

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

*O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)*

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.

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

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.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de abreviaturas

Lista de figuras

2.1	Arquitetura conceitual do kw <i>Fonte: tadokoro2025kworkflowSBES</i>	10
3.1	Exemplo do comando “group_show” sem um grupo especificado.	31
3.2	Exemplo do comando “group_show” para um grupo específico.	31

Lista de tabelas

2.1	comandos do kw. Fonte: Reproduzido de tadokoro2025kworkflowSBES	11
-----	--	----

Lista de programas

3.1	código remove_from antigo	16
3.2	código remove_from novo	17
3.3	código select_from antigo	17
3.4	código select_from novo	18
3.5	código generate_where_clause utilizado nas novas funções para gerar a cláusula WHERE SQL a partir dos parâmetros passados	19
3.6	snippet uso função select_from_antiga	20

3.7	snippet uso função <code>select_from_nova</code>	20
3.8	código <code>update_into</code>	20
3.9	código <code>generate_set_clause</code> utilizado para permitir especificar quais atributos serão alterados e quais serão seus novos valores	21
3.10	snippet uso função <code>update_into</code>	22
3.11	comandos kw <code>manage contacts</code>	23
3.12	<code>create_email_group</code> e <code>create_group</code>	23
3.13	<code>remove_email_group</code> e <code>remove_group</code>	25
3.14	<code>rename_email_group</code> e <code>rename_group</code>	26
3.15	<code>add_email_contacts</code> e <code>add_contact_group</code>	27
3.16	<code>show_email_groups</code> , <code>print_groups_infos</code> e <code>print_contacts_infos</code>	29
3.17	opções do comando <code>-send</code> do kw <code>send-patch</code> contendo o <code>to-groups</code> e o <code>cc-groups</code>	32
3.18	Função <code>send_patch_main</code> com métodos <code>-to-groups</code> e <code>cc-groups</code>	32

Sumário

Introdução	1
1 Fundamentação Teórica	5
1.1 Software livre	5
1.1.1 Contribuição em Software Livres	6
1.2 O Kernel Linux	6
1.2.1 O Modelo de desenvolvimento do Kernel Linux	7
1.2.2 Contribuindo para o Kernel Linux	7
2 Kernel Workflow	9
2.0.1 Arquitetura	9
2.0.2 Soluções	11
2.0.3 O problema da contribuição no desenvolvimento de software livre	12
3 Contribuições	15
3.1 CRUD banco de dados	15
3.2 KW Manage Contacts	22
3.2.1 Objetivos	22
3.2.2 Arquitetura	23
3.2.3 Funcionalidades	23
3.2.4 Enviar patches para grupos	31
3.2.5 Resultados	33
3.3 KW Patch track	34
3.3.1 Objetivos	34
3.3.2 Arquitetura	34
3.3.3 Integração com o mutt	35
3.3.4 Funcionalidades	35
3.3.5 Resultados	36

4	Considerações Finais	37
	Referências	41

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 (1) abstrai a complexidade do hardware específico que compõe as máquinas e (2) gerencia os recursos que tais hardwares representam (TANENBAUM e BOS, 2023), possibilitando a interação final entre ser humano e computador. Em sua composição, os sistemas operacionais são divididos em diversos componentes específicos, dentre os quais o kernel é considerado a parte central. Esse fato decorre principalmente das responsabilidades atribuídas a ele, que incluem a gestão da alocação de recursos entre programas em execução, a priorização de atividades críticas e o gerenciamento da comunicação entre periféricos e o sistema, permitindo que o computador processe ações do usuário por meio dos dispositivos de entrada, coordene o processamento interno dessas atividades e, por fim, apresente os resultados de forma significativa ao usuário, através dos dispositivos de saída.

Entre os diversos sistemas de kernel existentes, o Linux, desenvolvido por Linus Torvalds em 1991, destaca-se como um dos mais relevantes. Assim como outros softwares livres, o Linux surgiu como uma alternativa à hegemonia dos softwares proprietários, 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, contando com diversas distribuições e sendo responsável por aproximadamente 57

Para sustentar esse ciclo contínuo de desenvolvimento, o kernel Linux adota um modelo rigoroso descrito por Feitelson como *perpetual development*, 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, corresponde ao período em que as contribuições dos desenvolvedores são enviadas aos mantenedores — integrantes da comunidade responsáveis pela administração das submissões e pela verificação da conformidade com os padrões do projeto. A partir dessas integrações, é lançada uma versão inicial do novo kernel, 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 é oficialmente lançada, e uma equipe reduzida passa a atuar na manutenção contínua, liberando novas correções enquanto uma nova janela de mesclagem é aberta.

Assim como o kernel, os patches incorporados durante a janela de mesclagem também 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 Kernel Workflow (KW), uma ferramenta de software livre desenvolvida majoritariamente em bash, que tem como objetivo oferecer uma solução unificada para os diversos desafios enfrentados pelos contribuidores do kernel. Para isso, o KW integra e simplifica automações amplamente consolidadas na comunidade, como git, lore e b4, criando soluções locais quando necessário. A ferramenta organiza-se como um hub, 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. 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 desenvol-

vimento. 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

1.1 Software livre

No mercado computacional, dois conceitos principais dominam o cenário no que se refere ao desenvolvimento como software proprietário ou software livre. Em geral, define-se como software proprietário, o software que é desenvolvido de maneira privada, com apenas a aplicação sendo acessível para usuários. Enquanto que, softwares livres são projetos cujo código fonte é de livre acesso para qualquer um que queira estudar.

Historicamente, o mercado computacional era dominado por grandes corporações, que detinham o monopólio do processo e conhecimento e recursos necessários para o desenvolvimento e 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).

Essa abordagem possui 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 contraditória, 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, softwares livres e proprietários 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 Contribuição em Software Livres

Em projetos de software livre, para que a gestão das contribuições seja possível, os sistemas geralmente contam com uma equipe mantenedores, grupo interno de desenvolvedores que possuem responsabilidade geral pelo código principal. Desse modo, para que sejam integradas, as contribuições enviadas 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 um *patch* é necessário, 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 a contribuição seja feita de maneira correta, os *patches* 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 Fonte: <https://w3techs.com/technologies/details/os-linux>

1.2.1 O Modelo de desenvolvimento do Kernel Linux

Ainda que tenha sido lançado há mais três décadas, novas versões do kernel Linux continuam sendo lançadas até hoje. Essas versões, chamadas “kernels estáveis”, são construídas, em parte, através da contribuição de diversos desenvolvedores ao redor do mundo, através da submissão de *patches* 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 *perpetual development*, 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:

Durante a primeira etapa, a maior parte das alterações serão integradas à nova versão do kernel. Essas mesclagens ocorrem a partir de *patches* que foram previamente preparados, testados e coletados. Com base nessa nova versão, o primeiro kernel RC (Release Candidate) será lançado, encerrando a janela de mesclagem e iniciando a próxima etapa.

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, superados por versões anteriores e reintroduzidos durante a janela de mesclagem, são corrigidas.

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. Por conta desse fato, após a criação da versão estável, uma equipe de desenvolvedores é designada para a terceira etapa, a manutenção contínua, lançando novas atualizações ocasionais com correções 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 conjunto 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 e de redes sem fio. Desse modo, além dos *patches* recebidos diretamente por contribuidores, os mantenedores podem receber também *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, 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 serão acumuladas e testadas previamente. Posteriormente, para que as novas versões sejam construídas, durante a janela de mesclagem, são construídas as árvores-next, árvores de integração ramificadas da mainline que integrará as novas submissões dispersas nas árvores de sistema para a nova release. 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 venha a ser integradas na árvore principal, mainline, administrada diretamente por Linux Trovald, a versão oficial do kernel.

Assim como o kernel, os *patches* que são incorporados durante a janela de mesclagem precisam ser previamente preparados e, durante esse preparo, passam por algumas etapas. Esse processo, ainda que informal, serve para garantir que cada *patch* possa ser revisado e tenha sua qualidade garantida antes que a alteração seja incorporada ao kernel principal ([THE LINUX KERNEL DOCUMENTATION, 2023](#)).

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

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 essas ferramentas, o Kernel Workflow - KW, 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 um contribuidor pode vir a encontrar. Promovendo um ambiente mais simples e rápido de desenvolvimento, reduzindo a carga de conhecimento prévio necessária para futuros contribuidores além de consolidar um meio pelo qual seja possível medir de forma precisa o ciclo de contribuição do desenvolvedor do kernel, possibilitando que ainda mais soluções possam ser planejadas e que o impacto real das soluções já empregadas seja medido.

2.0.1 Arquitetura

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

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 serve como um hub, sendo o responsável por receber os comandos iniciais dos usuários, digitados no terminal, e redirecionar a execução para a ferramenta especificada.

2. Componentes: Cada ferramenta do kw possui um arquivo principal, contendo o processamento central do comando do usuário, a lista de comandos específicos para aquela ferramenta e uma sessão de ajuda, que serve para guiar novos usuários em caso de dúvidas.
3. Bibliotecas: Durante o desenvolvimento, muitos dos códigos criados para permitir a execução dos comandos de uma ferramenta podem ser compartilhados em locais diversos e/ou entre ferramentas. Dessa forma, o kw utiliza-se de um esquema de bibliotecas, que são implementações de soluções de maneira genérica que podem ser reutilizados em todo o código. Em geral, essas implementações são agrupadas em arquivos por similaridade do contexto das soluções, por exemplo, bibliotecas para manipulação de textos, para operações no banco de dados ou manipulação de elementos como data e hora.
4. Plugins: Adicionalmente, existem seções de códigos na implementação das ferramentas que, por dependerem de contextos ou implementações externas, são pouco reutilizáveis e/ou muito voláteis, devido ao fato do desenvolvimento imprevisível desses ambientes. Assim, esses códigos são isolados em arquivos específicos, chamados plugins, de modo que o código principal fique isolado e não precise passar por alterações constantes, aproveitando apenas dos métodos declarados nesses plugins independentemente da forma como estão implementados no momento.
5. Documentação: Para manter registro das implementações e todas as outras informações necessárias para atuais e novos colaboradores, o kw também mantém um sistema de documentação, utilizado também para a construção do blog da ferramenta.

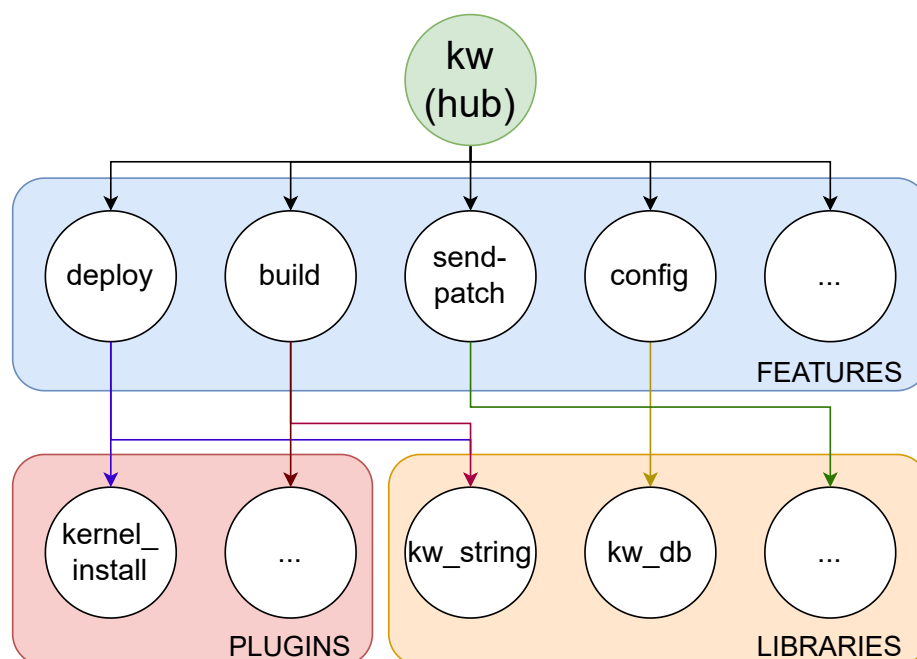


Figura 2.1: Arquitetura conceitual do kw

Fonte: [tadokoro2025kworkflowSBES](#)

2.0.2 Soluções

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 **tadokoro2023kwlore**, as automações desenvolvidas para o kw podem ser tanto práticas, que afetam diretamente a implementação para o kernel, como, por exemplo, o kw build e kw deploy, utilizados, respectivamente para criação e aplicação da imagem do kernel com as alterações do desenvolvedor, ou indiretas, afetando o fluxo de desenvolvimento como um todo, como no caso do kw send-patch ou do kw-patch hub, ferramentas utilizadas, respectivamente, para submissão de patches e consulta de patches do 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 implementações existentes na ferramenta, são:

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 **tadokoro2025kworkflowSBES**

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 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.

2.0.3 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 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*,¹ *gerrit*² 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, 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.

¹ <https://github.com>

² <https://www.gerritcodereview.com>

Capítulo 3

Contribuições

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 **gomes2022kernelworkflow**, 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 **tadokoro2023kwlore**, que implementa o *patch-hub* – interface gráfica 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 foca em oferecer melhorias 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 features, 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 queries.

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 queries, tornando necessário o uso de comandos SQL em alguns casos, como no comando de seleção, que recebiam trechos de comando SQL com a cláusula *where*, que serve para especificar quais critérios devem atender os parâmetros que serão selecionados ou removidos. Além disso, especificamente nos comandos de seleção, também foram inseridas alterações que permitem, para essa função, também foi implementado um parâmetro adicional para permitir o uso da cláusula *ordered_by*, que permite especificar uma ordenação para os dados retornados com base em um atributo comparável entre eles. Por fim, essa alteração também permitiu que comparações de desigualdades fossem feitas de forma mais abrangente e ordenada, visto que na função de remoção apenas operações de comparação eram possíveis e que na função de seleção apenas com o código SQL explícito:

Programa 3.1 código remove_from antigo

```
# This function removes every matching row from a given table.
#
# @table:      Table to replace/insert data into
# @_condition_array: An array reference of condition pairs
#               <column,value> to match rows
# @db:         Name of the database file
# @db_folder:  Path to the folder that contains @db
#
# Return:
# 2 if db doesn't exist;
# 22 if empty table, columns or values are passed;
# 0 if succesful.
function remove_from()
{
    local table="$1"
    local -n _condition_array="$2"
    local db="${3:-"${DB_NAME}"}"
    local db_folder="${4:-"${KW_DATA_DIR}"}"
    local flag=${5:-'SILENT'}
    local where_clause=''
    local db_path
    db_path="$(join_path "${db_folder}" "$db")"
    if [[ ! -f "${db_path}" ]]; then
        complain 'Database does not exist'
        return 2
    fi
}
```

cont →

```

→ cont
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 3.2 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"
}

```

Programa 3.3 código select_from antigo

```

# This function gets the values in the table of given database
#
# @flag:      Flag to control function output

```

cont →

→ *cont*

```
# @table:      Table to select info from
# @columns:    Columns of the table to get
# @pre_cmd:    Pre command to execute
# @order_by:   List of attributes to use for ordering
# @db:         Name of the database file
# @db_folder:  Path to the folder that contains @db
#
# Return:
# 2 if db doesn't exist; 22 if table is empty
# 0 if succesful; non-zero otherwise
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 3.4 código select_from novo

```
function select_from()
{
    local table="$1"
    local columns="${2:-"*}"
    local pre_cmd="$3"
    local _condition_array="$4"
```

cont →

```

→ cont
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 3.5 código generate_where_clause utilizado nas novas funções para gerar a cláusula WHERE SQL a partir dos parâmetros passados

```

# This function receives a condition_array and then generate
# the infos that will be used by the WHERE clause to specify
# the data whe want.
#
# @condition_array_ref: The condition array containing the conditions
#
# Returns:
# A string containing the generated clause
function generate_where_clause()
{
    local -n condition_array_ref="$1"
    local clause
    local relational_op='='
    local attribute
    local where_clause="WHERE "

```

cont →

```

→ cont
local value

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 3.6 snippet uso função select_from_antiga

```

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

```

Programa 3.7 snippet uso função select_from_nova

```

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

```

Além disso, outra implementação desenvolvida nessa etapa, foi a implementação do novo método *update_into*, que permitia a alteração pontual de algum atributo dentro de uma entidade do banco de dados e da função *generate_set_clause*, utilizada para gerar a clausula set, 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.

Programa 3.8 código update_into

```

# This function update the set of values in the table of given database
# with the given conditions.
#
# @table:      Table to select info from
# @set:       An array reference of condition pairs that will be
# updated in the db
# @pre_cmd:    Pre command to execute
# @condition_array: An array reference of condition pairs
# specifying the data that will be updated
# @flag:       Flag to control function output
# @db:         Name of the database file

```

cont →

→ *cont*

```
# @db_folder: Path to the folder that contains @db
#
# Return:
# 2 if db doesn't exist; 22 if table is empty
# 0 if succesful; non-zero otherwise
function update_into()
{
    local table="$1"
    local _updates_array="$2"
    local pre_cmd="$3"
    local _condition_array="$4"
    local flag=${5:-'SILENT'}
    local db="${6:-"$DB_NAME"}"
    local db_folder="${7:-"$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 [[ -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

```
# This function receives a condition_array and then generate
# the infos that will be used by the SET clause to update
# the data fields whe want.
#
```

cont →

```

→ cont
# @condition_array_ref: The condition array containing the conditions
#
# Returns:
# A string containing the generated clause
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'

```

3.2 KW Manage Contacts

3.2.1 Objetivos

No fluxo atual, patches podem ser submetidos através do kw com o uso da ferramenta *kw send-patch*, que recebe como dados a lista de commits que deverão ser enviados e a lista de usuários, em geral, mantenedores, que deve ser notificada da submissão do patch. A partir desse comando, um patch será gerado com as alterações especificadas e, através do *git email*, ferramenta de email's do *github* os usuários receberão um email com as implementações do usuário.

Ao lidar com e-mails, é sempre comum que hajam grupos de pessoas que sempre serão endereçadas durante a submissão. Atualmente, o kw fornece uma funcionalidade, o *get_maintainers* que lista mantenedores responsáveis pelos sistemas alterados, facilitando a identificação do ou dos responsáveis, o que, porém, não atende alguns grupos não oficiais, como, por exemplo, colegas de trabalho ou outros grupos externos envolvidos com o patch. Dessa forma, havia a necessidade de uma ferramenta que pudesse oferecer um sistema de gerenciamento de grupos de e-mail integrado ao fluxo do kw, garantindo maior praticidade e consistência na comunicação.

Assim, o objetivo principal foi o de criar um sistema que permitisse gerenciar grupos de e-mail de forma centralizada, com armazenamento persistente em banco de dados 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 mail.

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.

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 incluem:

Programa 3.11 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

Essa função recebe o nome de um novo grupo e realiza a validação antes de sua criação. A validação consiste em verificar se o nome já não existe no banco de dados, se não contém caracteres especiais e se o tamanho é inferior a 50 caracteres. Caso todas as condições sejam atendidas, o grupo é criado e persistido no banco com sucesso.

Programa 3.12 create_email_group e create_group

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

cont →

```

    → cont
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'
    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

Essa função remove um grupo de e-mails específico e todas as suas referências no banco de dados. Nesse caso, é verificada apenas a existência do grupo, uma vez que, se já foi previamente adicionado, deve atender aos critérios de validação. A remoção utiliza a cláusula CASCADE na tabela de associação *email_contact_group*, garantindo que todas as

relações existentes para aquele grupo sejam automaticamente excluídas. Além disso, os contatos que permanecerem sem associação a grupos também são removidos.

Programa 3.13 remove_email_group e remove_group

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

    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

Essa função realiza a renomeação de um grupo de e-mails existente, recebendo como parâmetros o nome atual e o novo nome desejado. Antes da atualização, o novo nome do grupo passa novamente pelo processo de validação, garantindo que não corresponda a um grupo já existente no banco de dados e que siga as mesmas regras de restrição aplicadas na

criação — como não conter caracteres especiais e ter comprimento inferior a 50 caracteres. Se as condições forem atendidas, o nome é atualizado com sucesso.

Programa 3.14 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"
    fi
}
```

cont →

```

→ cont
return 22 # EINVAL
elif [[ "$ret" -ne 0 ]]; then
    complain "($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

Essa função é responsável por associar novos contatos a um grupo de e-mails existente. Ela recebe como parâmetros o nome do grupo e uma lista de contatos no formato “NOME_CONTATO <EMAIL_CONTATO>, NOME_CONTATO <EMAIL_CONTATO>, ...”. Inicialmente, a função valida a existência do grupo no banco de dados. Em seguida, divide a lista de contatos e executa verificações individuais para cada um, como a validade do endereço de e-mail e a presença de um nome associado. Após essas validações, os contatos são inseridos no banco de dados e vinculados ao grupo correspondente.

Programa 3.15 add_email_contacts e add_contact_group

```

function add_email_contacts()
{
    local contacts_list="$1"
    local group_name="$2"
    local group_id
    declare -A _contacts_array

    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, unable to add contacts to nonexistent group'
        return 22 # EINVAL
    fi

    split_contact_infos "$contacts_list" _contacts_array

    if [[ "$?" -ne 0 ]]; then

```

cont →

```

    → cont
    return 22 # EINVAL
fi

add_contacts _contacts_array

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

add_contact_group _contacts_array "$group_id"

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

return 0
}

# This function add the association between the contacts
# and its group in the database
#
# @contacts_array: The contact name
# @group_id: The id of group which the contacts will be associated
#
# Returns:
# returns 0 if successful, non-zero otherwise
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)' "$values" '' 'VERBOSE')
        ret="$?"
    done
}

```

cont →


```

→ cont
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 função `show_email_groups` é utilizada para exibir informações sobre os grupos de e-mail cadastrados. Caso receba como parâmetro o nome de um grupo específico, a função valida sua existência e, em seguida, mostra os contatos associados a esse grupo. Por outro lado, quando nenhum parâmetro é informado, são exibidas informações gerais sobre todos os grupos existentes no banco de dados.

Programa 3.16 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"

```

cont →

```

    → cont
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
    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))

```

cont →

```

→ cont

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 "${!groups_info[@]"; do
    IFS='|' read -r id name created_at <<< "${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 ' ' '-'
}

```

```

joao-souza@joao-souza-upp:~$ kw manage-contacts --group-show
ID      |Name      |Contacts      |Created at
-----|-----|-----|-----
2       |Trabalho  |0             |2025-09-15
3       |Faculdade |2             |2025-09-15

```

Figura 3.1: Exemplo do comando “group_show” sem um grupo especificado.

```

joao-souza@joao-souza-upp:~$ kw manage-contacts --group-show=Faculdade
ID      |Name      |Email      |Associated Groups |Created at
-----|-----|-----|-----|-----
2       |joaousp   |joaosouzaa12@usp.br |1                |2025-09-15
3       |joaonormal|joaosouzaa12@gmail.com|1                |2025-09-16

```

Figura 3.2: Exemplo do comando “group_show” para um grupo específico.

3.2.4 Enviar patches para grupos

Dois novos comandos, `-to-groups` e `-cc-groups` foram adicionados à ferramenta `send-patch` para permitir o envio de patches diretamente para grupos. Esses comandos recebem listas com os nomes dos grupos separados por vírgulas, busca os contatos associados no banco de dados e adiciona seus email’s como destinatários.

Programa 3.17 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 3.18 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
}
```

cont →

```

→ cont
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 "$cc_recipients" || exit_msg 'Please review your `--cc`
    ↪ list.'
    cmd+=" --cc=\"$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 gráfica que poderia ser desejável, principalmente para visualizar informações dos grupos e contatos de uma maneira mais organizada.

3.3 KW Patch track

3.3.1 Objetivos

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 revisões se acumulam, tornando cada vez mais difícil manter o controle sobre o histórico, as respostas recebidas e o estado atual de cada revisão.

Diante dessa limitação, surgiu a necessidade de uma ferramenta capaz de registrar, rastrear e atualizar automaticamente o status dos patches submetidos. 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.

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.

3.3.3 Integração com o mutt

3.3.4 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.

Registro e Rastreamento das contribuições

Durante a submissão dos patches com a ferramenta *kw send_patch*, uma contribuição pode ser especificada através da flag *-to-contribution=<nome-da-contribuição>*, ou através do terminal interativo que identificará e exibirá as contribuições ainda ativas do usuário para nova submissão além de oferecer a possibilidade de criar uma nova. Nesse momento, ainda de maneira interativa, caso o usuário esteja realizando uma nova contribuição ele tem a possibilidade de identificar um repositório e um mantenedor responsável. Após a submissão, cada novo patch enviado é registrado no banco de dados, juntamente com informações sobre sua versão, título, autor, data de criação, *commit_hash* e também a sua submissão, caso o patch já tenha sido submetido anteriormente (Caso já exista outro patch no banco de dados com o mesmo título, autor, *commit_hash* e contribuição), apenas a nova submissão será registrada. Por fim, é registrada uma nova *submission* agrupando as submissões individuais de cada patch enviado na sessão do *kw send_patch* e as relacionando à contribuição.

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, 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 hash's dos commits, avalia-se também os arquivos de patches preliminares, gerados pelo *kw send_patch* para pré-processamento interno. Ainda que parte dos dados extraídos 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.

Atualização Automática de Status

Atualização Manual de Status

Visualização de contribuição

Renomeação de contribuição

Realocação de submissões

Definição de repositórios

Definição de mantenedores

Arquivamento de contribuições

3.3.5 Resultados

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 como perpetual development até o lançamento de suas versões finais, os “kernels estáveis”. 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 Kernel Workflow 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 esse 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.

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 queries, exigindo inserção de trechos SQL (por exemplo WHERE) diretamente nos comandos de seleção; para corrigir isso, os comandos de leitura foram parametrizados para aceitar WHERE parametrizado, ORDER BY por parâmetro e 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 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_maintainers`, 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.

Além disso, focando-se de maneira mais específica no problema das listas de email como método de contribuição para o kernel Linux, a terceira implementação desse trabalho, o `kw patch-track`, foca justamente em oferecer uma ferramenta de gerenciamento local das submissões do usuário. Para isso, a ferramenta se integra também com a ferramenta de envio, `kw send-patch`, registrando submissões do usuário e utilizando raspagem de dados dos arquivos gerados durante esse processo para extrair as informações necessárias e as registrar no banco de dados. Além disso, através do comando `kw patch-track pull`, a ferramenta também oferece um sistema de atualização automática do status dos patches, utilizando heurísticas internas para identificar em qual estágio um patch estaria. A integração com o `mutt`, permite também que, através do terminal, o usuário seja capaz de interagir e visualizar a lista de submissões de suas contribuições, bem como as respostas e revisões de contribuidores e mantenedores da comunidade, podendo até mesmo respondê-las. Dessa forma, suprimindo a dependência de ferramentas externas para o processo de acompanhamento das revisões, além de fornecer uma solução integrada ao `kw`, que permite a extração e análise de dados de forma concreta sobre essa etapa para estudos futuros.

A integração 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).
- [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](https://doi.org/10.21275/SR24127210817) (citado na pg. 12).
- [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 na pg. 7).
- [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=zPnlcyGllu7llcmK> (acesso em 03/09/2025) (citado na pg. 13).
- [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. 12).
- [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](https://doi.org/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/v4.14/process/2.Process.html> (acesso em 04/09/2025) (citado nas pgs. 7, 8).