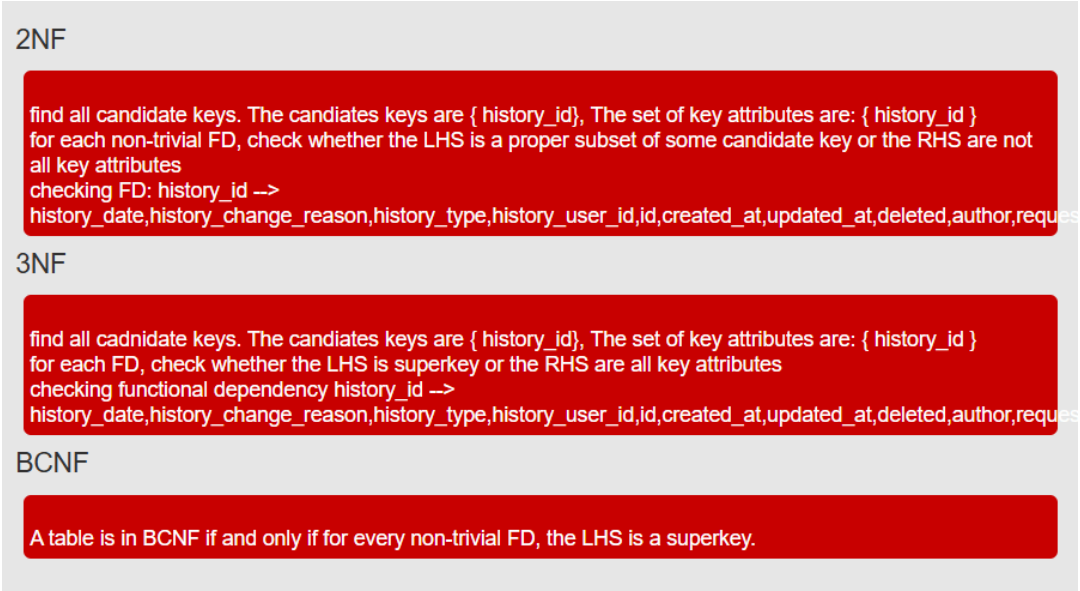
Fonte: Elaborado pelo autor (2024).

Figura A.8 – Verificação de forma normal de “setting”



Fonte: Elaborado pelo autor (2024).

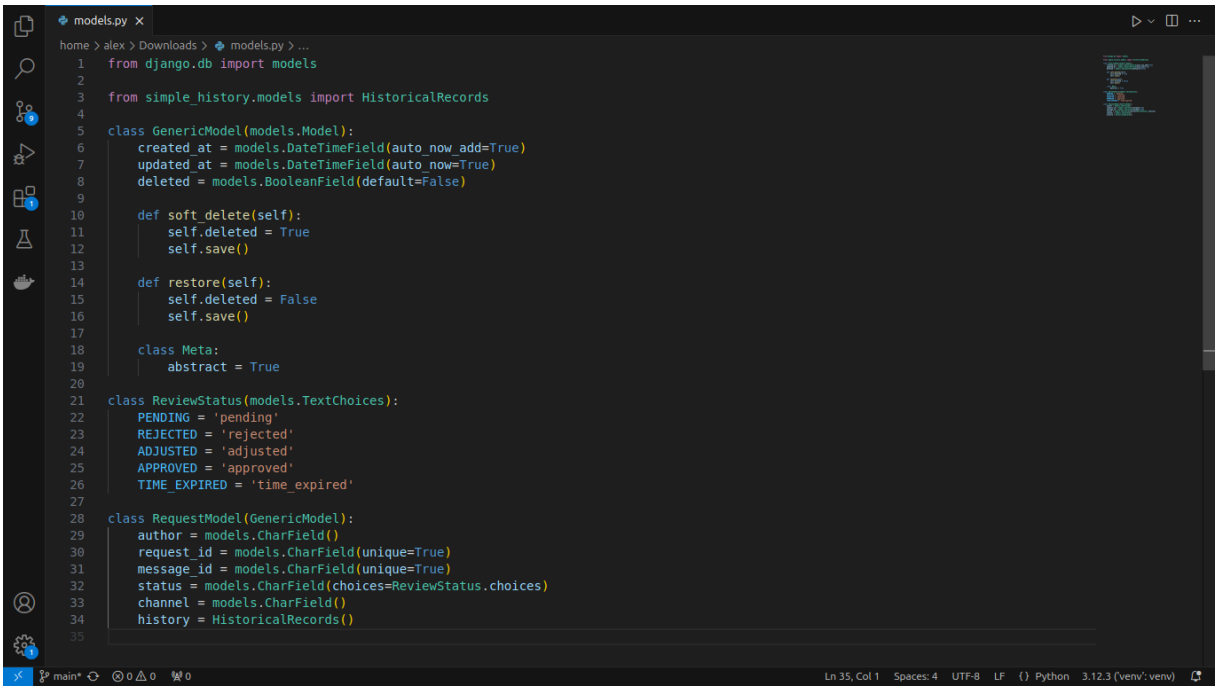
Figura A.9 – Verificação de forma normal de “historical_request”



Fonte: Elaborado pelo autor (2024).

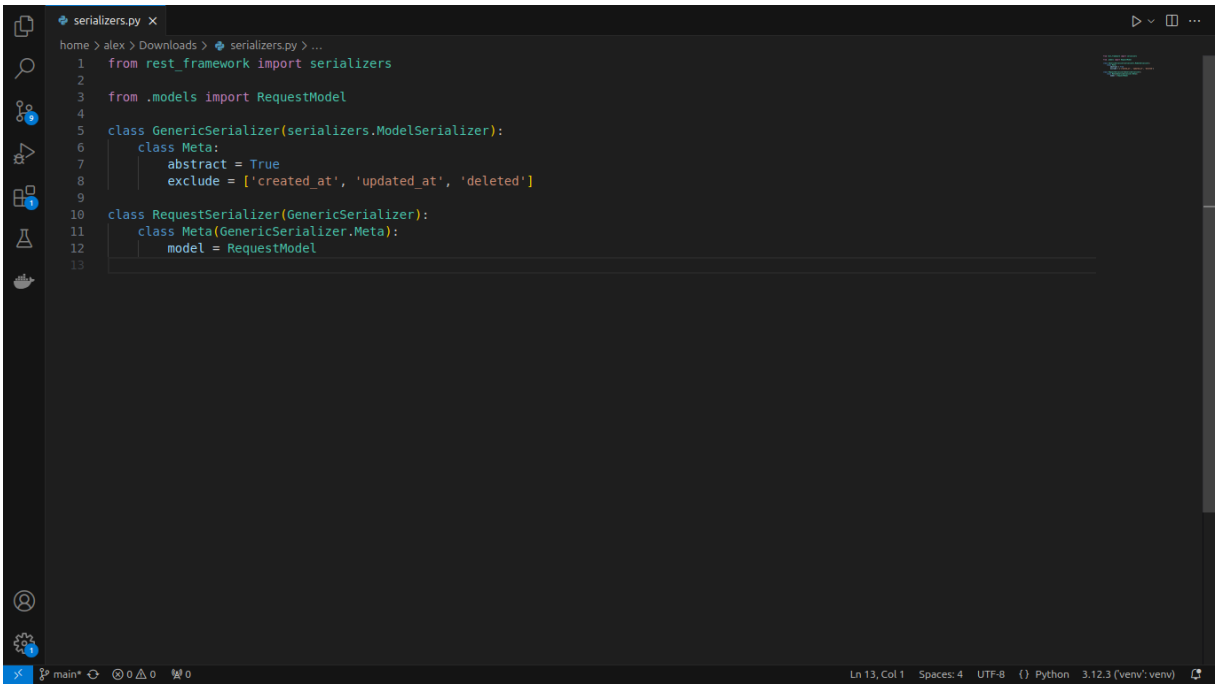
APÊNDICE B – CÓDIGOS IMPORTANTES

Figura B.1 – Modelo de RequestModel



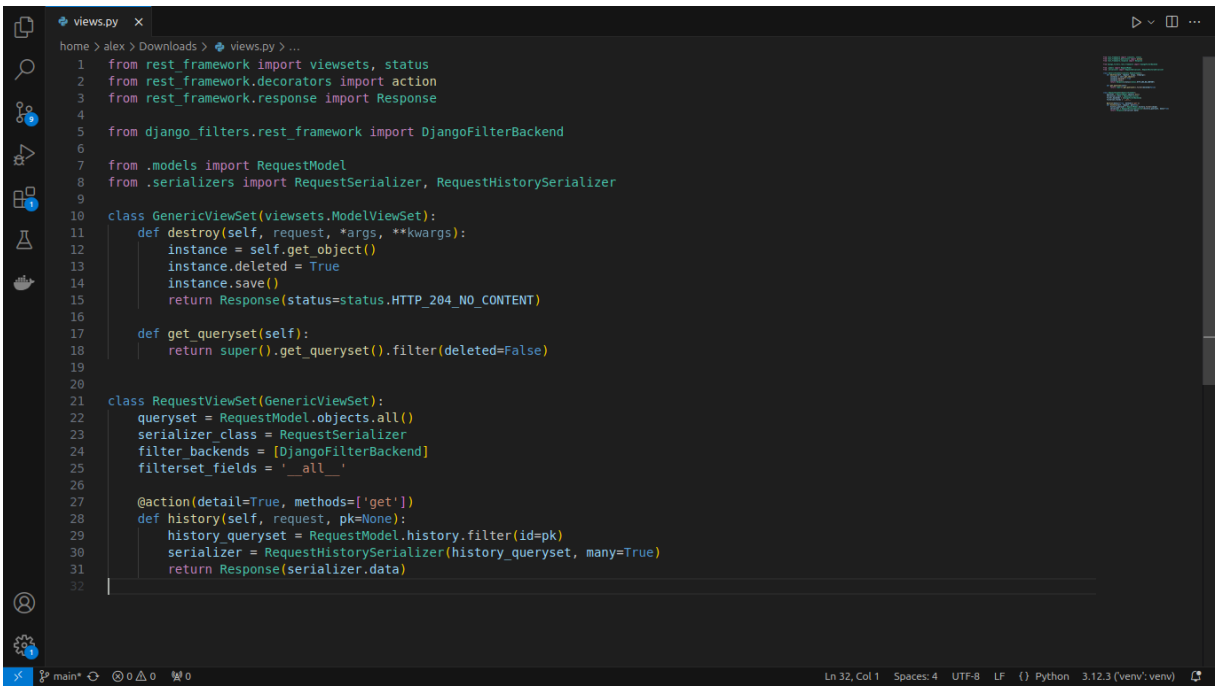
Fonte: Elaborado pelo autor (2024).

Figura B.2 – Serializador de RequestSerializer



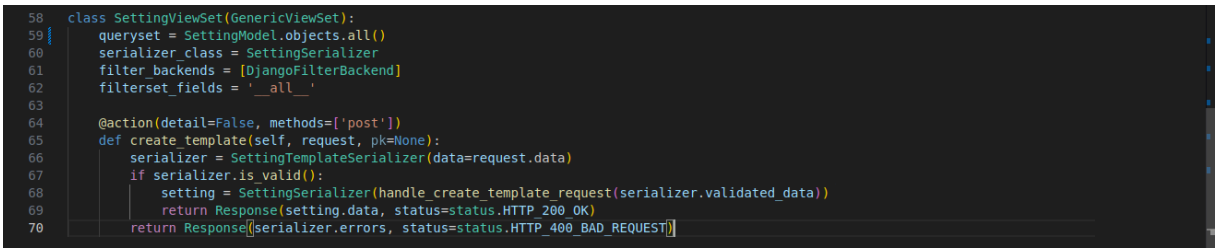
Fonte: Elaborado pelo autor (2024).

Figura B.3 – Visão de RequestViewSet



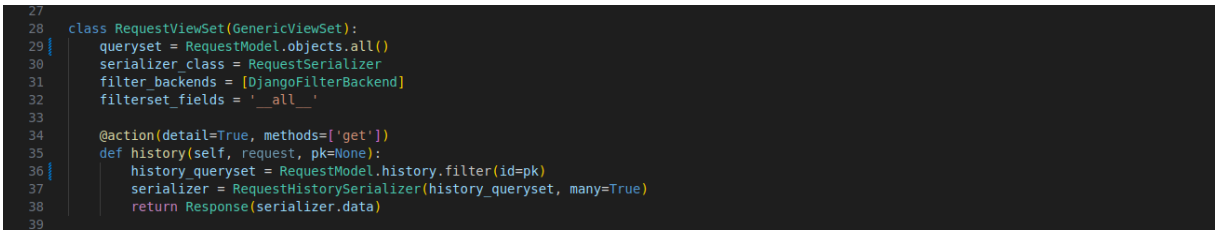
Fonte: Elaborado pelo autor (2024).

Figura B.4 – Criar processo de revisão de código (padrão)



Fonte: Elaborado pelo autor (2024).

Figura B.5 – Obter histórico de eventos



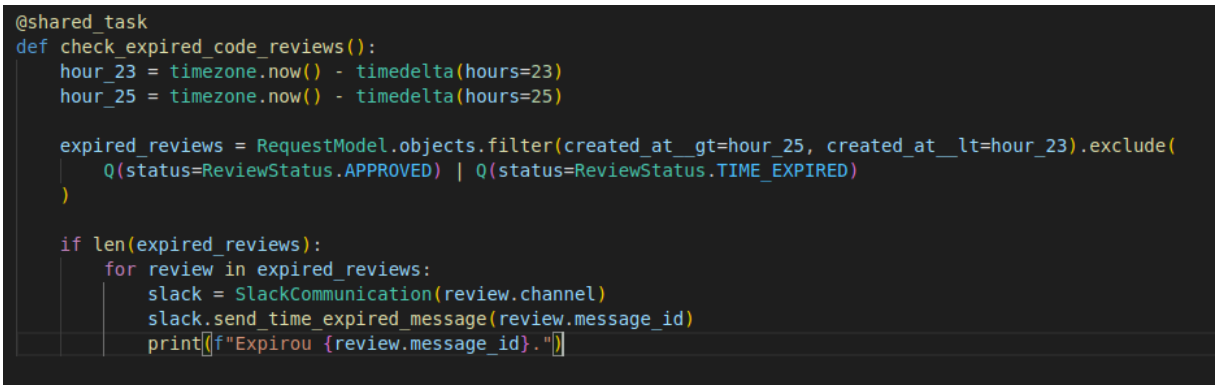
Fonte: Elaborado pelo autor (2024).

Figura B.6 – Configuração de beat

```
CELERY_BEAT_SCHEDULE = {
    'check_expired_code_reviews': {
        'task': 'review_actions.tasks.check_expired_code_reviews',
        'schedule': crontab(minute=0, hour='*'),
    },
}
```

Fonte: Elaborado pelo autor (2024).

Figura B.7– Verificar solicitações expiradas (função)



```
@shared_task
def check_expired_code_reviews():
    hour_23 = timezone.now() - timedelta(hours=23)
    hour_25 = timezone.now() - timedelta(hours=25)

    expired_reviews = RequestModel.objects.filter(created_at__gt=hour_25, created_at__lt=hour_23).exclude(
        Q(status=ReviewStatus.APPROVED) | Q(status=ReviewStatus.TIME_EXPIRED)
    )

    if len(expired_reviews):
        for review in expired_reviews:
            slack = SlackCommunication(review.channel)
            slack.send_time_expired_message(review.message_id)
            print(f"Expirou {review.message_id}.")
```

Fonte: Elaborado pelo autor (2024).

APÊNDICE C – MANUAL DE IMPLANTAÇÃO DO SMART REVIEW

Manual de implantação da automação do processo de comunicação de revisão de código contemporâneo aplicado na equipe front-end do SmartRetail.

I. Introdução

- **Objetivo**: Este manual fornece as etapas detalhadas para implantar a automação do processo de comunicação de revisão de código contemporâneo no ambiente de produção.
- **Público-alvo**: Equipe de Tecnologia da Informação com perfil, preferencialmente, de DevSecOps.

II. Visão Geral

- **Descrição**: Smart Review trata-se de uma solução com propósito de integrar duas ferramentas: controle de versão e comunicação de equipes. Ela proporciona automatizar a comunicação de etapas referente ao processo de revisão de código contemporâneo.
- **Pré-requisitos**:
  - *Hardware*:

Recurso	**Recurso**
Memória RAM	0.2 GB
Armazenamento	1 GB

  - *Software*:

Software	**Requisito Mínimo**
Distribuição Linux	Docker, versão 27.3.1. Postman, versão 11.18.
PostgreSQL	1 Instância, versão 17.
Redis	1 instância, versão 7.
GitLab	1 instância.
Slack	1 instância.

  - *Rede:*

Recurso	**Acesso**	**Descrição**

5432	Interno	Porta para acesso ao banco de dados PostgreSQL.
6379	Interno	Porta para acesso ao banco de dados em memória Redis.
8000	Externo	Porta para acesso API de Smart Review.
8001	Interno	Porta para acesso ao worker (celery).
8002	Interno	Porta para acesso ao beat (celery).

III. Etapas de Implantação

A. Instalação da solução

1. Criar Slack App (aplicativo/robot)

Etapa	Tarefa
01	Acessar https://api.slack.com/.
02	Acessar **Your Apps**.
03	Clicar no botão nomeado como **Create New App**.
04	Selecionar opção de **From scratch**.
05	Preencher os campos de **App Name e Pick a workspace to develop your app in**. ● O campo de **App Name** refere-se ao nome do aplicativo/robot. Recomenda **Hermes** (deus da comunicação na mitologia grega). ● O campo de **Pick a workspace to develop your app in**, por sua vez, deve ser escolhido o **workspace** que o aplicativo/robot terá acesso.
06	Clicar no botão **Create App**.
07	Acessar o menu lateral e selecionar **OAuth & Permissions**.
08	Ir para seção de **Bot Token Scopes**, em **Scopes**.
09	Atribuir as seguintes permissões: ● *channels:read* ● *chat:write* ● *groups:read* ● *im:read* ● *mpim:read* ● *reactions:write* ● *users:read*
10	Ir para seção de **OAuth Tokens**.
11	Clicar no botão de **Install to [Workspace Name]**.

12	Permitir acesso ao **[Workspace Name]**.
13	Guarda **Bot User OAuth Token**.

2. Adicionar aplicativo/robot no canal de comunicação

Etapa	Tarefa
01	Acessar **workspace** no Slack.
02	Criar **canal público** para comunicação de revisão de código.
03	Acessar modal de **todos os membros do canal**.
04	Acessar aba de **integrações**.
05	Clicar no botão de **Adicionar apps**.
06	Adicionar aplicativo/robot (**Hermes**).

3. Instalar Smart Review

Etapa	Tarefa
01	Organizar as seguintes variáveis de ambientes: ● **DEBUG** (False) ● **SECRET_KEY** (django-insecure...sa3fia9p7zc) ● **SLACK_API_TOKEN** (xoxb--63aRR...Qwpme3HUk) ● **CELERY_BROKER_URL** (redis://0.0.0.0:6379/0) ● **DB_NAME** (codereview) ● **DB_USER** (user) ● **DB_PASSWORD** (asdf123$) ● **DB_HOST** (127.0.0.1) ● **DB_PORT** (5432)
02	Instanciar imagem docker de **SmartReviewAPI**.
03	Acessar container de **SmartReviewAPI**.
04	Criar arquivo de migração de modelos: ● *python manage.py makemigrations*
05	Aplicar migração de modelo: ● *python manage.py migrate*
06	Instanciar imagem docker de **SmartReviewWorker**.
07	Instanciar imagem docker de **SmartReviewBeat**.

4. Configurar projeto no GitLab

Etapa	Tarefa
01	Acessar ambiente de configuração do projeto alvo.
02	Acessar funcionalidade de **Webhooks**.

03	Clica no botão de **Add new webhook**.
04	Preencher formulário. A priori, só necessita-se preencher os seguintes campos: ● **URL** (127.0.0.1:8000/slack/hook) ● **Show full URL** (deixar selecionado) ● **Secret token** ○ Abordagem de geração fica livre, uma forma de fazer isso seria utilizando o comando de ***openssl rand -hex 24***. ● **Trigger** ○ **Merge request events** (deixar selecionado).
05	Clicar no botão de **Add webhook**.
06	Acessar funcionalidade de **labels**.
07	Clicar no botão de **New label**.
08	Preencher o campo de título com **code review** e clicar no botão de **Create label**.
09	Clicar no botão de **New label**.
10	Preencher o campo de título com **resolve** e clicar no botão de **Create label**.

5. Configurar etapas do processo de comunicação de revisão de código contemporâneo

Etapa	**Tarefa**
01	Abrir Postman.
02	Importar **Smart Review.postman_collection.json**.
03	Executar requisição **TOKEN**, com os devidos ajustes em: ● Headers: ○ **username** ○ **password**
04	Executar requisição **TEMPLATE**, com os devidos ajustes em: ● Headers: ○ **Authorization** (utilizar token gerado); ● Body: ○ **name** ○ **project_url** ○ **access_token** ○ **channel**

B. Instalação da solução

1. Verificar segurança de SmartReviewAPI

Etapa	**Tarefa**
01	Acessar a raíz de **SmartReviewAPI** sem realizar autenticação.

02	Constatar presença de **HTTP 401 Unauthorized**.

2. Verificar operação de SmartReviewWorker

Etapa	Tarefa
01	Acessar logs do container de **SmartReviewWorker**. ● Ex.: *docker logs [container name]*
02	Verificar se o status de operação é **ready**.

3. Verificar operação de SmartReviewBeat

Etapa	Tarefa
01	Acessar logs do container de SmartReviewBeat. ● Ex.: *docker logs [container name]*
02	Verificar se o status de operação é **starting**.

4. Verificar configurações de projeto

Etapa	Tarefa
01	Acessar a área de autenticação de Smart Review API. ● Ex.: http://127.0.0.1:8000/admin
02	Realizar autenticação, isto é, informar **Username** e **Password**, além de clicar no botão de **Log in**.
03	Acessar a raíz de **Smart Review API**.
04	Acessar recurso de **rules** e constatar presença de nove objetos, a saber: ● Regra 01: ○ *id = 1* ○ *json_path = object_attributes.action* ○ *condition_type = equals* ○ *deny_condition = false* ○ *value = open* ● Regra 02: ○ *id = 2* ○ *json_path = labels.[].title* ○ *condition_type = contains* ○ *deny_condition = false* ○ *value = code review* ● Regra 03: ○ id = 3 ○ json_path = object_attributes.action ○ condition_type = equals ○ deny_condition = false ○ value = update ● Regra 04: ○ *id = 4* ○ *json_path = changes.reviewers* ○ *condition_type = exists* ○ *deny_condition = false*

	○ *value = null* ● Regra 05: ○ *id = 5* ○ *json_path = changes.labels.previous.[].title* ○ *condition_type = contains* ○ *deny_condition = true* ○ *value = resolve* ● Regra 06: ○ *id = 6* ○ *json_path = changes.labels.current.[].title* ○ *condition_type = contains* ○ *deny_condition = false* ○ *value = resolve* ● Regra 07: ○ i*d = 7* ○ *json_path = changes.labels.previous.[].title* ○ *condition_type = contains* ○ *deny_condition = false* ○ *value = resolve* ● Regra 08: ○ *id = 8* ○ *json_path = changes.labels.current.[].title* ○ *condition_type = contains* ○ *deny_condition = true* ○ *value = resolve* ● Regra 09: ○ *id = 9* ○ *json_path = object_attributes.action* ○ *condition_type = equals* ○ *deny_condition = false* ○ *value = approved*
05	Acessar recurso de **data-paths** e constatar presença um objeto, a saber: ● Data Path: ○ i*d = 1* ○ *json_path_request_id = object_attributes.iid* ○ *json_path_title = object_attributes.title* ○ *json_path_author = user.username* ○ *json_path_reviewers = reviewers.[].username*
06	Acessar recurso de **stages** e constatar presença cinco objetos, a saber: ● Stage 01 (solicitação para revisão de código): ○ *id = 1* ○ *rules = 1, 2* ○ *status = pending* ● Stage 02 (atribuição de revisores): ○ *id = 2* ○ *rules = 3, 4* ○ *status = assignee* ● Stage 03 (recusa de solicitação): ○ i*d = 3* ○ *rules = 3, 5, 6* ○ *status = rejected* ● Stage 04 (realização de ajustes no código): ○ *id = 4* ○ *rules = 3, 7, 8* ○ *status = adjusted*

	● Stage 05 (aprovação de solicitação): ○ *id = 5* ○ *rules = 9* ○ *status = approved*
07	Acessar recurso de settings e constatar presença de um objeto, a saber: ● Setting 01: ○ i*d = 1* ○ *stages = 1, 2, 3, 4, 5* ○ *name = [definido pelo cliente]* ○ *project_url = [definido pelo cliente]* ○ *access_token = [definido pelo cliente]* ○ *channel = [definido pelo cliente]* ○ *enable = true* ○ *data_path = 1*

5. Verificar integração de Smart Review com Slack

Etapa	**Tarefa**
01	Abrir Postman.
02	Importar **Smart Review.postman_collection.json**.
03	Executar requisição **PENDING** (adequar a URL para o contexto de uso). Constatar recebimento de notificação no Slack.
04	Executar requisição **ASSIGNEE** (adequar a URL para o contexto de uso). Constatar recebimento de notificação no Slack.
05	Executar requisição **REJECTED** (adequar a URL para o contexto de uso). Constatar recebimento de notificação no Slack.
06	Executar requisição **ADJUSTED** (adequar a URL para o contexto de uso). Constatar recebimento de notificação no Slack.
07	Executar requisição **APPROVED** (adequar a URL para o contexto de uso). Constatar recebimento de notificação no Slack.

6. Verificar execução de tarefa para notificar sobre expiração de solicitação

Etapa	**Tarefa**
01	Abrir Postman.
02	Importar **Smart Review.postman_collection.json**.
03	Executar requisição **EXPIRED** (adequar a URL para o contexto de uso).
04	Aguardar 24 horas. Depois disso, constatar recebimento de

	notificação no Slack.

IV. Controle de Acessos e Permissões

A. Configuração de Segurança

1. Cadastrar super usuário para acessar Smart Review API

Etapa	Tarefa
01	Acesse a seção de **project members** no **GitLab**.
02	Recuperar o nome de usuário (GitLab) do desenvolvedor alvo.
03	Acessar serviço de **slack/users** em **Smart Review API** ● Ex.: http://127.0.0.1:8000/slack/users
04	Recuperar o nome de usuário (Slack) do desenvolvedor alvo.
05	Acessar o recurso de **users** em **Smart Review API**.
06	Preencher o formulário com **Git username** e **Chat username**, depois submeter.

APÊNDICE D – QUESTIONÁRIO

SEÇÃO DE IDENTIFICAÇÃO

1) Perfil?
    a) Bolsista;
    b) Pesquisador Convidado;
    c) Consolidação das Leis do Trabalho (CLT).
2) Participação no SmartRetail?
    a) Até 6 meses;
    b) Entre 6 meses e 1 ano;
    c) Entre 1 ano e 2 anos;
    d) Entre 2 anos e 3 anos;
    e) Mais de 3 anos.

SEÇÃO DE REVISÃO DE CÓDIGO

3) Em uma escala de 1 a 5, como você avalia o seu nível de experiência com revisão de código? (1 = Nenhuma experiência, 5 = Muita experiência)?
    a) 1 (nenhuma experiência);
    b) 2;
    c) 3;
    d) 4;
    e) 5 (muita experiência).
4) Na hipótese de surgir uma solicitação de revisão de código, qual o seu nível de confiança para ser o revisor?
    a) 1 (pouco confiante);
    b) 2;
    c) 3;
    d) 4;
    e) 5 (muito confiante).
5) Você prefere que o processo de revisão de código tenha um deadline definido ou prefere que ele seja realizado sem um prazo fixo?
    a) Prefiro que o processo tenha um deadline definido;
    b) Prefiro que o processo seja realizado sem um prazo fixo;
    c) Depende da complexidade da revisão;
    d) Não tenho uma preferência clara.

SEÇÃO DE PROCESSO DE COMUNICAÇÃO

6) Quanto ao tempo dedicado ao processo de revisão de código, qual abordagem de comunicação você considera mais eficiente?
    a) Comunicação manual (escrita pelos envolvidos);
    b) Comunicação automatizada;
    c) Ambas são igualmente eficientes.
7) A automação reduz o esforço necessário para registrar comentários repetitivos ou recorrentes?
    a) Sim, significativamente;
    b) Sim, mas de forma limitada;
    c) Não, o esforço é o mesmo;
    d) Não, manual é mais eficiente.
8) As mensagens geradas automaticamente são mais claras e objetivas que as escritas manualmente?
    a) Sim, sempre;
    b) Sim, na maioria das vezes;
    c) Não, são igualmente claras;
    d) Não, manual é mais claro.
9) O *feedback* automatizado cobre todos os pontos necessários de uma revisão?
    a) Sim, sempre;
    b) Frequentemente, mas nem sempre;
    c) Ocasionalmente, precisa de complementação manual;
    d) Não, geralmente falta algo importante.
10) Qual método é mais eficaz para garantir que os comentários sigam os padrões e diretrizes definidos pela equipe?
    a) Automação;
    b) Manual;
    c) Ambos são equivalentes.
11) A automação oferece mais consistência nos comentários em comparação ao método manual?
    a) Sim, sempre;
    b) Sim, frequentemente;
    c) Não, ambos têm a mesma consistência;
    d) Não, manual é mais consistente.
12) A automação facilita a colaboração entre os revisores e o autor do código?
    a) Sim, significativamente;

b) Sim, de forma limitada;
c) Não, a comunicação manual é mais eficiente;
d) Não vejo diferença.

13) A automação reduz mal-entendidos ou ambiguidades na comunicação em comparação ao método manual?
a) Sim, sempre;
b) Sim, na maioria das vezes;
c) Não, o manual é mais claro;
d) Não vejo diferença.

14) Você acredita que a automação torna o processo de revisão de código mais satisfatório?
a) Sim;
b) Não;
c) Talvez.

15) O *feedback* gerado automaticamente ajuda a poupar tempo sem comprometer a qualidade?
a) Sim;
b) Não;
c) Talvez.

16) Na sua opinião, qual abordagem é mais eficiente para o estabelecimento de deadlines no processo de revisão de código?
a) Automação;
b) Autonomia dos participantes;
c) Ambas são igualmente eficientes;
d) Nenhuma das opções é eficiente.

17) Comentários?

18) Sugestões?