Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Automatização do fluxo de submissões de patches para o kernel Linux através do kworkflow

João Guilherme Barbosa de Souza

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: David de Barros Tadokoro
Cossupervisor: Paulo Roberto Miranda Meirelles

São Paulo
2025

Resumo

João Guilherme Barbosa de Souza. **Automatização do fluxo de submissões de patches para o kernel Linux através do kworkflow**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

O desenvolvimento do kernel Linux ocorre em um ambiente de grande escala e alta complexidade, baseado em um modelo de desenvolvimento perpétuo que envolve ciclos contínuos de integração, estabilização e manutenção de versões. Nesse contexto, o processo de submissão e revisão de patches é realizado majoritariamente por meio de listas de e-mail, o que impõe desafios significativos relacionados à organização das contribuições, à rastreabilidade das revisões, à sobrecarga dos mantenedores e ao alto custo de entrada para novos desenvolvedores, além de fragmentar o fluxo de trabalho ao exigir o uso de múltiplas ferramentas externas. Com o objetivo de mitigar essas limitações, este trabalho propõe a ampliação do Kworkflow (kw), uma ferramenta de software livre voltada à automação do fluxo de contribuição ao kernel Linux, por meio da introdução de mecanismos para a gestão e o acompanhamento de patches durante a fase de revisão, concretizados nos módulos kw manage contact, responsável pela organização e disponibilização de informações sobre mantenedores e revisores, e kw patch-track, voltado ao monitoramento do estado e da evolução dos patches submetidos às listas de e-mail. As soluções apresentadas integram-se às funcionalidades existentes do kw, permitindo centralizar informações provenientes das listas de e-mail, automatizar etapas recorrentes do processo de revisão e oferecer uma visão mais integrada do ciclo de contribuição, contribuindo para a redução da sobrecarga cognitiva dos desenvolvedores e para a melhoria da eficiência e da transparência do processo de desenvolvimento do kernel Linux.

Palavras-chave: kernel Linux. kworkflow. kw mange-contacts. kw patch-track. Fluxo de submissão de patches. Gerenciamento de contatos de email.

Abstract

João Guilherme Barbosa de Souza. **Automating the Linux kernel patch submission flow using kworkflow**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

The development of the Linux kernel takes place in a large-scale and highly complex environment, based on a perpetual development model that involves continuous cycles of integration, stabilization, and maintenance of released versions. In this context, the submission and review of patches are conducted primarily through mailing lists, which introduces significant challenges related to contribution organization, review traceability, maintainer workload, and the high entry barrier for new developers, as well as fragmenting the workflow by requiring the use of multiple external tools. To address these limitations, this work proposes the extension of Kworkflow (kw), a Free/Libre and Open Source Software (FLOSS) tool aimed at automating the Linux kernel contribution process, through the introduction of mechanisms for managing and tracking patches during the review phase, implemented in the kw manage contact module, which organizes and provides information about maintainers and reviewers, and the kw patch-track module, which monitors the status and evolution of patches submitted to mailing lists. The proposed solutions integrate with existing kw functionalities, enabling the centralization of information from mailing lists, the automation of recurring review tasks, and the provision of a more integrated view of the contribution lifecycle, thereby contributing to reduced developer cognitive load and to improvements in the efficiency and transparency of the Linux kernel development process.

Keywords: kernel Linux. kworkflow. kw mange-contacts. kw patch-track. Patches submission workflow. email contacts managment.

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Introdução

Computadores são parte central da vida em sociedade, servindo como pilar das relações modernas. Um *Sistema Operacional* (SO) é um conjunto de softwares que, de acordo com Tanenbaum e Bos (2023), realiza duas funções principais: **abstração** e **gerenciamento** do hardware que compõe as máquinas. Isto possibilita a interação entre ser humano e computador, tanto para a sua programação quanto para o seu uso, de forma simplificada e eficiente. Em sua composição, os SOs são divididos em diversos componentes específicos, dentre os quais o *kernel* (em português, *núcleo*) é considerado a parte central. Este fato decorre principalmente das responsabilidades atribuídas a ele, que incluem a gestão da alocação de recursos entre programas em execução, o escalonamento de atividades críticas e o gerenciamento da comunicação entre periféricos (mouse, teclado, placas de vídeo dedicadas, entre outros) e o sistema. Absorver estas e inúmeras outras complexidades permite, por exemplo, que um simples programa colete input do usuário e o imprima na tela sem que o programador se preocupe em **como** o computador faz isto. Por baixo dos panos, o kernel processa ações do usuário por meio dos dispositivos de entrada, coordena o uso do processador, memória e outros recursos internos e, por fim, apresenta os resultados de forma significativa ao usuário, através dos dispositivos de saída. Nesta ilustração, se faz clara a responsabilidade do kernel de abstrair as especificidades de como realizar tais operações provendo uma *interface de chamada de sistema* ao mesmo tempo que gerencia os recursos sendo usados (Silberschatz *et al.*, 2018).

Dentre as diversas implementações de kernel existentes, o Linux, criado por Linus Torvalds e lançado em 1991, destaca-se como um dos mais relevantes. Mesmo que não advogando explicitamente pelo movimento de software livre, Torvalds começou o projeto Linux como uma alternativa à hegemonia dos SOs proprietários (Torvalds, 1991a; Torvalds, 1991b), como o Unix, sendo construído de forma colaborativa por uma comunidade de desenvolvedores e disponibilizando livre acesso ao seu código e documentação. O kernel Linux é atualmente o maior projeto de software livre do mundo, utilizado por grandes empresas de tecnologia e computação, com incontáveis SOs que o usam como kernel (as chamadas **distribuições Linux**), rodando em, pelo menos, aproximadamente 58%[1] de todos os servidores web. Do ponto de vista de engenharia de software, após mais de três décadas desde seu lançamento, o projeto vem tendo um aumento no número de contribuições e pessoas envolvidas em cada ciclo de desenvolvimento das versões *stable kernels* (Passos *et al.*, 2025), sem considerar outros esforços como desenvolvimento *downstream*.

[1] 31% constam como SOs desconhecidos e acredita-se que boa parte destes sejam Linux. Fonte: https://w3techs.com/technologies/details/os-unix; acessado em 9 de dezembro de 2025.

Para sustentar esse ciclo contínuo de desenvolvimento, o kernel Linux adota um modelo rigoroso descrito por Feitelson (Feitelson, 2012) como modelo de desenvolvimento perpétuo, no qual novas funcionalidades, correções e versões de produção são liberadas continuamente, ao mesmo tempo em que versões mais antigas permanecem em manutenção. Esse modelo é estruturado em três etapas principais. A primeira, denominada janela de mesclagem (do inglês, merge window), corresponde ao período em que os patches dos subsistemas e drivers já testados e validados são enviados à mainline, sob supervisão de Linus Torvalds, para integração ao kernel principal. Na prática, os contribuidores enviam patches continuamente aos subsistemas de que participam (por exemplo, IIO ou AMD-GFX), independentemente da fase do ciclo de releases. Durante a merge window, esses patches previamente testados e acumulados são submetidos à mainline, dando início ao processo formal de integração. A partir dessas integrações, é lançada uma versão inicial do novo kernel, denominada de -rc1, iniciando-se a segunda etapa, o período de estabilização, durante o qual apenas correções e melhorias incrementais são aceitas.

Por fim, ao atingir o nível de qualidade necessário, a versão final para este ciclo é oficialmente lançada, e uma equipe reduzida (conhecida como o *stable team*) passa a atuar na manutenção contínua desta versão, liberando novas correções enquanto uma nova janela de mesclagem é aberta.

Fora do ciclo de releases, embora este ocorra de forma consecutiva e estruturada, os contribuidores enviam suas contribuições continuamente para os subsistemas específicos, como IIO, AMD-GFX e outros, testando e validando localmente seus patches. Mantenedores e a comunidade se encarregam de gerir essas contribuições, aplicando revisões e testes, garantindo a integração adequada com a mainline, e é neste contexto que o processo de revisão de código se torna mais significativo, mesmo que algumas interações diretas com Linus ou entre subsistemas ocorram em casos específicos.

Dentro desse fluxo contínuo de contribuição, os patches exigem um processo de preparação que envolve diversas etapas, como o design — em que são definidas as concepções iniciais e as implementações necessárias —, a revisão — em que as contribuições são avaliadas pela comunidade e pelos mantenedores —, e a fase de mesclagem e manutenção, em que o desenvolvedor continua responsável por eventuais ajustes após a integração. Considerando a complexidade inerente a um sistema operacional, desenvolver para o kernel Linux representa um desafio significativo para a maioria dos programadores, em razão do amplo conhecimento prático e teórico exigido.

Com o intuito de reduzir parte dessas dificuldades, a comunidade desenvolveu ferramentas destinadas à automação dos fluxos de trabalho. Entre elas, destaca-se o *Kworkflow* (kw), uma ferramenta de software livre desenvolvida majoritariamente em Bash script, que tem como objetivo oferecer uma solução unificada para os diversos desafios enfrentados pelos desenvolvedores do kernel. Para isso, o kw integra e simplifica ferramentas e serviços amplamente consolidados na comunidade, como Git, o arquivo do Lore e o b4, criando soluções locais quando necessário. A ferramenta organiza-se como um hub de funcionalidades, recebendo comandos do usuário via linha de comando e redirecionando a execução para o módulo apropriado, de modo a oferecer uma interface única para todo o processo.

Apesar da ampla estrutura já existente, compreender de forma completa o fluxo de contribuição ao kernel continua sendo um desafio que o kw busca superar, permanecendo

em constante desenvolvimento pela comunidade. Além de automatizar o workflow de desenvolvimento de patches, o kw tem o objetivo de se constituir como um software científico e, para isso, precisa ser capaz de fornecer rastreabilidade das contribuições, coletar dados do processo e possibilitar sua análise empírica, tornando possível que pesquisas, principalmente em Engenharia de Software, sejam desenvolvidas tendo como base a ferramenta.

Um dos processos ainda em aberto consiste em automatizar a gestão dos patches após a submissão e antes da aprovação, período em que as contribuições passam pela revisão dos mantenedores — uma etapa particularmente complexa no modelo de contribuição por listas de e-mail adotado pelo projeto Linux.

Um dos grandes desafios no desenvolvimento de sistemas de software é coordenar o trabalho simultâneo de diversos colaboradores, o que envolve a gestão de versões, submissões e atualizações. Antes do surgimento dos sistemas de controle de versão, esse processo era realizado manualmente, com métodos como cópias redundantes e convenções de nomenclatura, o que se mostrava inconsistente e de difícil manutenção. Com a introdução dos Version Control Systems (VCS), tornou-se possível registrar o histórico das alterações e recuperar versões anteriores. A evolução desses sistemas levou ao surgimento dos modelos distribuídos, como o Git, que permitiram maior flexibilidade e paralelismo, possibilitando que cada colaborador mantivesse uma cópia local do código e realizasse integrações controladas de suas modificações.

Mesmo assim, conforme aponta Greg Kroah Hartman (2016), ferramentas como GitHub e Gerrit, embora adequadas a projetos menores, ainda apresentam limitações quando aplicadas a softwares de grande escala, como o kernel Linux. Entre os principais entraves estão o tempo elevado de revisão, a dificuldade de organização e categorização de problemas, a baixa acessibilidade das discussões internas e a sobrecarga das listas de pendências dos mantenedores. Parte dessas dificuldades é mitigada pelo uso de servidores de e-mail como meio principal de contribuição, que, embora resolvam alguns problemas de escalabilidade, introduzem outros desafios, como o alto custo de entrada para novos desenvolvedores, a rastreabilidade limitada das revisões, a sobrecarga das caixas de entrada, a possibilidade de corrupção de arquivos e a dificuldade de coletar métricas sobre o processo de desenvolvimento. Além disso, a necessidade de recorrer a ferramentas externas, como navegadores ou clientes de e-mail, fragmenta o fluxo de trabalho, afastando-se do princípio do kw de oferecer uma experiência integrada.

A dependência de sistemas de e-mail representa, portanto, uma limitação à proposta do kw de abranger o processo de desenvolvimento de forma holística. O usuário submete suas alterações por meio do kw, mas precisa recorrer a outros meios para acompanhar revisões e retornar à ferramenta para atualizar suas submissões, o que dificulta também a análise completa do fluxo de contribuição — um dos objetivos centrais do projeto.

Considerando esse cenário, este trabalho dá continuidade ao processo de melhoria contínua do kw, iniciado por iniciativas anteriores, como Simplificando o processo de contribuição para o kernel Linux (Neto, 2022) e Integrating the Kworkflow system with the Lore archives: Enhancing the Linux kernel developer interaction with mailing lists (Barros Tadokoro, 2023). A proposta aqui apresentada consiste em oferecer aprimoramentos e automatizações voltadas à gestão de patches durante o processo de revisão, integrando-se

às implementações anteriores que introduzem, respectivamente, os fluxos de envio e de consulta de patches.

Capítulo 1

Fundamentação Teórica

Este capítulo discute os fundamentos teóricos levantados para a compreensão do funcionamento e da gestão do kernel Linux. Nele, são abordados a estrutura de gestão do projeto e de outros softwares livres, o papel dos mantenedores e o fluxo de desenvolvimento por meio de *patches* e árvores de repositórios. Tais definições estabelecem a base conceitual necessária para detalhar as etapas de integração de código, a organização dos subsistemas e a dinâmica de manutenção contínua do sistema.

1.1 Software livre

No mercado computacional, dois principais modos dominam o cenário no que se refere ao desenvolvimento como software proprietário ou software livre. Em geral, define-se como software proprietário o software desenvolvido de maneira privada, em que apenas a aplicação é acessível aos usuários. Em contrapartida, o software livre fundamenta-se na garantia de acesso ao código-fonte e é definido por quatro liberdades estabelecidas pela Foundation, 2023: a liberdade de executar o programa para qualquer propósito (liberdade 0), de estudar seu funcionamento e adaptá-lo (liberdade 1), de redistribuir cópias (liberdade 2) e de distribuir versões modificadas a terceiros (liberdade 3). Esse conceito abrange também o movimento *Open Source*, que, embora apresente motivações distintas voltadas à eficiência técnica, compartilha o princípio do acesso aberto ao código para viabilizar o desenvolvimento colaborativo e a transparência do projeto

Historicamente, o mercado computacional era dominado por grandes corporações, que detinham o monopólio do processo, conhecimento e recursos necessários para o desenvolvimento do software como um todo, dificultando o ingresso de outros competidores no mercado. Nesse cenário, projetos de software livre surgem como um processo disruptivo, compartilhando o acesso a esse conhecimento, de modo a promover a colaboração e inovação na indústria (Avatavului *et al.*, 2023).

Esssa abordagens possuem também grande impacto no modo como essas ferramentas são produzidas, comparativamente, softwares proprietários são geridos por empresas, que contratam equipes fixas de funcionários para trabalhar em período integral e de maneira exclusiva no projeto. Dessa maneira, a decisão quanto às novas implementações para

o software são centradas, com o principal objetivo, em muitos casos, sendo o de ganho financeiro. Essa prática, contudo, muitas vezes implica em que o foco constante seja em implementações de novas ferramentas ao invés da melhoria do software como um todo, favorecendo com que esses softwares sejam mais propensos à erros e falhas ocasionais. Por outro lado, a existência de responsáveis legais pelo projeto fazem com que esses softwares contem em grande parte com a existência de um suporte especializado, característica valorizada no mercado corporativo.

De maneira oposta, softwares livres são desenvolvidos de maneira colaborativa, contando com a contribuição voluntária de grandes quantidades de desenvolvedores ao redor do mundo inteiro. Por conta disso, as demandas surgem de forma espontânea, muitas vezes da necessidade do próprio usuário que depende do software para usos pessoais. Essa grande quantidade de contribuidores, aliada à dependência mútua destes com o software, garante atualizações frequentes de segurança e qualidade. Como consequência, esses sistemas tendem a ser menos propensos a erros, mas não contam com um suporte dedicado na maioria dos casos.

Hoje, modelos de software livre e proprietário continuam coexistindo no mercado, sendo constantemente comparados quanto à sua efetividade observada em projetos reais. Porém, a inegável vantagem do conhecimento colaborativo e do grande volume de contribuição que softwares livres conseguem apresentar, fazem com que hoje ele esteja em grande crescente no mercado computacional, sendo o método de desenvolvimento de diversos softwares relevantes mundialmente, como no caso do kernel Linux.

1.1.1 Processo de contribuição em Software Livre

Em projetos de software livre, para que a gestão das contribuições seja possível, os projetos geralmente contam com uma equipe de *mantenedores*, que é um grupo interno de desenvolvedores que possuem responsabilidade geral pelo código principal. Desse modo, para que sejam integradas, as contribuições enviadas pelos *contribuidores* precisam passar por revisões por parte dos mantenedores para garantir que atendam aos requisitos técnicos definidos para o projeto. De acordo com Tan *et al.*, 2020, os mantenedores devem avaliar principalmente se uma contribuição é necessária, se uma implementação apresenta falhas ou se existem eventuais melhorias na forma como a solução foi feita.

Em alguns casos, principalmente devido ao crescimento dos projetos, torna-se necessário também uma divisão em componentes do sistema principal, de modo que cada parte possui seus mantenedores dedicados. Dessa maneira, para que as contribuições sejam feitas de maneira correta, elas precisam ser enviados diretamente para o responsável do subsistema que será alterado.

1.2 O Kernel Linux

Em um computador, o Sistema Operacional é a parte responsável por lidar com as interações entre o hardware e o software, permitindo, de maneira eficiente, a interação final máquina-usuário. Para que isso seja possível, o sistema operacional precisa ser dividido em diversas partes, dentre essas, o kernel, considerado o núcleo dos sistemas

operacionais. Em geral, o kernel é um programa que opera a todo momento e é responsável pelo gerenciamento dos processos do sistema, alocando recursos para outros programas em atividade conforme a necessidade e prioridade de cada um.

Dentre os muitos sistemas de kernels existentes, o kernel Linux é um projeto desenvolvido por Linus Torvalds, em 1991, como alternativa ao Unix, pioneiro no mercado de sistemas operacionais. Hoje, o kernel Linux é o maior projeto de software livre do mundo, utilizado por algumas das maiores empresas de tecnologia, software e computação no mercado, possuindo diversas distribuições e constituindo aproximadamente 57% dos websites na internet cujos sistemas operacionais puderam ser identificados.[1]

1.2.1 O Modelo de desenvolvimento do kernel Linux

Ainda que tenha sido lançado há mais de três décadas, novas versões do kernel Linux continuam sendo lançadas até hoje. Cada uma das versões do Linux é construída através da contribuição de diversos desenvolvedores ao redor do mundo, por meio da submissão de *patches*[2] de melhorias que são integrados à versão principal para o desenvolvimento de futuras versões. De acordo com Feitelson, 2012, esse desenvolvimento do kernel Linux segue um modelo de desenvolvimento perpétuo, no qual novas funcionalidades, correções e versões de produção são liberadas continuamente, havendo também a manutenção de versões mais antigas. Segundo a The Linux Kernel Documentation, 2023, esse processo divide-se em três etapas bem distintas: a janela de mesclagem, o período de estabilização e a manutenção contínua.

Janela de Mesclagem (Merge Window) Durante a primeira etapa, a maior parte das alterações será integrada à nova versão do kernel. Essas mesclagens não ocorrem de forma imediata; elas ocorrem a partir de *patches* que foram previamente preparados, testados e coletados em árvores de subsistemas ao longo de semanas ou meses. Esse trabalho prévio de organização pelos mantenedores garante que, ao abrir a janela, o código já tenha passado por um ciclo de maturação inicial. Com base nessa nova versão, o primeiro kernel RC (*Release Candidate*) será lançado, encerrando a janela de mesclagem e iniciando a próxima etapa.

Período de Estabilização Durante a segunda etapa, apenas *patches* que sirvam para correção de *bugs* deverão ser enviados e novas versões de RC serão lançadas periodicamente até que uma versão estável seja atingida. De forma objetiva, uma versão estável é atingida quando todas as regressões — erros conhecidos que haviam sido superados por versões anteriores e foram reintroduzidos durante a janela de mesclagem — são corrigidas. O foco aqui é estritamente a confiabilidade do código integrado anteriormente.

---

[1] Fonte: https://w3techs.com/technologies/details/os-linux

[2] No contexto de desenvolvimento do kernel, um *patch* é um arquivo de texto que descreve as diferenças entre duas versões do código-fonte. Essas contribuições são enviadas via e-mail para listas de discussão públicas, contendo o código alterado e uma descrição das mudanças (o *commit message*), para serem revisadas pelos mantenedores antes da integração.

Manutenção Contínua (Stable Kernels) Contudo, dado o tempo limitado em que as etapas precisam ocorrer, eliminar todas as regressões das versões estáveis nem sempre é um desafio que pode ser atingido plenamente antes do lançamento. Por conta desse fato, após a criação da versão estável, o projeto entra na fase dos *stable kernels*. Nesta terceira etapa, uma equipe de desenvolvedores é designada para a manutenção contínua, lançando novas atualizações ocasionais (como as revisões pontuais da versão principal) com correções críticas para essa versão por um período de tempo, enquanto a janela de mesclagem se reinicia para a nova versão.

1.2.2 Contribuindo para o Kernel Linux

Assim como outros softwares livres, o kernel Linux também apresenta uma divisão lógica com base no seu conjuntos de subsistemas, como, por exemplo, os sistema de rede, gerenciamento de memória, dispositivos de vídeo, etc. Dentro desses subsistemas, cada mantenedor responsável administra um repositório de fontes do kernel, gerindo os *patches* enviados ao seu subsistema. Ainda segundo a documentação oficial (The Linux Kernel Documentation, 2023), eventualmente, esses subsistemas podem ser identificados de modo que um subsistema principal seja constituído por subsistemas menores, como, por exemplo, o subsistema de rede, que agrega também os repositórios dedicadas a drivers de dispositivos de rede cabeadas e de redes sem fio. Desse modo, além dos *patches* recebidos diretamente por contribuidores, os mantenedores podem receber também receber *patches* já aprovados por outros mantenedores, formando uma cadeia de confiança até que os *patches* cheguem a serem integrados.

Para gerir esse modelo de contribuição, o código do kernel Linux é organizado em um modelo de repositórios separados, conhecidos como *árvores do kernel* (*kernel trees*), contendo versões específicas do projeto, com suas respectivas finalidades e responsáveis. A princípio, cada subsistema do kernel possui uma árvore específica, gerida pelos mantenedores responsáveis pela seção do projeto, nas quais novas contribuições aceitas serão acumuladas e testadas previamente. Posteriormente, para que as novas versões sejam construídas, utiliza-se como base a árvore *mainline* — o repositório central e oficial do kernel, administrado diretamente por Linus Torvalds, que contém o código da versão em desenvolvimento e das futuras *releases*. Durante a janela de mesclagem, são construídas as árvores *-next*, que funcionam como árvores de integração ramificadas da *mainline* para reunir as novas submissões dispersas nas diversas árvores de subsistemas. A partir dela, após a janela de mesclagem e durante o período de estabilização, serão criadas as árvores de estabilização, nas quais serão organizadas as correções da nova versão até que, por fim, essas alterações venham a ser consolidadas na árvore *mainline* definitiva.

Em paralelo à execução das etapas formais do ciclo, o desenvolvimento nos subsistemas específicos, a exemplo do IIO e AMD-GFX, ocorre de forma ininterrupta. Antes da submissão aos mantenedores, os contribuidores realizam a validação local dos *patches*, iniciando um fluxo de trabalho que precede a janela de mesclagem. A gestão dessas contribuições pela comunidade e pelos mantenedores envolve ciclos constantes de testes e revisões, assegurando que o código esteja estabilizado para a integração na *mainline*. (The Linux Kernel Documentation, 2023).

Os principais estágios que um *patch* deve passar, são:

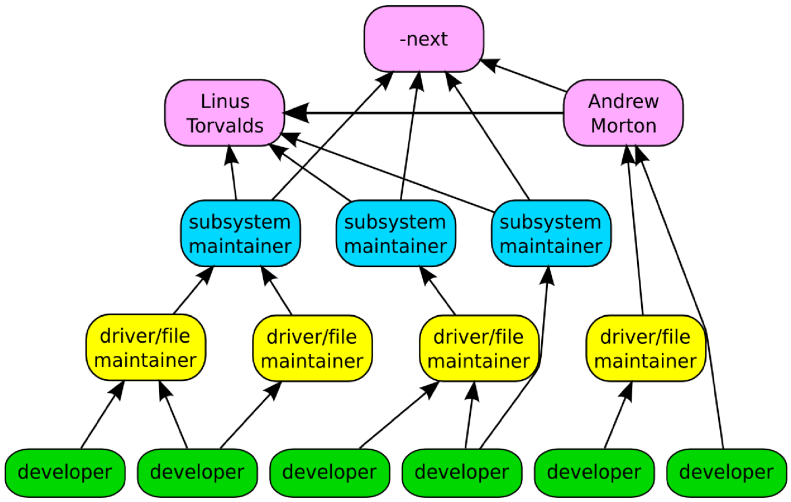

Figura 1.1: *O processo de um patch do kernel (Kroah-Hartman, 2018)*

1. **Design**: Nesta etapa, serão levantados os requisitos do *patch* e a forma com que serão atingidos, ou seja, a identificação dos seus objetivos e as necessidades técnicas que devem constar nessa implementação.
2. **Revisão antecipada**: Publicação dos *patches* na lista de discussão relevante para que desenvolvedores possam responder com comentários e ajudar a revelar quaisquer problemas iniciais.
3. **Revisão mais ampla**: Antes que o *patch* seja considerado para inclusão na versão principal, ele deve ser aceito por um mantenedor de subsistema, que o incluirá nas árvores *-next*. Com essa etapa, revisões mais elaboradas e possíveis problemas de integração com outras implementações poderão ser verificados.
4. **Mesclagem e manutenção de longo prazo**: Ainda que o *patch* possa ser mesclado e chegar efetivamente à versão estável do kernel, futuros problemas podem vir a aparecer durante essas fases, dessa forma, o desenvolvedor original deve continuar a assumir a responsabilidade da manutenção do código no futuro.

Capítulo 2

Kernel Workflow

Atualmente, considerada toda a complexidade envolvida em um sistema operacional, desenvolver para o kernel Linux pode ser uma tarefa extremamente desafiadora para a maioria dos desenvolvedores. Além do conhecimento teórico sobre a arquitetura do sistema, diversos conhecimentos práticos precisam ser empregados antes que qualquer contribuição possa, de fato, ser iniciada. A exemplo, por se tratar do núcleo de um sistema operacional, o domínio de ferramentas e técnicas para criação de ambientes seguros de teste, como o uso de máquinas virtuais e ambientes isolados, se fazem necessários para validar alterações sem comprometer o sistema principal do desenvolvedor. Além disso, é preciso saber construir e implantar esses ambientes — envolvendo etapas de build e deploy — de modo a reproduzir com precisão o comportamento do kernel em diferentes cenários e arquiteturas dos computadores.

Tendo conhecimento desses fatos, diversas ferramentas são construídas pela comunidade para automatização desses fluxos. Dentre elas, o *Kworkflow* (kw)[1] é uma ferramenta de software livre, desenvolvida principalmente em Bash, que surge com o objetivo de apresentar uma solução unificada para as diversas dificuldades que desenvolvedores do kernel podem encontrar. Além de automatizar o workflow de desenvolvimento de patches, um dos objetivos do kw é se constituir também como um software científico, permitindo que pesquisadores de Engenharia de Software e áreas correlatas estudem o desenvolvimento do kernel Linux de forma empírica e próxima da prática.

Para que o kw cumpra esse papel científico, é necessário que ele forneça mecanismos de coleta e registro de dados do processo de contribuição, garanta rastreabilidade das alterações e interações entre desenvolvedores, possibilite a reconstrução de cenários de revisão de código e permita a análise estatística ou qualitativa dos fluxos de trabalho. Essas funcionalidades tornam possível investigar padrões de colaboração, eficiência de processos, dificuldades enfrentadas pelos contribuidores e comportamento de manutenção de software de larga escala, transformando o kw em uma ferramenta de pesquisa robusta, além de um utilitário prático para o desenvolvimento de patches. Promovendo assim um ambiente de desenvolvimento mais simples e rápido, reduzindo a carga de conhecimento prévio

[1] O repositório do projeto está disponível em https://github.com/kworkflow/kworkflow e o seu sítio oficial em https://kworkflow.org/.

necessária para novos desenvolvedores e consolidando um meio pelo qual seja possível medir de forma precisa o ciclo de contribuição. Possibilitando ainda que novas soluções possam ser planejadas e que o impacto real das ferramentas já empregadas seja mensurado.

2.1 Arquitetura

Para que o software seja capaz de agrupar tantas ferramentas, o kw segue uma organização estrutural específica em 5 partes:

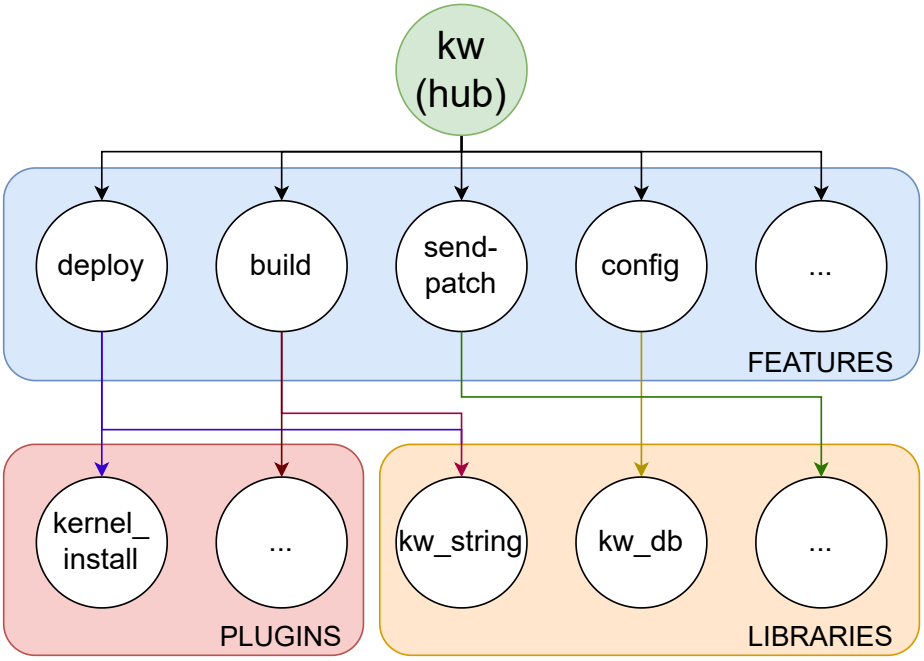

Figura 2.1: *Arquitetura conceitual do kw Fonte:* Tadokoro *et al., 2025*

1. **Hub**: Para permitir que todas as ferramentas do kw sejam oferecidas através de uma interface única, o software utiliza-se de um arquivo central, o *kw.sh*. Esse arquivo atua como um hub — representado pela cor verde na Figura 2.1 — sendo o responsável por receber os comandos iniciais dos usuários no terminal e redirecionar a execução para a ferramenta especificada.
2. **Componentes**: Cada ferramenta do kw possui um arquivo principal que contém o processamento central do comando. Esta camada corresponde à seção azul denominada *FEATURES* na Figura 2.1, incluindo a lista de comandos específicos para cada funcionalidade e uma seção de ajuda para orientação dos usuários.
3. **Bibliotecas**: O kw utiliza um esquema de bibliotecas para permitir o compartilhamento de código genérico entre diferentes ferramentas. Conforme ilustrado na seção amarela (*LIBRARIES*) da Figura 2.1, essas implementações são agrupadas por similaridade de contexto, como manipulação de textos, operações em banco de dados ou tratamento de data e hora.
4. **Plugins**: Códigos dependentes de contextos externos ou com alta volatilidade de desenvolvimento são isolados em arquivos específicos. Estes plugins, identificados na

cor vermelha (*PLUGINS*) na Figura 2.1, permitem que o código principal permaneça estável, aproveitando os métodos declarados independentemente das alterações nas implementações internas desses plugins.

5. **Documentação**: Para manter o registro das implementações e informações necessárias para colaboradores, o kw mantém um sistema de documentação, que também é utilizado para a construção do blog da ferramenta.

2.2 Funcionalidades

Para compor o seu ferramental e permitir um ambiente holístico, o kw utiliza-se da integração e simplificação de automações consolidadas na comunidade, como o Git, Lore, b4, e outros, desenvolvendo soluções locais quando necessário. De acordo com Barros Tadokoro, 2023, as automações desenvolvidas para o kw dividem-se em dois tipos. As implementações práticas afetam diretamente o desenvolvimento do kernel, como o *kw build* e o *kw deploy*, utilizados para a criação e aplicação da imagem com as alterações do desenvolvedor. Adicionalmente, existem as ferramentas indiretas, que impactam o fluxo de trabalho de forma abrangente, como o *kw send-patch* e o *kw-patch hub*, voltados, respectivamente, à submissão e consulta de *patches* no *lore*.

Por se tratar de uma ferramenta de terminal, os comandos do kw precisam ser invocados de forma escrita pelo usuário, seguindo, a seguinte estrutura: *kw <comando> <parâmetros>*. Até o momento, as principais funcionalidades existentes na ferramenta, são:

Apesar da grande estrutura, compreender de forma completa o fluxo do desenvolvedor do kernel ainda é um desafio que o kw busca superar, estando em constante processo de desenvolvimento por parte da sua comunidade. Um dos processos em abertos, é o de conseguir automatizar o fluxo de gestão dos patches após a submissão e antes da aprovação, na qual os patches passam pelo processo de revisão por parte dos mantenedores, que se torna um desafio em particular durante a contribuição para o kernel Linux dado o seu modelo de contribuição não trivial por listas de email.

2.3 O problema da contribuição no desenvolvimento de software livre

Um grande desafio encontrado durante a construção de sistemas de software é a dificuldade de conciliar o trabalho simultâneo dos diversos colaboradores, o que envolve a capacidade de coordenar as diferentes versões do projeto e as inúmeras submissões de alteração para a versão principal. Antes do advento dos sistemas de controle de versão, os programadores dependiam de métodos manuais para gerenciar suas modificações de código. Eles costumavam fazer backups regulares de seus arquivos de código ou adotar convenções de nomenclatura para distinguir entre as várias versões. Esse processo era bastante inconsistente e difícil de gerenciar, especialmente quando alguns desenvolvedores estavam trabalhando no mesmo projeto (Devineni, 2020).

Gerenciar as versões de um software se torna um problema ainda maior dependendo do

Command	Category	Description
`build`	kernel build/deploy	Build kernel and modules
`deploy`	kernel build/deploy	Deploy kernel and modules
`kernelconfig-manager`	kernel build/deploy	Manage `.config` files
`env`	kernel build/deploy	Manage different environments for same kernel tree
`bd`	kernel build/deploy	Build and Deploy kernel and modules
`send-patch`	patch submission	Send patches via email
`maintainers`	patch submission	`get_maintainers.pl` wrapper
`codestyle`	patch submission	`checkpatch.pl` wrapper
`remote`	target machine	Manage machines in the network
`vm`	target machine	QEMU wrapper
`ssh`	target machine	`ssh` wrapper
`device`	target machine	Show hardware information
`debug`	code inspection	Linux debug utilities
`explore`	code inspection	Explore string patterns
`diff`	code inspection	Diff files
`init`	kw management	Initialize kw kernel tree
`config`	kw management	Set kw configs
`self-update`	kw management	Self-update mechanism
`backup`	kw management	Save and restore kw data
`clear-cache`	kw management	Clear kw cache
`patch-hub`	misc	TUI for patches from lore.kernel.org
`drm`	misc	DRM specific utilities
`pomodoro`	misc	Pomodoro technique
`report`	misc	Show usage statistics

Tabela 2.1: *comandos do kw. Fonte: Reproduzido de Tadokoro et al., 2025, p. 4*

tamanho total do software, do número de contribuidores e da quantidade de contribuições sendo realizadas nele de maneira simultânea. No kernel, por exemplo, a versão 6.13, lançada em 19/01/2025, contou com mais de 206 contribuições por dia por parte de 2085 colaboradores, resultando em um código fonte final com mais de 39 milhões de linhas (Kroah-Hartman, 2025). Segundo a tendência, esses números devem seguir aumentando de forma constante conforme novas versões forem sendo desenvolvidas.

Buscando superar parte dessas dificuldades e melhorar o processo colaborativo de desenvolvimento de software, foram desenvolvidos os sistemas de controle de versão. Ainda segundo Devineni, 2020, os primeiros Version Control Systems (VCS), permitiam que os desenvolvedores mantivessem um histórico das alterações realizadas nos arquivos, o que facilitava a reversão de mudanças e oferecia visibilidade sobre a evolução do código. No entanto, o potencial colaborativo ainda era limitado, exigindo muitos acordos e gestões manuais por parte dos colaboradores.

Como segunda opção, surgem os *Concurrent Versions System (CVS)*, baseados em um modelo de repositório central. Nele, os desenvolvedores podiam obter os arquivos, aplicar suas modificações e submetê-las novamente ao repositório. Esse modelo contribuiu para maior agilidade em equipes de desenvolvimento, ao permitir que várias pessoas trabalhassem simultaneamente na mesma base de código. Ainda assim, em projetos de grande porte ou com equipes distribuídas geograficamente, os sistemas centralizados apresentavam limitações no gerenciamento eficiente do trabalho.

Por fim, surgem os modelos mais utilizados atualmente, os *Distributed Concurrent Versions System - DVCS*, como o Git. Esses sistemas, ao contrário da versão anterior, distribuía as cópias do código central entre os desenvolvedores, permitindo um método mais flexível de colaboração. Como cada colaborador poderia ter uma versão local do código, as mudanças realizadas por ele ao código principal poderiam ser administradas localmente antes de serem integradas, permitindo trabalhos offline e que alterações fossem submetidas em lotes ao invés de individualmente.

Contudo, de acordo com Greg Kroah Hartman, 2016, ainda que softwares como *github*,[2] *gerrit*[3] ou outros DVCS possam ser úteis para gerir o fluxo de submissões de softwares menores, eles ainda apresentam muitos problemas para escalar para softwares maiores. Dentre os principais motivos, são citados, por exemplo, a maneira como o fluxo para revisão desses softwares é mais demorado e diminui a produtividade dos mantenedores, a dificuldade de gerenciar e categorizar os inúmeros problemas e submissões com os recursos oferecidos, a maneira como as discussões e comentários dentro da comunidade são pouco acessíveis à outros contribuidores, dificultando a propagação de informação e gerando retrabalho, a dificuldade para que desenvolvedores possam se conectar à listas de discussões e serem notificados sempre que uma novidade relevante ocorra, entre outros. Parte desses problemas da comunidade, porém, ainda segundo Greg Kroah Hartman, 2016, são solucionados ao se substituir os softwares de DVCS por servidores de email, como é feito para a contribuição do kernel.

Essa substituição, entre tanto, também apresenta suas dificuldades, uma vez que, sendo um sistema com perspectiva muito mais abrangente, servidores de email não apresentam funcionalidades e melhorias para esse fluxo. Entre os diversos problemas enfrentados pelo usuário, destacam-se principalmente o grande *overhead* inicial para novos contribuidores, a má rastreabilidade do históricos de submissões e revisões, a escalabilidade limitada, sobrecarregando a lista de email de alguns mantenedores, problemas de corrupção de arquivos, e a dificuldade de se capturar métricas. Além disso, é também nesse fluxo que ocorre a revisão dos *patches*, ou seja, a comunicação direta entre desenvolvedores e mantenedores, sendo essencial que as respostas e notificações ocorram de forma rápida, dado que submissões realizadas durante o período de estabilização ou durante a janela de mesclagem precisam ser avaliadas dentro desses períodos fixos de tempo.

Dada a natureza dessa submissão, em muitos casos, isso implica ainda que os desenvolvedores dependam de ferramentas externas, que ainda precisariam ser configuradas, ou do próprio navegador para checar a lista de e-mails em softwares acessíveis através da web,

[2] https://github.com

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

como o gmail, para responder mensagens e acompanhar o status dos *patches*, sendo um desafio ainda maior quando o endereço de e-mail utilizado para submissões é reutilizado para outros contextos, pois isso aumenta a complexidade de filtrar, organizar e priorizar as mensagens relevantes, gerando ruído na comunicação e dificultando a identificação rápida de respostas e revisões. Por fim, da perspectiva do kw, a dependência de sistemas de email também representa uma fragmentação no fluxo do software e em seu princípio de englobar de forma holística o processo de desenvolvimento. Isso porque o usuário submete alterações pelo Kworkflow, mas precisa recorrer a outros meios para acompanhar revisões e depois retornar para atualizar suas submissões, além de impedir a análise completa do fluxo de contribuição, que é um dos objetivos futuros do projeto.

Capítulo 3

Contribuições para o kw

Esse trabalho dá continuidade à um processo de melhoria continua ao software do kw, iniciada anteriormente através de outros trabalhos como o *Simplificando o processo de contribuição para o kernel Linux* de Neto (2022), que estrutura e refatora a documentação da ferramenta, implementa a versão inicial do banco de dados e também da funcionalidade *kw mail*, posteriormente renomeada para *kw send_patch*, utilizada para submissão de patches através do envio de email's; e também o trabalho *Integrating the Kworkflow system with the Lore archives: Enhancing the Linux kernel developer interaction with mailing lists*, desenvolvido por Barros Tadokoro (2023), que implementa o *patch-hub* – interface de terminal para os arquivos Lore, permitindo o acesso à uma lista oficial de discussões e patches do Kernel Linux.

Como proposta, as contribuições oferecidas por esse trabalho focam em oferecer melhorarias e automatizações para a gestão de patches submetidos por parte dos contribuidores do kernel linux enquanto estão sob processo de revisão, integrando-se ao fluxo de implementações de seus predecessores que, respectivamente, introduzem o processo de envio e de consulta de patches já enviados.

3.1 CRUD banco de dados

Para poder dar suporte para suas diversas funcionalidades, o kw conta com um sistema de banco de dados, desenvolvido em SQLite3, que armazena informações necessárias para o funcionamento, principalmente, das as ferramentas kw pomodoro e kw patch-hub, além de possuir dados de telemetria sobre a utilização do software pelos usuários. Para garantir a consistência e segurança dos dados entre o banco de dados e a aplicação, é crucial desenvolver operações que lidem com operações de manipulação, como inserção, leitura, atualização e deleção de dados (conhecidas como CRUD - create, read, update, delete). Essas operações servem como interface entre as diferentes partes do sistema, permitindo uma interação eficaz e garantindo que os dados sejam gerenciados de forma precisa e confiável.

Contudo, deixar instruções SQL dispersas diretamente no código de aplicação não é considerado uma boa prática de engenharia de software, pois dificulta a manutenção, a legibilidade e a evolução do sistema. O ideal é encapsular o acesso ao banco de dados em

funções ou camadas de abstração que forneçam operações de mais alto nível, reduzindo o acoplamento entre a lógica de negócio e as consultas.

No caso do Kworkflow, desenvolvido em Bash, tal abordagem é limitada pela própria linguagem, que não dispõe de mecanismos nativos para abstração de consultas SQL. Assim, a interação com o banco de dados precisa ser realizada diretamente por meio de comandos de script, o que torna essa separação menos natural, embora ainda desejável para organizar e isolar responsabilidades.

Para tal, o software contava com algumas funções implementadas que permitiam a interação com o banco de dados, mas que não isolavam suficientemente o código e as instruções de acesso aos dados, tornando necessário o uso de comandos SQL em alguns casos. Esse cenário era visível nos comandos de seleção originais (Programa 3.1), que recebiam trechos na linguagem SQL com a cláusula *where*, utilizada para especificar quais critérios devem atender os parâmetros selecionados ou removidos, como exemplificado no Programa 3.2.

Programa 3.1 código select_from antigo.

```
function select_from()
{
  local table="$1"
  local columns="${2:-"*"}"
  local pre_cmd="$3"
  local order_by="$4"
  local flag=${5:-'SILENT'}
  local db="${6:-$DB_NAME}"
  local db_folder="${7:-$KW_DATA_DIR}"
  local db_path
  local query
  local cmd

  db_path="$(join_path "$db_folder" "$db")"

  if [[ ! -f "$db_path" ]]; then
    complain 'Database does not exist'
    return 2
  fi
  if [[ -z "$table" ]]; then
    complain 'Empty table.'
    return 22 # EINVAL
  fi

  query="SELECT $columns FROM $table ;"
  if [[ -n "${order_by}" ]]; then
    query="SELECT $columns FROM $table ORDER BY ${order_by} ;"
  fi

  cmd="sqlite3 -init ${KW_DB_DIR}/pre_cmd.sql -cmd \"${pre_cmd}\" \"${db_path}\"
   ↪ -batch \"${query}\""
  cmd_manager "$flag" "$cmd"
}
```

Programa 3.2 snippet uso função select_from_antiga.

```
is_on_database="$(select_from "kernel_config WHERE name IS '${config_name}'" '' ''
 ↪  '' "$flag")"
```

Além disso, especificamente nos comandos de seleção (Programa 3.3), também foi implementado um parâmetro adicional para permitir o uso da clausula *ordered_by*, que possibilita especificar uma ordenação para os dados retornados com base em um atributo comparável entre eles. O resultado dessa transição para o uso de parâmetros pode ser observado no Programa 3.4.

Programa 3.3 código select_from novo.

```
function select_from()
{
  local table="$1"
  local columns="${2:-"*"}"
  local pre_cmd="$3"
  local _condition_array="$4"
  local order_by=${5:-''}
  local flag=${6:-'SILENT'}
  local db="${7:-"$DB_NAME"}"
  local db_folder="${8:-"$KW_DATA_DIR"}"
  local where_clause
  local db_path
  local query

  db_path="$(join_path "$db_folder" "$db")"

  if [[ ! -f "$db_path" ]]; then
    complain 'Database does not exist'
    return 2
  fi
  if [[ -z "$table" ]]; then
    complain 'Empty table.'
    return 22 # EINVAL
  fi

  if [[ -n "$_condition_array" ]]; then
    where_clause="$(generate_where_clause "$_condition_array")"
  fi

  query="SELECT ${columns} FROM ${table} ${where_clause} ;"

  if [[ -n "${order_by}" ]]; then
    query="${query::-2} ORDER BY ${order_by} ;"
  fi

  cmd="sqlite3 -init "${KW_DB_DIR}/pre_cmd.sql" -cmd \"${pre_cmd}\" \"${db_path}\"
   ↪  -batch \"$query\""
  cmd_manager "$flag" "$cmd"
}
```

Programa 3.4 snippet uso função select_from_nova.

```
condition_array=(['name']="${config_name}")
is_on_database="$(select_from 'kernel_config' '' '' 'condition_array' '' "$flag")"
```

Por fim, essa alteração também permitiu que comparações de desigualdades fossem feitas de forma mais abrangente e ordenada, visto que na função de remoção antiga (Programa 3.5) apenas operações de comparação eram possíveis e que na função de seleção apenas com o código SQL explícito. Para modernizar esse fluxo, implementou-se a nova função de remoção (Programa 3.7) baseada na lógica de geração automática de cláusulas do Programa 3.6.

Programa 3.5 código remove_from antigo.

```
function remove_from()
{
    db_path="$(join_path "${db_folder}" "$db")"
    if [[ ! -f "${db_path}" ]]; then
    complain 'Database does not exist'
    return 2
    fi

    if [[ -z "$table" || -z "${!_condition_array[*]}" ]]; then
    complain 'Empty table or condition array.'
    return 22 # EINVAL
    fi

    for column in "${!_condition_array[@]}"; do
    where_clause+="$column='${_condition_array["${column}"]}'"
    where_clause+=' AND '
    done
    # Remove trailing ' AND '
    where_clause="${where_clause::-5}"

    cmd="sqlite3 -init "${KW_DB_DIR}/pre_cmd.sql" \"${db_path}\" -batch \"DELETE
     ↪  FROM ${table} WHERE ${where_clause};\""
    cmd_manager "$flag" "$cmd"
}
```

Programa 3.6 código generate_where_clause utilizado nas novas funções para gerar a cláusula WHERE SQL a partir dos parâmetros passados.

```
function generate_where_clause()
{
  local -n condition_array_ref="$1"
  local clause
  local relational_op='='
  local attribute
  local where_clause="WHERE "
  local value
```

cont ⟶

⟶ *cont*

```
  for clause in "${!condition_array_ref[@]}"; do
    attribute="$(cut --delimiter=',' --fields=1 <<< "$clause")"
    value="${condition_array_ref["${clause}"]}"

    if [[ "$clause" =~ "," ]]; then
      relational_op=$(cut --delimiter=',' --fields=2 <<< "$clause")
    fi

    where_clause+="${attribute}${relational_op}'${value}'"
    where_clause+=' AND '
  done

  printf '%s' "${where_clause::-5}" # Remove trailing ' AND '
}
```

Programa 3.7 código remove_from novo. .

```
function remove_from()
{
  local table="$1"
  local _condition_array="$2"
  local db="${3:-"${DB_NAME}"}"
  local db_folder="${4:-"${KW_DATA_DIR}"}"
  local flag=${5:-'SILENT'}

  local db_path
  db_path="$(join_path "${db_folder}" "$db")"
  if [[ ! -f "${db_path}" ]]; then
    complain 'Database does not exist'
    return 2
  fi

  if [[ -z "$table" || -z "$_condition_array" ]]; then
    complain 'Empty table or condition array.'
    return 22 # EINVAL
  fi

  where_clause="$(generate_where_clause "$_condition_array")"
  query="DELETE FROM ${table} ${where_clause} ;"

  cmd="sqlite3 -init "${KW_DB_DIR}/pre_cmd.sql" \"${db_path}\" -batch \"$query\""
  cmd_manager "$flag" "$cmd"
}
```

Além disso, outra implementação desenvolvida nessa etapa, foi a implementação do novo método *update_into* (Programa 3.7), que permitia a alteração pontual de algum atributo dentro de uma entidade do banco de dados e da função *generate_set_clause* (Programa 3.9), utilizada para gerar a clausula *set* do SQL, que define quais conjuntos de atributos serão alterados e quais os novos valores para esses atributos. Essa implementação também faz uso da função *generate_where_clause*, uma vez que na maioria das alterações

se faz necessário especificar qual entidade/conjunto de entidades receberá as alterações, como exemplificado no Programa 3.10.

Programa 3.8 código update_into

```
function update_into()
{
  db_path="$(join_path "$db_folder" "$db")"

  if [[ ! -f "$db_path" ]]; then
    complain 'Database does not exist'
    return 2
  fi

  if [[ -z "$table" ]]; then
    complain 'Empty table.'
    return 22 # EINVAL
  fi

  if [[ -z "$_condition_array" || -z "$_updates_array" ]]; then
    complain 'Empty condition or updates array.'
    return 22 #EINVAL
  fi

  where_clause="$(generate_where_clause "$_condition_array")"
  set_clause="$(generate_set_clause "$_updates_array")"

  query="UPDATE ${table} SET ${set_clause} ${where_clause} ;"

  cmd="sqlite3 -init "${KW_DB_DIR}/pre_cmd.sql" -cmd \"${pre_cmd}\" \"${db_path}\"
   ↪ -batch \"$query\""
  cmd_manager "$flag" "$cmd"
}
```

Programa 3.9 código generate_set_clause utilizado para permitir especificar quais atributos serão alterados e quais serão seus novos valores.

```
function generate_set_clause()
{
  local -n condition_array_ref="$1"
  local attribute
  local set_clause
  local value

  for attribute in "${!condition_array_ref[@]}"; do
    value="${condition_array_ref["${attribute}"]}"
    set_clause+="${attribute} = '${value}'"
    set_clause+=', '
  done

  printf '%s' "${set_clause::-2}" # Remove trailing ', '
}
```

Programa 3.10 snippet uso função update_into.

```
# update one row using one unique attribute
condition_array=(['name']='name19')
updates_array=(['attribute1']='att1.2' ['attribute2']='att2.2' ['rank']='10')
update_into 'fake_table' 'updates_array' '' 'condition_array'
```

A padronização dessas operações de CRUD e o isolamento das consultas SQL em funções parametrizadas foram fundamentais para garantir a escalabilidade do sistema. Esta base técnica de persistência de dados permitiu o desenvolvimento de funcionalidades que exigem um gerenciamento mais complexo de informações, como a automação de contatos e grupos, que será detalhada na seção seguinte.

3.2 KW Manage Contacts

No fluxo atual, a submissão de *patches* pode ser realizada por meio da ferramenta *kw send-patch*. Este comando recebe como parâmetros a lista de *commits* a serem enviados, os usuários responsáveis — geralmente mantenedores — e as listas de discussão do subsistema pertinente que devem ser notificadas da contribuição. A partir desses dados, o *kw* gera as modificações necessárias e utiliza internamente o *git send-email* para a transmissão das mensagens aos destinatários. Essa abordagem reflete a função do *kw* como um *hub*, que agrega e simplifica o uso de ferramentas já consolidadas na comunidade, automatizando etapas manuais e aprimorando o fluxo de submissão do desenvolvedor.

Ao lidar com e-mails, é comum que existam grupos de destinatários recorrentes durante a submissão de *patches*. Atualmente, o *kw* disponibiliza a funcionalidade *kw maintainers* que, ao utilizar internamente o *script get_maintainers.pl* do kernel, lista os mantenedores responsáveis pelos subsistemas alterados. Embora essa ferramenta facilite a identificação dos responsáveis oficiais, ela não contempla grupos não oficiais ou externos, como equipes de trabalho e colaboradores de projetos específicos. Diante disso, surgiu a necessidade de uma ferramenta integrada ao fluxo do *kw* para o gerenciamento de grupos de e-mail, visando garantir maior praticidade e consistência na comunicação do desenvolvedor.

3.2.1 Objetivos

Assim, o objetivo principal foi o de criar um sistema que permitisse gerenciar grupos de e-mail de forma centralizada, com armazenamento persistente no banco de dados do kw e acesso através de interface em linha de comando (CLI). Permitindo, através disso, que o usuário pudesse cadastrar contatos, organizar esses contatos em grupos e, posteriormente, incluir automaticamente tais grupos ao enviar patches utilizando o kw send-patch.

3.2.2 Arquitetura

A arquitetura da solução foi planejada de forma modular. O banco de dados é responsável por armazenar contatos individuais (*email_contact*), os grupos (*email_group*) e suas associações (*email_contact_group*), enquanto a interface de linha de comando fornece os comandos necessários para manipulação dessas informações. O modelo de dados contempla

as entidades contato, grupo e a relação entre elas, garantindo flexibilidade para gerenciar múltiplos contextos e equipes (Figura 3.1).

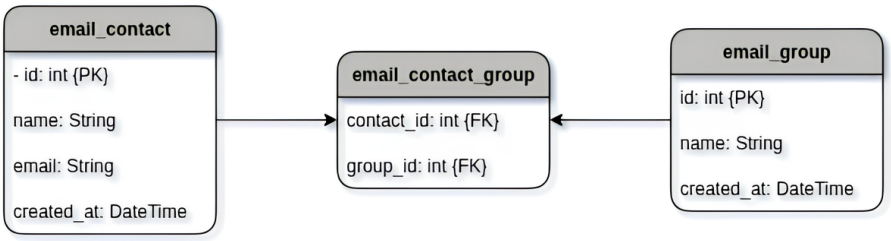

Figura 3.1: *Diagrama Entidade-Relacionamento do kw manage-contacts*

Programa 3.11 modelagem sql kw manage-contacts

```sql
-- Table containing the kw email groups infos
CREATE TABLE IF NOT EXISTS "email_group" (
    "id" INTEGER NOT NULL UNIQUE,
    "name" VARCHAR(50) NOT NULL UNIQUE,
    "created_at" TEXT DEFAULT (date('now', 'localtime')),
    PRIMARY KEY("id")
);

-- Table containing the kw email contacts infos
CREATE TABLE IF NOT EXISTS "email_contact" (
    "id" INTEGER NOT NULL UNIQUE,
    "name" VARCHAR(100) NOT NULL,
    "email" VARCHAR(100) NOT NULL UNIQUE,
    "created_at" TEXT DEFAULT (date('now', 'localtime')),
    PRIMARY KEY("id")
);

-- Table containing the association between a kw email group and it's contacts
CREATE TABLE IF NOT EXISTS "email_contact_group" (
    "contact_id" INTEGER,
    "group_id" INTEGER,
    PRIMARY KEY ("contact_id", "group_id"),
    FOREIGN KEY ("contact_id") REFERENCES "email_contact"("id") ON DELETE CASCADE,
    FOREIGN KEY ("group_id") REFERENCES "email_group"("id") ON DELETE CASCADE
);

CREATE TRIGGER IF NOT EXISTS "delete_contact_if_no_group"
AFTER DELETE ON "email_contact_group"
FOR EACH ROW
WHEN (SELECT COUNT(*) FROM "email_contact_group" WHERE "contact_id" =
 ↪  OLD.contact_id) = 0
BEGIN
    DELETE FROM "email_contact" WHERE "id" = OLD.contact_id;
END;
```

3.2.3 Funcionalidades

A interação com a ferramenta ocorre exclusivamente pelo terminal, de forma a manter compatibilidade com o fluxo tradicional do kw. Foram definidos comandos claros e diretos, permitindo que o usuário visualize grupos existentes, adicione novos contatos, associe-os a diferentes grupos e utilize esses grupos diretamente no envio de e-mails.

As principais funcionalidades implementadas são referenciadas no Programa 3.12.

Programa 3.12 comandos kw manage contacts.

```
'kw manage-contacts:' \
'  manage-contacts (-c | --group-create) [<name>] - create new group' \
'  manage-contacts (-r | --group-remove) [<name>] - remove existing group' \
'  manage-contacts --group-rename [<old_name>:<new_name>] - rename existent
 ↪  group' \
'  manage-contacts --group-add "[<group_name>]:[<contact1_name>]
 ↪  <[<contact1_email>]>, [<contact2_name>] <[<contact2_email>]>, ..." - add
 ↪  contact to existent group' \
'  manage-contacts --group-remove-email "[<group_name>]:[<contact_name>]" -
 ↪  remove contact from existent group' \
'  manage-contacts --group-show=[<group_name>] - show existent groups or
 ↪  specific group contacts'
```

Criação de Grupos

A funcionalidade de criação de grupos, executada por meio do comando `kw manage-contacts group-create`, permite a organização estruturada de contatos para facilitar as submissões. O comando recebe o nome do novo grupo, que é submetido a algumas de validações de integridade antes de sua persistência no banco de dados. Essas verificações asseguram que o identificador proposto seja único, não contenha caracteres especiais e respeite o limite de 50 caracteres. Uma vez atendidos os requisitos, o sistema realiza a inserção do registro (Programa 3.13).

Programa 3.13 create_email_group e create_group.

```
function create_email_group()
{
  local group_name="$1"
  local values

  validate_group_name "$group_name"

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi

  check_existent_group "$group_name"

  if [[ "$?" -ne 0 ]]; then
    warning 'This group already exists'
```

cont ⟶

cont ⟶

```
    return 22 # EINVAL
  fi

  create_group "$group_name"

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi

  return 0
}

function create_group()
{
  local group_name="$1"
  local sql_operation_result

  values="$(format_values_db 1 "$group_name")"

  sql_operation_result=$(insert_into "$DATABASE_TABLE_GROUP" '(name)' "$values" ''
   ↪  'VERBOSE')
  ret="$?"

  if [[ "$ret" -eq 2 || "$ret" -eq 61 ]]; then
    complain "$sql_operation_result"
    return 22 # EINVAL
  elif [[ "$ret" -ne 0 ]]; then
    complain "($LINENO):" $'Error while inserting group into the database with
     ↪  command:\n' "${sql_operation_result}"
    return 22 # EINVAL
  fi

  return 0
}
```

Exclusão de Grupos

A funcionalidade de exclusão, invocada pelo comando `kw manage-contacts group-remove`, permite a remoção definitiva de uma categoria de contatos e de todas as suas referências no sistema. A operação exige a validação da existência prévia do grupo no banco de dados e utiliza a cláusula *CASCADE* na tabela de associação para garantir a consistência dos dados (Programa 3.11). Este mecanismo extingue automaticamente todos os vínculos do grupo, enquanto uma rotina adicional remove contatos que permaneçam sem qualquer outra associação (Programa 3.21).

Programa 3.14 remove_email_group e remove_group.

```
function remove_email_group()
{
  local group_name="$1"
```

cont ⟶

⟶ *cont*

```
  check_existent_group "$group_name"

  if [[ "$?" -eq 0 ]]; then
    warning 'Error, this group does not exist'
    return 22 #EINVAL
  fi

  remove_group "$group_name"

  if [[ "$?" -ne 0 ]]; then
    return 22 #EINVAL
  fi

  return 0
}

function remove_group()
{
  local group_name="$1"
  local sql_operation_result

  condition_array=(['name']="${group_name}")

  sql_operation_result=$(remove_from "$DATABASE_TABLE_GROUP" 'condition_array' '' ''
   ↪  'VERBOSE')
  ret="$?"

  if [[ "$ret" -eq 2 || "$ret" -eq 61 ]]; then
    complain "$sql_operation_result"
    return 22 # EINVAL
  elif [[ "$ret" -ne 0 ]]; then
    complain $'Error while removing group from the database with
     ↪  command:\n'"${sql_operation_result}"
    return 22 # EINVAL
  fi

  return 0
}
```

Renomeação de Grupos

A funcionalidade de renomeação, acessada através do comando `kw manage-contacts group-rename`, permite a alteração de identificadores existentes sem a necessidade de excluir e recriar registros. A operação recebe como parâmetros o nome atual e o novo rótulo, submetendo este último ao mesmo processo de validação do ciclo de criação (limite de caracteres e ausência de símbolos especiais). Essa revalidação assegura que a consistência da base de dados seja preservada, mantendo íntegras as associações de contatos vinculadas ao grupo (Programa 3.15).

Programa 3.15 rename_email_group e rename_group.

```
function rename_email_group()
{
  local old_name="$1"
  local new_name="$2"
  local group_id

  if [[ -z "$old_name" ]]; then
    complain 'Error, group name is empty'
    return 61 # ENODATA
  fi

  check_existent_group "$old_name"

  if [[ "$?" -eq 0 ]]; then
    warning 'This group does not exist so it can not be renamed'
    return 22 # EINVAL
  fi

  validate_group_name "$new_name"

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi

  rename_group "$old_name" "$new_name"

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi

  return 0
}

function rename_group()
{
  local old_name="$1"
  local new_name="$2"
  local sql_operation_result
  local ret

  condition_array=(['name']="${old_name}")
  updates_array=(['name']="${new_name}")

  sql_operation_result=$(update_into "$DATABASE_TABLE_GROUP" 'updates_array' ''
   ↪  'condition_array' 'VERBOSE')
  ret="$?"

  if [[ "$ret" -eq 2 || "$ret" -eq 61 ]]; then
    complain "$sql_operation_result"
    return 22 # EINVAL
  elif [[ "$ret" -ne 0 ]]; then
```

cont ⟶

cont ⟶

```
    complain "($LINENO):" $'Error while removing group from the database with
     ↪  command:\n'"${sql_operation_result}"
    return 22 # EINVAL
  fi

  return 0
}
```

Adicionar contatos à Grupos de Email

A funcionalidade de associação de contatos, executada pelo comando `kw group-add-contact`, permite o gerenciamento e a expansão de grupos por meio da inserção em lote. A operação recebe o nome do grupo alvo e uma lista estruturada no formato `NOME <EMAIL>`. O sistema valida a existência do grupo, segmenta a entrada e executa verificações de integridade sintática em cada endereço de e-mail. Uma vez validados, os contatos são persistidos e vinculados ao grupo correspondente (Programa 3.16).

Programa 3.16 add_email_contacts e add_contact_group

```
function add_email_contacts()
{
  if [[ -z "$contacts_list" ]]; then
    complain 'The contacts list is empty'
    return 61 # ENODATA
  fi

  if [[ -z "$group_name" ]]; then
    complain 'The group name is empty'
    return 61 # ENODATA
  fi

  check_existent_group "$group_name"
  group_id="$?"

  if [[ "$group_id" -eq 0 ]]; then
    complain 'Error, ubable to add contacts to unexistent group'
    return 22 # EINVAL
  fi

  split_contact_infos "$contacts_list" _contacts_array

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi

  add_contacts _contacts_array

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi
```

cont ⟶

⟶ *cont*

```
  add_contact_group _contacts_array "$group_id"

  if [[ "$?" -ne 0 ]]; then
    return 22 # EINVAL
  fi

  return 0
}

function add_contact_group()
{
  local -n contacts_array="$1"
  local group_id="$2"
  local values
  local email
  local contact_id
  local ctt_group_association
  local sql_operation_result
  local ret

  for email in "${!contacts_array[@]}"; do
    condition_array=(['email']="${email}")
    contact_id="$(select_from "$DATABASE_TABLE_CONTACT" 'id' '' 'condition_array')"
    values="$(format_values_db 2 "$contact_id" "$group_id")"

    condition_array=(['contact_id']="${contact_id}" ['group_id']="${group_id}")
    ctt_group_association="$(select_from "$DATABASE_TABLE_CONTACT_GROUP" 'contact_id,
    ↪  group_id' '' 'condition_array')"
    if [[ -n "$ctt_group_association" ]]; then
      continue
    fi

    sql_operation_result=$(insert_into "$DATABASE_TABLE_CONTACT_GROUP" '(contact_id,
    ↪  group_id)' "$values" '' 'VERBOSE')
    ret="$?"

    if [[ "$ret" -eq 2 || "$ret" -eq 61 ]]; then
      complain "$sql_operation_result"
      return 22 # EINVAL
    elif [[ "$ret" -ne 0 ]]; then
      complain "($LINENO):" $'Error while trying to insert contact group into the
      ↪  database with the command:\n'"${sql_operation_result}"
      return 22 # EINVAL
    fi

  done

  return 0
}
```

Exibir grupos

A funcionalidade de consulta, operada pelo comando `kw manage-contacts group-show`, permite visualizar as informações armazenadas no banco de dados de duas maneiras. Quando um identificador de grupo é fornecido como parâmetro, o sistema valida sua existência e lista todos os contatos vinculados (Figura 3.2). Caso o comando seja invocado sem parâmetros, o sistema apresenta um resumo de todos os grupos cadastrados (Figura 3.3). A lógica de processamento e formatação da saída de dados detalhada pode ser observada no Programa 3.17.

Programa 3.17 show email groups, print_groups_infos e print_contacts_infos.

```
function show_email_groups()
{
  local group_name="$1"
  local columns="$2"
  local groups_info
  local contacts_info
  local contact_id
  declare -a contacts_array
  declare -a groups_array

  if [[ -n "$group_name" ]]; then

    check_existent_group "$group_name"

    if [[ "$?" -eq 0 ]]; then
      complain 'Error unexistent group'
      return 22 #EINVAL
    fi

    contacts_info="$(get_groups_contacts_infos "$group_name" '*')"
    IFS=',' read -ra contacts_array <<< "$contacts_info"
    print_contact_infos "$group_name" 'contacts_array' "$columns"
    return
  fi

  groups_info="$(select_from "$DATABASE_TABLE_GROUP")"
  readarray -t groups_array <<< "$groups_info"
  print_groups_infos 'groups_array' "$columns"
}

function print_contact_infos()
{
  local group_name="$1"
  local -n _contacts_array="$2"
  local columns="$3"
  local group_name_width=${#group_name}
  local trim_width=$(((columns - group_name_width) / 2))
  local remaining_width=$((columns - group_name_width - trim_width))
  local id_width=8
  local name_width=50
```

cont ⟶

⟶ *cont*

```
  local associate_groups_width=20
  local created_at_width=12
  local email_width=$((columns - id_width - name_width - associate_groups_width -
   ↪  created_at_width - 8))

  if [[ -z $columns ]]; then
    columns="$(tput cols)"
  fi

  printf "%*s%s%*s\n" "$trim_width" "" "$group_name" "$remaining_width" "" | tr ' '
   ↪  '-'

  printf "%-${id_width}s|%-${name_width}s|%-${email_width}s|" \
       "%-${associate_groups_width}s|%-${created_at_width}s\n" \
       "ID" "Name" "Email" "Associated Groups" "Created at"

  printf "%-${columns}s\n" | tr ' ' '-'

  for contact in "${_contacts_array[@]}"; do
    IFS='|' read -r id name email created_at <<< "$contact"
    condition_array=(['contact_id']="$id")
    associate_groups_num="$(select_from "$DATABASE_TABLE_CONTACT_GROUP" 'COUNT(*)'
     ↪  '' 'condition_array')"
    printf "%-${id_width}s|%-${name_width}s|%-${email_width}s|" \
       "%-${associate_groups_width}s|%-${created_at_width}s\n" \
       "$id" "$name" "$email" "$associate_groups_num" "$created_at"

  done
  printf "%-${columns}s\n" | tr ' ' '-'

}

function print_groups_infos()
{
  local -n groups_info="$1"
  local columns="$2"
  local id_width=8
  local contact_num_width=25
  local created_at_width=20
  local name_width=$(("$columns" - id_width - contact_num_width - created_at_width -
   ↪  6))

  if [[ -z $columns ]]; then
    columns="$(tput cols)"
  fi

  printf
   ↪  "%-${id_width}s|%-${name_width}s|%-${contact_num_width}s|%-${created_at_width}s\n"
   ↪  "ID" "Name" "Contacts" "Created at"
  printf "%-${columns}s\n" | tr ' ' '-'

  for group in "${!groups_info[@]}"; do
```

cont ⟶

⟶ *cont*

```
    IFS='|' read -r id name created_at <<< "${groups_info[$group]}"
    condition_array=(['group_id']="$id")
    contact_num="$(select_from "$DATABASE_TABLE_CONTACT_GROUP" 'COUNT(*)' ''
     ↪  'condition_array')"
    printf
     ↪  "%-${id_width}s|%-${name_width}s|%-${contact_num_width}s|%-${created_at_width}s\n"
     ↪  "$id" "$name" "$contact_num" "$created_at"
  done

  printf "%-${columns}s\n" | tr ' ' '-'
}
```

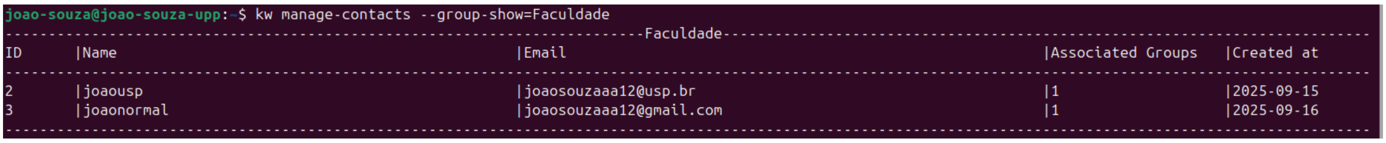

Figura 3.2: *Exemplo do comando "group_show" para um grupo específico.*

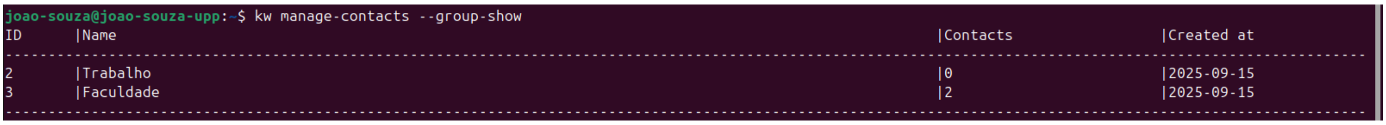

Figura 3.3: *Exemplo do comando "group_show" sem um grupo especificado.*

3.2.4 Enviar patches para grupos

A integração de grupos de e-mail ao fluxo de submissões é viabilizada pelos novos parâmetros `-to-groups` e `-cc-groups`, incorporadas a feature *kw send-patch*. Essa funcionalidade permite ao usuário enviar patches a grupos de contatos pré-definidos, eliminando a necessidade de inserção manual de múltiplos endereços, principalmente em submissões recorrentes.

Para que isso seja possível, o *send-patch* recebe o nome dos grupos em listas contendo identificadores de grupos separados por vírgulas. Internamente, o sistema processa essa entrada realizando consultas ao banco de dados para realizar a consulta dos nomes de grupos e encontrar os endereços de e-mail válidos, que são então injetados nos campos de destinatário (*To*) ou de cópia (*Cc*) da mensagem (Programa 3.19). Os novos parâmetros podem ser encontrados em: Programas 3.18.

Programa 3.18 opções do comando –send do kw send-patch contendo o to-groups e o cc-groups.

```
| *kw send-patch* (-s | \--send) [\--simulate] [\--private] [\--rfc]
[\--to='<recipient>,...'] [\--cc='<recipient>,...']
[\--to-groups='<recipient>,...'] [\--cc-groups='<recipient>,...']
[<rev-range>...] [-v<version>] [\-- <extra-args>...]
```

Programa 3.19 Função send_patch_main com métodos –to-groups e cc-groups.

```
function send_patch_main()
{
  local flag
  flag=${flag:-'SILENT'}
  if [[ "$1" =~ -h|--help ]]; then
    send_patch_help "$1"
        @@ -89,7 +90,9 @@
  local flag="$1"
  local opts="${send_patch_config[send_opts]}"
  local to_recipients="${options_values['TO']}"
  local to_groups_recipients="${options_values['TO_GROUPS']}"
  local cc_recipients="${options_values['CC']}"
  local cc_groups_recipients="${options_values['CC_GROUPS']}"
  local dryrun="${options_values['SIMULATE']}"
  local commit_range="${options_values['COMMIT_RANGE']}"
  local version="${options_values['PATCH_VERSION']}"
  local extra_opts="${options_values['PASS_OPTION_TO_SEND_EMAIL']}"
  local private="${options_values['PRIVATE']}"
  local rfc="${options_values['RFC']}"
  local kernel_root
  local patch_count=0
  local cmd='git send-email'
  flag=${flag:-'SILENT'}

  [[ -n "$dryrun" ]] && cmd+=" $dryrun"

  if [[ -n "$to_groups_recipients" ]]; then
    validate_email_group_list "$to_groups_recipients" || exit_msg 'Please review
     ↪  your `--to-groups` list.'
    if [[ -n "$to_recipients" ]]; then
      to_recipients+=','
    fi
    to_recipients+=$(get_groups_contacts_infos "$to_groups_recipients" 'email')
  fi

  if [[ -n "$cc_groups_recipients" ]]; then
    validate_email_group_list "$cc_groups_recipients" || exit_msg 'Please review
     ↪  your `--cc-groups` list.'
    if [[ -n "$cc_recipients" ]]; then
      cc_recipients+=','
    fi
    cc_recipients+=$(get_groups_contacts_infos "$cc_groups_recipients" 'email')
  fi

  if [[ -n "$to_recipients" ]]; then
    validate_email_list "$to_recipients" || exit_msg 'Please review your `--to`
     ↪  list.'
    cmd+=" --to=\"$to_recipients\""
  fi
  if [[ -n "$cc_recipients" ]]; then
    validate_email_list "$cc_recipients" || exit_msg 'Please review your `--cc`
     ↪  list.'
```

cont ⟶

```
    ⟶ cont
      cmd+=" --cc=\"$cc_recipients\""
    fi
    # Don't generate a cover letter when sending only one patch
    patch_count="$(pre_generate_patches "$commit_range" "$version")"
    if [[ "$patch_count" -eq 1 ]]; then
      opts="$(sed 's/--cover-letter//g' <<< "$opts")"
    fi
    kernel_root="$(find_kernel_root "$PWD")"
    # if inside a kernel repo use get_maintainer to populate recipients
    if [[ -z "$private" && -n "$kernel_root" ]]; then
      generate_kernel_recipients "$kernel_root"
      cmd+=" --to-cmd='bash ${KW_PLUGINS_DIR}/kw_mail/to_cc_cmd.sh ${KW_CACHE_DIR}
       ↪  to'"
      cmd+=" --cc-cmd='bash ${KW_PLUGINS_DIR}/kw_mail/to_cc_cmd.sh ${KW_CACHE_DIR}
       ↪  cc'"
    fi
          @@ -931,27 +950,27 @@
      [[ "$1" =~ ^--$ ]] && dash_dash=1
      # The added quotes ensure arguments are correctly separated
      options="$options \"$1\""
      shift
    done
    if [[ -n "$commit_count" ]]; then
      # add `--` if not present
      [[ "$dash_dash" == 0 ]] && options="$options --"
      options="$options $commit_count"
    fi
    printf '%s' "$options"
}
```

3.2.5 Resultados

Entre os benefícios da abordagem adotada estão a maior praticidade no gerenciamento de destinatários, a redução de erros manuais na inclusão de e-mails e a possibilidade de reutilização de grupos em diferentes contextos. Isso se traduz em um processo mais ágil e confiável no envio de patches.

Apesar dos avanços alcançados, algumas limitações ainda podem ser apontadas. A ferramenta oferece suporte apenas via CLI, não possuindo interface para o usuário via terminal, que poderia ser desejável, principalmente, para visualizar informações dos grupos e contatos de uma maneira mais organizada.

3.3 KW Patch track

Atualmente, embora seja possível submeter *patches* através do *kw*, ainda não existe um mecanismo eficaz para acompanhar e gerenciar o ciclo de vida dessas submissões. Conforme novas versões de um mesmo *patch* são enviadas e as revisões se acumulam, torna-se cada vez mais difícil manter o controle sobre o histórico, as respostas recebidas e o estado atual de cada alteração. Esta lacuna é significativa, pois obriga o desenvolvedor

a realizar esse acompanhamento de forma externa e manual, o que fragmenta o fluxo de trabalho e aumenta a carga cognitiva. Diante dessa limitação, surgiu a necessidade de uma funcionalidade capaz de registrar, rastrear e atualizar automaticamente o status dos *patches* submetidos, centralizando o gerenciamento do ciclo de vida diretamente no ecossistema do *kw*.

Um vídeo demonstrativo dessa ferramenta pode ser encontrado em: https://jgbsouza.github.io/Mac0499---TCC/demonstracao_kw_patch_track.webm

3.3.1 Objetivos

Assim, o objetivo principal do *Patch Track* é permitir que o usuário acompanhe de forma automatizada o progresso de suas contribuições, desde o envio inicial até a integração no repositório, reduzindo o esforço manual e promovendo maior clareza sobre o processo de revisão.

Além disso, o sistema busca oferecer uma base sólida para extensões futuras, como integração com repositórios oficiais e coleta de métricas sobre o fluxo de contribuição, incluindo tempo médio de resposta, aprovação e integração de patches. Essa capacidade de extração de dados é fundamental para consolidar o *kw* como uma ferramenta de suporte à pesquisa científica, permitindo o estudo empírico e sistemático do modelo de desenvolvimento do kernel Linux e o entendimento aprofundado de suas dinâmicas de colaboração em larga escala.

3.3.2 Arquitetura

A arquitetura do *Patch Track* foi projetada de forma modular e baseada em um modelo relacional de entidades interligadas. Todas as informações são armazenadas em banco de dados, garantindo rastreabilidade e consistência das submissões.

A entidade central, *patch*, armazena informações como o autor, o *message-id* da submissão, a versão e o status atual do patch. O campo *outdated* indica quando uma versão mais recente substitui outra, preservando o histórico completo das alterações.

A entidade *contribution* agrupa logicamente diferentes versões de um mesmo trabalho, mantendo informações sobre a data da última interação e o repositório de destino. Esse repositório é representado pela entidade *repository*, que contém dados como nome, URL e *branch* associada na qual a contribuição deve ser integrada assim que aprovada, além dos mantenedores vinculados à esse repositório, permitindo identificar revisores e correlacionar respostas relevantes nas threads de e-mail.

O rastreamento dos envios é realizado pela tabela *patch_submission*, que registra o identificador da mensagem, o remetente e o vínculo entre cada envio e o patch correspondente. O sistema também oferece suporte a *tags*, utilizadas como marcadores semânticos para facilitar a filtragem, a categorização e a exibição das informações.

Essa estrutura de dados estabelece uma base robusta para o controle do ciclo de vida dos patches e possibilita futuras expansões, como integração com serviços externos de revisão e automação de métricas analíticas (Figura 3.4).

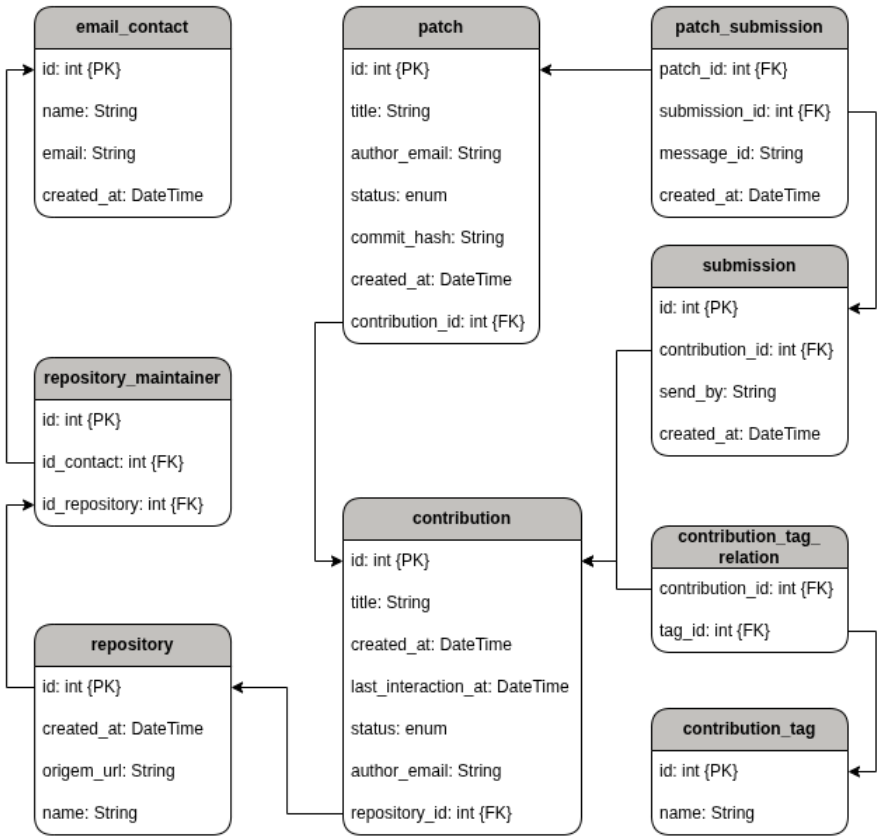

Figura 3.4: *Diagrama Entidade-Relacionamento do Kw patch_track*

3.3.3 Funcionalidades

O *kw patch_track* oferece um conjunto de funcionalidades voltadas à automatização e ao gerenciamento das submissões de patches. Todas as interações ocorrem de forma integrada ao fluxo do *kw*, mantendo a compatibilidade com a ferramenta principal de envio.

As principais funcionalidades implementadas são referenciadas no Programa 3.20.

Programa 3.20 comandos kw patch-track.

```
'kw patch-track:' \
'  patch-track (--show-patches) [[--from <YYYY-MM-DD>] | [--after <YYYY-MM-DD>]
 ↪  [--before <YYYY-MM-DD>]] - Show patches dashboard in chronological order ' \
'  patch-track (-d | --show-contributions) - Show all contributions ' \
'  patch-track (--id <num>) [-s <status> | --set-status <status>] - Set a patch
 ↪  status ' \
'  patch-track (-u | --update) - Update patch statuses using heuristics ' \
'  patch-track (-c | --contribution-id <id>) (--set-repository <name:url>) -
 ↪  Associate a repository to a contribution ' \
'  patch-track (-r | --repository-id <id>) (-m | --set-maintainer <name:email>) -
 ↪  Associate a maintainer to a repository ' \
'  patch-track (-c | --contribution-id <id>) (-o | --open-contribution) - Open
 ↪  contribution email thread in mutt '
```

Registro e Rastreamento das submissões e contribuições

Durante a submissão dos patches com a ferramenta *kw send_patch*, o sistema permite identificar ou criar uma contribuição por meio do terminal interativo, que lista as contribuições ativas do usuário para reutilização ou criação de uma nova (Figura 3.5). Após a submissão, cada patch enviado é cadastrado no banco de dados com informações como versão, título, autor, data de criação e *commit_hash*.

Quando um patch corresponde a uma versão já existente — isto é, quando título, autor, *commit_hash* e contribuição coincidem — apenas a nova submissão é registrada, evitando duplicação de versões. Em seguida, é criada uma nova *submission*, agregando todas as submissões individuais feitas naquela execução do *kw send-patch* e vinculando-as à contribuição correspondente.

```
joao-souza@joao-souza:~/kworkflow$ kw send-patch --send --to="joaosouzaaa12@usp.br"
Select one of the contributions bellow or write a new title:
 (1 - previous_contribution_title
2 - monografia_demo_test):
```

Figura 3.5: *Identificando a contribuição Kw patch_track*

Para extrair e salvar as informações dos patches submetidos, a ferramenta se utiliza da técnica de raspagem de dados de dois tipos de arquivos gerados durante a etapa de envio. O primeiro desses arquivos (Figura 3.6), gerado temporariamente para esse fluxo, é resultado do redirecionamento da saída do comando *git send-email*, utilizado pelo *send-patch* para publicação dos patches. Desse arquivo então o *kw patch-track* extrai grande parte das informações, como o título, email do autor do commit/patch, email do remetente (pode não ser o mesmo usado para criar os commits), os emails dos destinatários, data e horário de submissão e por fim o message-id. Adicionalmente, para ter acesso aos hashes dos commits, avalia-se também os arquivos de patches (Figura 3.7) preliminares, gerados pelo *kw send-patch* para pré-processamento interno. Ainda que parte dos dados extraidos do resultado da submissão estejam disponíveis também no arquivo do patch, o fato de que parte das informações como títulos, autor e destinatários podem ser reescrita durante a submissão somado ao fato de que esses arquivos contém textos adicionais com o conteúdo do patch, poderiam levar a erros de julgamento ou informações imprecisas na hora da extração.

Integração com o mutt

O *kw patch_track* oferece integração com o cliente de e-mail em terminal *mutt*,[1] um cliente amplamente utilizado pela capacidade de exibir emails diretamente no terminal. O objetivo dessa integração é permitir que o usuário visualize, de forma prática, os e-mails relacionados às submissões de patches e até mesmo os responda através do comando *kw patch_track open-contribution <contribution-id>* (Figura 3.8), ao mesmo tempo em que o sistema utiliza o *mutt* como ferramenta auxiliar para automatizar a análise das mensagens e *headers* de emails para identificar informações relevantes para o fluxo de atualização de status.

[1] https://mutt.org

```
/tmp/KMNHKe3S5p/0001-feat-add-file-with-template-configs-for-mutt.patch
(mbox) Adding cc: JGBSouza <joaosouzaaa12@usp.br> from line 'From: JGBSouza <joaosouzaaa12@usp.br>'
(body) Adding cc: JGBSouza <joaosouzaaa12@usp.br> from line 'Signed-off-by: JGBSouza <joaosouzaaa12@usp.br>'

From: JGBSouza <joaosouzaaa12@usp.br>
To: joaosouzaaa12@usp.br
Subject: [PATCH] feat: add file with template configs for mutt
Date: Mon, 22 Dec 2025 23:27:31 -0300
Message-ID: <20251223022807.38905-1-joaosouzaaa12@usp.br>
X-Mailer: git-send-email 2.43.0
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit

OK. Log says:
Server: smtp.gmail.com
MAIL FROM:<joaosouzaaa12@usp.br>
RCPT TO:<joaosouzaaa12@usp.br>
From: JGBSouza <joaosouzaaa12@usp.br>
To: joaosouzaaa12@usp.br
Subject: [PATCH] feat: add file with template configs for mutt
Date: Mon, 22 Dec 2025 23:27:31 -0300
Message-ID: <20251223022807.38905-1-joaosouzaaa12@usp.br>
X-Mailer: git-send-email 2.43.0
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit

Result: 250
```

Figura 3.6: *Resultado do comando send_patch*

```
  GNU nano 7.2 /tmp/KMNHKe3S5p/0001-feat-add-file-with-template-configs-for-mutt.patch
From eaf5a64b0c57d29d7c3faaa42697de2093e99eee Mon Sep 17 00:00:00 2001
From: JGBSouza <joaosouzaaa12@usp.br>
Date: Mon, 22 Dec 2025 22:25:55 -0300
Subject: [PATCH] feat: add file with template configs for mutt

Signed-off-by: JGBSouza <joaosouzaaa12@usp.br>
---
 etc/patch_track_mutt.config | 8 ++++++++
 1 file changed, 8 insertions(+)
 create mode 100644 etc/patch_track_mutt.config

diff --git a/etc/patch_track_mutt.config b/etc/patch_track_mutt.config
new file mode 100644
index 0000000..a985870
--- /dev/null
+++ b/etc/patch_track_mutt.config
@@ -0,0 +1,8 @@
+# Mutt options to be used with patch-track
+imap_user=
+imap_pass=
+
+folder="imaps://imap.gmail.com"
+spoolfile="+[Gmail]/Todos os e-mails"
+record="+[Gmail]/Sent Mail"
+mbox_type="Maildir"
--
2.43.0
```

Figura 3.7: *Arquivo de um patch com alterações propostas*

Para viabilizar essa funcionalidade, os pacotes `mutt`, `xvfb` e `xterm` foram adicionados às dependências do projeto. Essas ferramentas são instaladas automaticamente durante a instalação do *kworkflow* ao executar o arquivo de setup: *./setup.sh -i*.[2]

Além das dependências, um arquivo de configuração padrão do *mutt* é criado durante a instalação, contendo os parâmetros necessários para autenticação e leitura de e-mails via IMAP. O template atual foi configurado para uso com contas do Gmail e define opções como

[2] Esse processo segue o procedimento documentado na página oficial de instalação do *kworkflow* https://kworkflow.org/content/installanduninstall.html

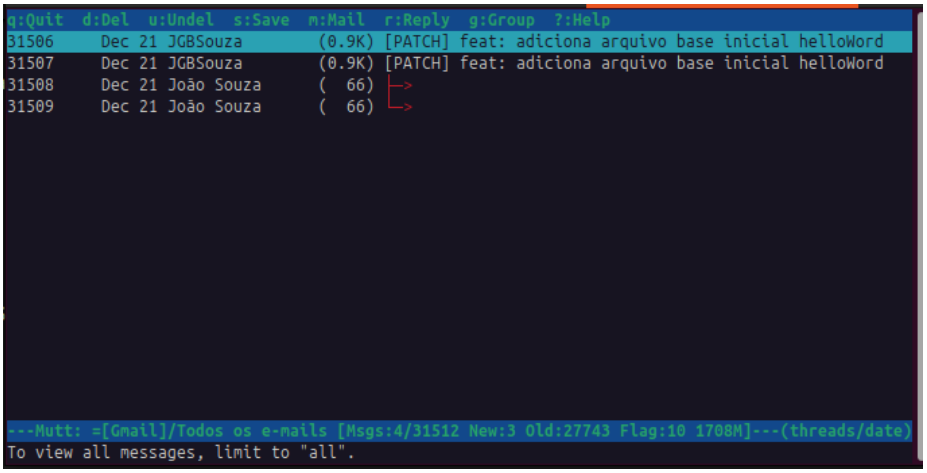

Figura 3.8: *Abrindo uma contribuição no mutt*

o servidor IMAP, a mailbox padrão e o tipo de armazenamento. Entre as configurações incluídas estão:

Programa 3.21 arquivo de configurações mutt.

```
# Mutt options to be used with patch-track
imap_user=
imap_pass=

folder="imaps://imap.gmail.com"
spoolfile="+[Gmail]/Todos os e-mails"
record="+[Gmail]/Sent Mail"
mbox_type="Maildir"
```

Além dessas configurações, durante a primeira execução do kw patch_track, o usuário fornecerá de maneira interativa o seu imap_user e imap_pass (Figura 3.9), respectivamente o seu email e a senha de aplicativo gerada para a sua conta do Gmail. Com todas essas configurações definidas, o *kw patch-track* consegue que o *mutt* abra diretamente a mailbox "Todos os e-mails" do Gmail, possibilitando que o usuário visualize suas mensagens pelo terminal. Paralelamente, o *kw patch_track* utiliza o *mutt* de forma programática para listar mensagens, extrair *headers* e identificar respostas, novas versões e outros elementos essenciais para o rastreamento automático dos patches — sem exigir interação do usuário para essas operações.

Definição de Repositório

Após selecionar ou criar uma contribuição durante o processo de envio pelo *kw send_patch*, o usuário pode definir o repositório associado àquela contribuição através do comando *kw patch-track –set-repository <repository_name:repository_origin_url> –contribution <contribution_id>* (Figura 3.10).

O repositório definido é armazenado na contribuição e, além de melhorar o contexto do registro das submissões, permite que em futuras implementações esse dado seja utilizado

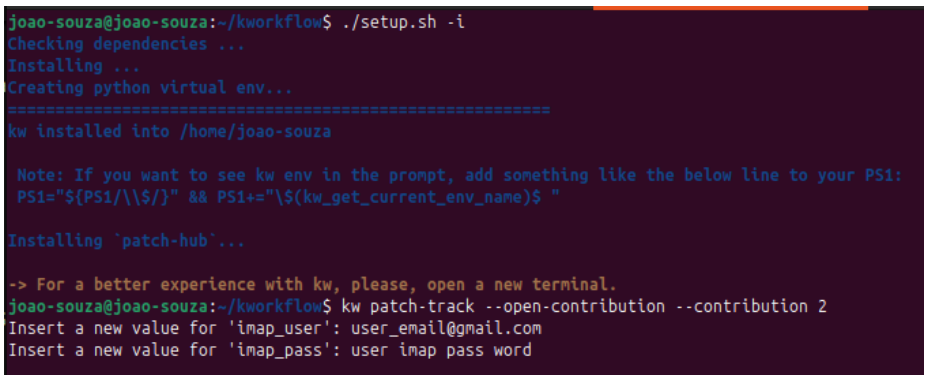

Figura 3.9: *Configurando imap_user e imap_pass para o mutt*

pelo sistema para determinar o destino previsto para integração dos patches e se essa integração já foi realizada. Essa informação também auxilia na recuperação de contexto para futuras submissões vinculadas à mesma contribuição, garantindo consistência no fluxo de trabalho.

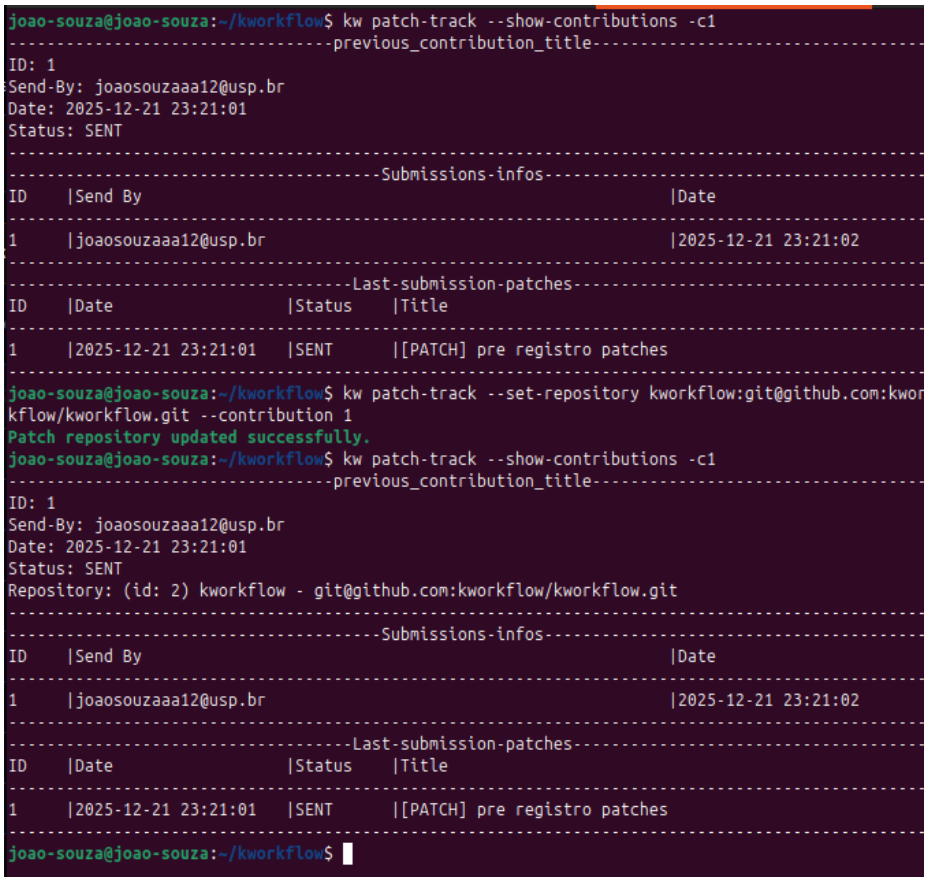

Figura 3.10: *Identificando o repositório de uma contribuição*

Definição de Mantenedor

Após a definição do repositório ligado à contribuição, o sistema oferece ao usuário a possibilidade de indicar um mantenedor responsável por aquele repositório através do co-

mando *kw patch-track –set-maintainer <maintainer_name:maintainer_email>* (Figura 3.11).

A associação entre repositório e mantenedor facilita a identificação de revisores potenciais, bem como a correlação de mensagens relevantes nas threads de e-mail. Embora não interfira diretamente no processo de submissão, essa informação contribui para uma melhor organização e para o acompanhamento do fluxo de revisão, garantindo maior rastreabilidade no ciclo de vida das contribuições. Futuramente, essa informação pode ser utilizada para melhor determinar e-mails que identifiquem a aprovação de uma submissão antes de sua integração final.

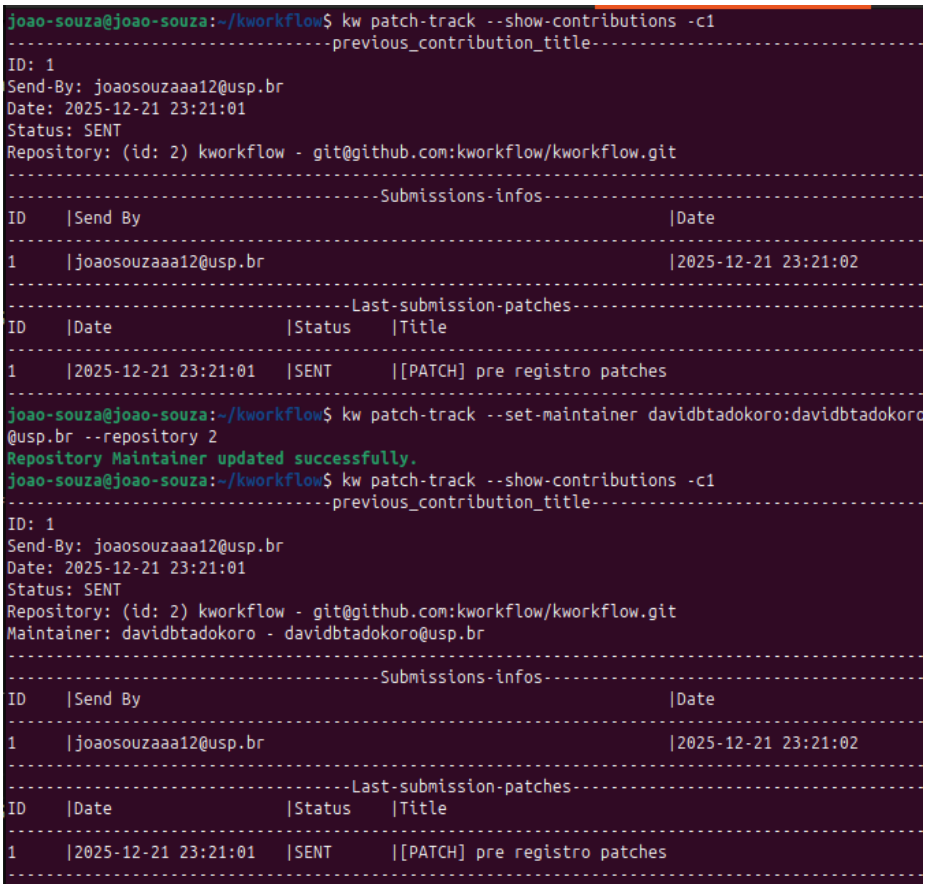

Figura 3.11: *Identificando o mantenedor de um repositório*

Atualização Automática de Status

O sistema implementa uma lógica de atualização automática dos status das contribuições através do comando *kw patch-track update-contribution <contribution_id>*, baseada em heurísticas inspiradas no fluxo de revisão do *kernel Linux* (Programa 3.22). Durante essa etapa, o comando atualiza o status dos patches indivudalmente (Programa 3.23) e, por fim, o estado final da contribuição (Programa 3.24):

Os estados possíveis para um patch incluem:

- **Submetido/Em revisão:** atribuído a patches recém-enviados;
- **Revisado:** mantido enquanto há respostas na thread sem substituições;

- **Aprovado:** definido ao detectar respostas contendo marcadores como *Reviewed-by ou Approved*;
- **Mergeado:** atribuído quando a contribuição correspondente é identificada no repositório de destino.

Os estados possíveis para uma contribuição incluem:

- **Submetido / Em revisão**: representa o estado inicial ou ativo, aplicado quando o conjunto de patches ainda não foi totalmente aprovado ou integrado, e não há sinalização específica de revisão pendente para patches individuais.
- **Revisado**: indica que o fluxo de *feedback* foi iniciado, sendo atribuído sempre que ao menos um patch da contribuição for movido para o estado de revisão.
- **Aprovado**: definido caso um email de aprovação tenha sido encontrado e nenhum patch esteja no estado de revisão.
- **Mergeado**: definido caso a totalidade dos patches tenha sido integrada com sucesso ao repositório de destino.

Embora o sistema contemple o estado **Mergeado**, a detecção automática desse evento ainda não foi implementada. Inicialmente foi investigada a possibilidade de localizar, no repositório, o hash do commit gerado a partir do patch submetido. Contudo, no modelo de contribuição do *kernel Linux*, o commit final costuma ser modificado pelos mantenedores — o hash muda devido a alterações no *message*, ajustes manuais, aplicação com `-signoff`, rebase ou integração via mecanismos internos de manutenção. Como consequência, não é possível inferir de forma confiável a correspondência direta entre um patch enviado por e-mail e o commit final integrado ao repositório, inviabilizando uma heurística simples para essa etapa. Além disso, a implementação atual também não realiza distinções entre os remetentes dos e-mails que identifiquem o estado de aprovação de um patch, permitindo com que tanto mantenedores quanto não mantenedores sejam considerados nesse processo.

Programa 3.22 Código update_contribution_status

```
function update_contribution_status()
{
  condition_array=(['id']="$contribution_id")
  contribution_repository_id="$(get_contribution_info 'repository_id'
   ↪  'condition_array')"

  condition_array=(['id']="$contribution_repository_id")
  repository_origin_url="$(get_repository_info 'origin_url' 'condition_array')"

  submission_id="$(get_last_submission_infos_by_contribution_id 'id'
   ↪  "$contribution_id")" || return 22
  condition_array=(['submission_id']="$submission_id")

  patch_submission_infos="$(
    get_patch_submission_info 'patch_id, submission_id, message_id'
     ↪  'condition_array'
```

cont ⟶

```
  ⟶ cont
)" || return 22

for patch_id in $patch_submission_infos; do
  IFS='|' read -r patch_id submission_id message_id <<< "$patch_submission_infos"

  condition_array=(['patch_id']="$patch_id"
    ['submission_id']="$submission_id"
    ['message_id']="$message_id")
  message_id="$(get_patch_submission_info 'message_id' 'condition_array')" ||
   ↪ continue

  condition_array=(['id']="$patch_id")
  patch_infos="$(get_patch_info 'commit_hash, status' 'condition_array')" ||
   ↪ continue

  IFS='|' read -r commit_hash status <<< "$patch_infos"

  final_status="$(
    update_patch_status \
      "$patch_id" \
      "$message_id" \
      "$commit_hash" \
      "$status"
  )"
  patches_status["$patch_id"]="$final_status"
done

decide_contribution_status "$contribution_id" 'patches_status'
show_contributions_dashboard '' '' "$contribution_id"
}
```

Programa 3.23 Código update_patch_status

```
function update_patch_status()
{
  if [[ "$patch_current_status" == 'MERGED' ]]; then
    printf '%s' 'MERGED'
    return 0
  fi

  [[ -f '/tmp/mutt-status' ]] && rm -f '/tmp/mutt-status'

  xterm -iconic -e \
    sh -c "mutt -F ${MUTT_RC_PATH} \
      -e 'push \"l ((~i ${patch_message_id})|(~x ${patch_message_id})) ~b .*
       ↪ <enter><pause><enter>q\"' \
      ; echo finished > /tmp/mutt-status" \
    > /dev/null 2>&1 &

  while [[ ! -s '/tmp/mutt-status' ]]; do
    sleep 0.1
```

cont ⟶

⟶ *cont*

```
  done

  mapfile -t reply_files < <(grep -R -l "In-Reply-To:.*${patch_message_id}"
   ↪  "$KW_MUTT_MESSAGES_DIR")

  if [[ "${#reply_files[@]}" -eq 0 ]]; then
    set_patch_status "$patch_id" 'SENT'
    printf '%s' 'SENT'
    return 0
  fi

  for file in "${reply_files[@]}"; do
    if grep -Eiw "Approved|Reviewed-by" "$file" > /dev/null 2>&1; then
      set_patch_status "$patch_id" 'APPROVED'
      printf '%s' 'APPROVED'
      return 0
    fi
  done

  last_msg_file=$(printf '%s\n' "${reply_files[@]}" |
    xargs -d '\n' stat --format='%Y %n' 2> /dev/null |
    sort -n |
    tail -n1 |
    cut -d' ' -f2-)

  if [[ -z "$last_msg_file" ]]; then
    set_patch_status "$patch_id" 'SENT'
    printf '%s' 'SENT'
    return 0
  fi

  last_author=$(grep -i '^From:' "$last_msg_file" |
    sed -E 's/From:.*<([^>]+)>.*/\1/')

  if [[ "$last_author" == "${patch_track_mutt_config['imap_user']}" ]]; then
    set_patch_status "$patch_id" 'SENT'
    printf '%s' 'SENT'
  else
    set_patch_status "$patch_id" 'REVIEWED'
    printf '%s' 'REVIEWED'
  fi

  return 0
}
```

Programa 3.24 Código decide_contribution_status

```
function decide_contribution_status()
{
  local contribution_id="$1"
  local -n _contribution_patches_status="$2"
```

cont ⟶

⟶ *cont*

```
  local has_revisado=0
  local has_aprovado=0
  local has_mergeado=0
  local has_sent=0

  for pid in "${!_contribution_patches_status[@]}"; do
    case "${_contribution_patches_status[$pid]}" in
      REVIEWED) has_revisado=1 ;;
      APPROVED) has_aprovado=1 ;;
      MERGED) has_mergeado=1 ;;
      SENT) has_sent=1 ;;
    esac
  done

  if [[ $has_revisado -eq 1 ]]; then
    contribution_status="REVIEWED"
  elif [[ $has_aprovado -eq 1 ]]; then
    contribution_status="APPROVED"
  elif [[ $has_sent -eq 1 ]]; then
    contribution_status="SENT"
  elif [[ $has_mergeado -eq 1 ]]
    contribution_status="MERGED"
  fi

  condition_array=(['id']="$contribution_id")
  set_contribution_status "$contribution_id" "$contribution_status"

  return 0
}
```

Atualização Manual de Status

Além da atualização automática baseada em heurísticas, é desejável oferecer ao usuário a possibilidade de ajustar manualmente o status de um patch. Essa funcionalidade permitiria corrigir interpretações equivocadas da heurística, lidar com casos excepcionais e manter controle total sobre o histórico de evolução de cada contribuição. Uma interface de atualização manual complementaria a lógica automática sem substituí-la, fornecendo maior flexibilidade operacional (Figura 3.12).

3.3.4 Próximos Passos

Atualmente, o *patch-track* encontra-se estruturado com um objetivo claro e funcionalidades essenciais de rastreamento. No entanto, para evoluir para uma versão de uso pleno, é fundamental expandir a liberdade de edição e a flexibilidade da ferramenta, permitindo que o usuário gerencie o ciclo de vida das informações de forma mais autônoma. Entre os pontos de desenvolvimento futuros, destacam-se:

Flexibilidade na Gestão de Dados e Experiência do Usuário

Permitir mais flexibilidade do usuário para editar dados registrados, como, por exemplo:

```
joao-souza@joao-souza:~/kworkflow$ kw patch-track --show-contributions -c2
-------------------------------------monografia_demo_test-------------------------------------
ID: 2
Send-By: joaosouzaaa12@usp.br
Date: 2025-12-21 23:21:56
Status: APPROVED
Repository: (id: 1) monografia_demo_test - git@github.com:JGBSouza/monografia_demo_test.git
Maintainer: JGBSouza - joaosouzaaa12email.teste@gmail.com
-----------------------------------------------------------------------------------------------
-------------------------------------Submissions-infos-----------------------------------------
ID    |Send By                                                        |Date
-----------------------------------------------------------------------------------------------
2     |joaosouzaaa12@usp.br                                           |2025-12-21 23:21:56
3     |joaosouzaaa12@usp.br                                           |2025-12-21 23:22:20
-----------------------------------------------------------------------------------------------
---------------------------------Last-submission-patches-----------------------------------------
ID    |Date                 |Status    |Title
-----------------------------------------------------------------------------------------------
2     |2025-12-21 23:21:56  |APPROVED  |[PATCH] feat: adiciona arquivo base inicial helloWord
-----------------------------------------------------------------------------------------------
joao-souza@joao-souza:~/kworkflow$ kw patch-track --set-status 'MERGED' --id 2
Patch status updated successfully.
joao-souza@joao-souza:~/kworkflow$ kw patch-track --show-contributions -c2
-------------------------------------monografia_demo_test-------------------------------------
ID: 2
Send-By: joaosouzaaa12@usp.br
Date: 2025-12-21 23:21:56
Status: APPROVED
Repository: (id: 1) monografia_demo_test - git@github.com:JGBSouza/monografia_demo_test.git
Maintainer: JGBSouza - joaosouzaaa12email.teste@gmail.com
-----------------------------------------------------------------------------------------------
-------------------------------------Submissions-infos-----------------------------------------
ID    |Send By                                                        |Date
-----------------------------------------------------------------------------------------------
2     |joaosouzaaa12@usp.br                                           |2025-12-21 23:21:56
3     |joaosouzaaa12@usp.br                                           |2025-12-21 23:22:20
-----------------------------------------------------------------------------------------------
---------------------------------Last-submission-patches-----------------------------------------
ID    |Date                 |Status    |Title
-----------------------------------------------------------------------------------------------
2     |2025-12-21 23:21:56  |MERGED    |[PATCH] feat: adiciona arquivo base inicial helloWord
-----------------------------------------------------------------------------------------------
```

Figura 3.12: *Atualização manual do status de um patch via Kw patch-track*

- **Renomeação de Contribuições:** Implementar a capacidade de renomear contribuições existentes. Como o nome da contribuição possui valor apenas para a organização pessoal do desenvolvedor, essa funcionalidade permitiria correções após a criação sem impactar a lógica técnica do sistema.

- **Realocação de Submissões:** Permitir que o usuário mova submissões de uma contribuição para outra. Esta melhoria de experiência do usuário (UX) é vital para corrigir erros de identificação ocorridos durante o envio de patches via `kw send-patch`, garantindo que o histórico de evolução reflita o agrupamento pretendido.

- **Gestão de Mantenedores e Repositórios:** Expandir as capacidades de edição e exclusão para as entidades de mantenedores e repositórios. Isso inclui a alteração de metadados (como e-mails e URLs) e a remoção de registros obsoletos ou duplicados, mantendo a base de dados limpa e atualizada conforme a rotatividade orgânica dos subsistemas do *kernel*.

- **Edição de Metadados de Contribuição:** Oferecer uma interface para ajustar informações vinculadas à contribuição após sua criação, como a mudança do repositório de destino ou do mantenedor associado, conferindo maior resiliência a mudanças de contexto no fluxo de trabalho.

Automatização do Rastreio de Integração (*Merge*)

Implementar mecanismos para localizar automaticamente o ponto de integração definitiva de um patch no histórico de ramos (*branches*) do repositório. Como o ciclo de vida de uma contribuição no *kernel* é concluído apenas com o *merge* em árvores estáveis ou de subsistemas, essa funcionalidade permitiria que o *patch-track* fechasse o ciclo de monitoramento de forma autônoma, informando ao usuário precisamente em qual versão do código sua contribuição foi incorporada.

Aprimoramento da Corretude das Heurísticas de Aprovação

Evoluir a lógica de análise de mensagens para aumentar a confiabilidade na transição de estados. Atualmente baseada em buscas por padrões textuais, a heurística deve ser refinada para validar se o autor de uma mensagem de aprovação (como *Acked-by* ou *Reviewed-by*) corresponde ao mantenedor previamente associado ao repositório. Adicionalmente, prevê-se a integração do *kw* com plataformas de revisão e bancos de dados de listas de discussão (como o *lore.kernel.org* ou instâncias do *Patchwork*), permitindo o uso de metadados estruturados e marcadores específicos que garantam um julgamento de status imune a falso-positivos de conversas casuais nas *threads* de e-mail.

Segmentação de Contexto via *Mailboxes* Específicas:

Implementar o suporte à organização de mensagens em caixas de correio (*mailboxes*) dedicadas exclusivamente aos fluxos de cada contribuição ou projeto. Atualmente, a integração com o *mutt* pode exigir a varredura de caixas de entrada genéricas com volumes massivos de dados, típicos de listas de discussão do *kernel*. Ao viabilizar o isolamento das comunicações em pastas específicas, o *patch-track* permitiria que o *mutt* operasse com um contexto de dados reduzido, otimizando significativamente a performance de indexação e a agilidade da interface. Além disso, essa delimitação de escopo aumentaria a eficiência das heurísticas de análise automática, que passariam a processar apenas mensagens previamente filtradas e relevantes ao histórico do usuário.

3.3.5 Resultados

Com a introdução do *kw patch track*, o processo de contribuição via *kw* deve tornar-se mais organizado e automatizado. A ferramenta deve permitir acompanhar o ciclo de vida de cada patch de forma centralizada, eliminando a necessidade de acompanhamento manual e reduzindo o risco de perda de informações, apresentando maior clareza e rastreabilidade no fluxo de revisões, economia de tempo no acompanhamento de submissões, histórico completo e versionado de cada contribuição, uma base estruturada para análise estatística e integração futura com outras ferramentas, além de um ambiente mais unificado para colaboração no kernel, reduzindo dependências de outras ferramentas, como softwares gerenciadores de email.

Capítulo 4

Considerações Finais

Esse trabalho apresentou de forma geral, uma análise sobre o kernel Linux, explorando a sua importância e relevância no cenário, suas etapas de desenvolvimento, denominadas por Feitelson (Feitelson, 2012) como modelo de desenvolvimento perpétuo até o lançamento de suas versões para o usuário final, os *stable kernels*. Nesse processo, também foi discutido a relevância do seu desenvolvimento como software livre, o que influencia significativamente a sua segmentação em diversos componentes e o seu modelo de contribuição para permitir a colaboração de uma comunidade de desenvolvedores em sua implementação. Ainda nessa análise, evidencia-se também a complexidade que o sistema adquiriu ao longo dos anos, exigindo uma grande carga de conhecimento técnico e prático antes que possam de fato desenvolver para o sistema.

Nesse contexto, diversas ferramentas surgem de forma a mitigar as dificuldades associadas à esse fluxo de contribuição. Dentre essas ferramentas, o Kworkflow se destaca pela busca em oferecer uma interface única e integrada para todas essas dificuldades. Para isso, o kw é construído como um hub modular, integrando funcionalidades locais e externas, oferecendo suporte tanto a tarefas práticas — como a compilação e o deploy de versões do kernel — quanto a processos indiretos, como a submissão de patches, através de comandos de terminal.

Entretanto, compreender e automatizar integralmente o ciclo de contribuição ainda constitui um desafio que o kw busca superar. Dentre as dificuldades ainda não mapeadas, uma das etapas mais críticas é o gerenciamento de patches após a submissão, período em que as contribuições passam por revisões e discussões por parte dos mantenedores e da comunidade de desenvolvedores. Dada a insuficiência das ferramentas de gerenciamento de versão em suprir as necessidades de um software com as dimensões do kernel Linux, hoje, as contribuições para a ferramenta são submetidas através de listas de email, o que representam dificuldades ainda mais significativas para este processo, como a dependência de ferramentas externas não gerenciáveis, a baixa rastreabilidade e controle das submissões, escalabilidade limitada, além de representar uma ruptura no fluxo de desenvolvimento, que o kw pretende englobar.

Este trabalho dá continuidade ao desenvolvimento do Kworkflow, integrando-se à linha de evolução de projetos anteriores, como Simplificando o processo de contribuição

para o kernel Linux (Neto, 2022) e Integrating the Kworkflow system with the Lore archives (Barros Tadokoro, 2023). Esses trabalhos estabeleceram as bases estruturais do sistema, consolidando o uso de um banco de dados interno, a automatização do envio de patches por e-mail e a integração com os arquivos de discussão oficiais do kernel, sobre as quais o presente estudo se apoia. Ao compreender a etapa de revisão dos patches, esse trabalho integra de forma completa o processo de gestão de contribuições, permitindo análises holísticas do processo, alinhando-se para permitir o caráter de software científico almejado pelo kw.

A primeira contribuição foca na melhoria do sistema de CRUD do banco de dados. Apesar do Kworkflow já contar com funções que permitiam interação com o SQLite3, essas funções não isolavam suficientemente o código das instruções de acesso aos dados, exigindo inserção de trechos SQL (por exemplo, uso de cláusulas *WHERE*) diretamente nos comandos de seleção. Para corrigir isso, os comandos de leitura foram parametrizados para aceitar as buscas com o parâmetro de especificação *WHERE*, ordenação dos resultados com o uso do parâmetro *ORDER BY* e limite de itens na resposta com o parâmetro *LIMIT*, além da refatoração para permitir que tanto o processo de remoção aceitasse comparações além da igualdade (>, <, >=, <=, !=) e combinações de critérios mais complexas. Além disso, foi incorporado também o método *update_into* para permitir alterações pontuais de atributos em uma entidade essas mudanças melhoraram o fluxo de interação com o banco de dados, viabilizando implementações subsequentes para esse trabalho, como o *kw manage contacts* e o *kw patch track*, que dependem de buscas parametrizáveis, ordenação e limites para operar corretamente.

A segunda contribuição, o *kw manage-contacts*, é uma ferramenta de suporte, permitindo a coordenação de grupos e contatos de contribuidores. Ainda que a ferramenta já possuísse suporte para envio das submissões para mantenedores responsáveis através do comando get_mantainers, isso não atende alguns grupos não oficiais, como, por exemplo, colegas de trabalho ou outros grupos externos envolvidos com o patch. Como solução, a ferramenta kw_manage contacts foi desenvolvida, utilizando-se do sistema de banco de dados para armazenar os dados dos contatos e grupos criados pelo usuário bem como das suas relações, denominando de quais grupos cada contato faz parte. Essa ferramenta também se integra diretamente com o sistema de submissões de patches, *kw manage-contacts*, permitindo, através dos comandos *to-groups* e *cc-groups*, que grupos sejam passados como parâmetro de submissão, garantindo uma submissão muito mais simples e coesa através do terminal de comandos, garantindo corretude nas submissões ao evitar que os contatos precisem ser digitados um a um de forma manual.

Com foco mais específico no problema das listas de e-mail como principal método de contribuição para o kernel Linux, a terceira implementação deste trabalho, o kw patch-track, funciona como uma camada de gerenciamento local das submissões do usuário. O módulo foi projetado para operar de forma integrada ao fluxo de envio do `kw send-patch`, registrando as submissões e utilizando técnicas de raspagem de dados (*data scraping*) nos arquivos temporários gerados durante o processo para extrair metadados e persisti-los no banco de dados.

Ainda em estágio de refinamento de heurísticas, a ferramenta oferece um sistema de atualização automática de estados através do comando `kw patch-track –update`. Este

comando processa as mensagens recebidas para identificar o estágio atual de cada patch, embora a corretude total dessas transições dependa da evolução dos algoritmos de análise textual. Complementarmente, a integração com o *mutt* via comando `kw patch-track -open-contribution` viabiliza uma interface terminal para que o usuário visualize e responda a revisões de forma contextualizada. Esta arquitetura visa mitigar a dependência de ferramentas externas e manuais no acompanhamento de contribuições, fornecendo uma base sólida para a extração de dados estruturados, ainda que o escopo atual do projeto preveja futuras expansões para suportar maior granularidade no histórico de interações e na precisão dos julgamentos automáticos.

A introdução dessas ferramentas ao Kworkflow traz impactos significativos tanto para contribuidores experientes quanto para novos participantes do desenvolvimento do kernel Linux. Ao centralizar operações que antes dependiam de múltiplos processos externos — como consultas manuais de listas de e-mail, organização de grupos de contatos e acompanhamento de revisões de patches — o sistema reduz o overhead de contribuintes recém-chegados, simplificando a configuração inicial e a compreensão do fluxo de submissão. O *kw manage-contacts* garante que grupos de destinatários recorrentes possam ser aplicados automaticamente, evitando erros manuais e agilizando a comunicação, enquanto o *kw patch-track* oferece rastreabilidade completa das contribuições, permitindo que o usuário visualize o histórico de submissões, respostas e revisões sem recorrer a ferramentas externas.

De forma geral, essas implementações ajudam a consolidar a proposta do Kworkflow de oferecer uma solução unificada e integrada para o ciclo de contribuição ao kernel Linux, promovendo maior previsibilidade, consistência e eficiência. Ao reduzir tarefas repetitivas e centralizar informações críticas, as ferramentas aumentam a produtividade, diminuem a curva de aprendizado e fornecem uma base sólida para automações e análises futuras. Dessa maneira, o trabalho não apenas melhora a experiência do desenvolvedor individual, mas também fortalece o ecossistema colaborativo do kernel, evidenciando a importância de soluções que conectem de forma coerente os diversos elementos do processo de desenvolvimento em um fluxo contínuo e gerenciável.

Referências

[Avatavului *et al.* 2023] Cristian Avatavului *et al.* “Open-source and closed-source projects: a fair comparison”. *Journal of Information Systems & Operations Management* 17.2 (dez. de 2023) (citado na pg. 5).

[Barros Tadokoro 2023] David de Barros Tadokoro. *Integrating the kworkflow system with the lore archives: enhancing the linux kernel developer interaction with mailing lists*. Monografia Final. Institute of Mathematics and Statistics, Bachelor of Computer Science. Supervisor: Paulo Meirelles. Co-supervisor: Rodrigo Siqueira. 2023 (citado nas pgs. 3, 13, 17, 50).

[Devineni 2020] Siva Karthik Devineni. “Version control systems (vcs) the pillars of modern software development: analyzing the past, present, and anticipating future trends”. *International Journal of Science and Research* 9.12 (2020), pp. 1816–1829. doi: 10.21275/SR24127210817 (citado nas pgs. 13, 14).

[Feitelson 2012] Dror G. Feitelson. “Perpetual development: a model of the linux kernel life cycle”. *Journal of Systems and Software* 85.4 (2012), pp. 859–875. url: https://www.cs.huji.ac.il/~feit/papers/LinuxDev12JSS.pdf (citado nas pgs. 2, 7, 49).

[Foundation 2023] Free Software Foundation. *A Definição de Software Livre*. Acesso em: 22 dez. 2025. 2023. url: https://www.gnu.org/philosophy/free-sw.pt-br.html (citado na pg. 5).

[Greg Kroah Hartman 2016] Greg Kroah Hartman. *Kernel Recipes 2016 - Patches carved into stone tablets… - Greg KH*. Acessado em: 03 set. 2025. 2016. url: https://youtu.be/L8OOzaqS37s?si=zPnIcyGIlu7lIcmK (acesso em 03/09/2025) (citado nas pgs. 3, 15).

[Kroah-Hartman 2018] Greg Kroah-Hartman. *Linux Kernel Development*. Acesso em: 12 dez. 2021. 2018. url: https://github.com/gregkh/kernel-development/blob/bd8d3673b33fa641d06046fa4bff103f78ec4e89/kernel-development.pdf (citado nas pgs. v, 9).

[Kroah-Hartman 2025] Greg Kroah-Hartman. *kernel-history: Linux kernel history logs and stats*. Repositório GitHub. Acesso em: 06 set. 2025. 2025. url: https://github.com/gregkh/kernel-history (citado na pg. 14).

[Neto 2022] Rubens Gomes Neto. *Simplificando o processo de contribuição para o kernel linux: a evolução da ferramenta kernelworkflow*. Monografia Final. MAC 499 — Trabalho de Formatura Supervisionado. Supervisor: Paulo Meirelles. Cossupervisor: Rodrigo Siqueira. 2022 (citado nas pgs. 3, 17, 50).

[Passos *et al.* 2025] Rafael Passos, Arthur Pilone, David Tadokoro e Paulo Meirelles. " Streamlining Analyses on the Linux Kernel with DUKS ". In: *2025 IEEE Working Conference on Software Visualization (VISSOFT)*. Los Alamitos, CA, USA: IEEE Computer Society, set. de 2025, pp. 125–128. doi: 10.1109/VISSOFT67405.2025.00025. url: https://doi.ieeecomputersociety.org/10.1109/VISSOFT67405.2025.00025 (citado na pg. 1).

[Silberschatz *et al.* 2018] A. Silberschatz, P.B. Galvin e G. Gagne. *Operating System Concepts*. Wiley, 2018. isbn: 9781119124894. url: https://books.google.com.br/books?id=FHJlDwAAQBAJ (citado na pg. 1).

[Tadokoro *et al.* 2025] David Tadokoro, Rodrigo Siqueira e Paulo Meirelles. "Kworkflow: a linux kernel developer automation workflow system". In: *Proceedings of the Free Software Competence Center, Institute of Mathematics and Statistics, University of São Paulo*. University of São Paulo. São Paulo, Brazil, 2025 (citado nas pgs. 12, 14).

[Tan *et al.* 2020] Xin Tan, Minghui Zhou e Brian Fitzgerald. "Scaling open source communities: an empirical study of the linux kernel". In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. ACM / IEEE Computer Society, 2020, pp. 1222–1234. doi: 10.1145/3377811.3380920 (citado na pg. 6).

[Tanenbaum e Bos 2023] A.S. Tanenbaum e H. Bos. *Modern Operating Systems, Global Edition*. Pearson Education, 2023. isbn: 9781292727899. url: https://books.google.com.br/books?id=cHa2EAAAQBAJ (citado na pg. 1).

[The Linux Kernel Documentation 2023] The Linux Kernel Documentation. *How the development process works*. Parte de "A guide to the Kernel Development Process" — versão v4.14. 2023. url: https://www.kernel.org/doc/html/v6.17/process/2.Process.html (acesso em 04/09/2025) (citado nas pgs. 7, 8).

[Torvalds 1991a] L. Torvalds. *Free minix-like kernel sources for 386-AT*. Email enviado para o newsgroup comp.os.minix. Arquivado em LWN.net: https://lwn.net/2001/0823/a/lt-release.php3. Out. de 1991 (citado na pg. 1).

[Torvalds 1991b] L. Torvalds. *What would you like to see most in minix?* Email enviado para o newsgroup comp.os.minix. Arquivado em LWN.net: https://lwn.net/2001/0823/a/lt-announcement.php3. Ago. de 1991 (citado na pg. 1).