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

LKML5Ws: Linux Mailing List Dataset

Eduardo Mendes Lopes

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: Rafael Passos
Cossupervisor: Paulo Meirelles

São Paulo
2025

Agradecimentos

No último dia de aula como graduando, acabei pegando o circular até em casa, como de costume, e, no ônibus, fui relembrando os anos que passei até chegar a esse dia. Lembrei-me de quando eu era apenas uma criança saindo do Maranhão e me mudando para o Paraná; lembrei dos meus anos difíceis no curso técnico em Eletrônica na UTFPR; dos meus dias e noites estudando para o vestibular; e, finalmente, de todos os meus dias na USP.

As coisas até aqui nunca foram fáceis. Batalhei e lutei muito, mas acredito que as coisas realmente valiosas na vida nunca são. Por isso, agradeço enormemente aos meus pais e à minha família por sempre me apoiarem e incentivarem nas minhas decisões; à minha avó e ao meu avô, que faleceram e não puderam ver o neto se formando, mas acredito que estariam muito felizes; aos meus amigos, por sempre estarem comigo em todos os momentos, virando noites no IME para estudar para provas e EPs.

Agradeço também aos meus professores, que são pessoas incríveis e que me ensinaram muito ao longo desses anos, especialmente ao professor Paulo Meirelles, que desde a Rede Linux me acompanha e incentiva. Agradeço ao meu orientador, Rafael Passos, por toda a paciência e pelos ensinamentos, e ao David e ao Arthur por também fazerem parte deste TCC.

Por fim, agradeço à USP e ao seu programa de apoio estudantil, que me possibilitou estar em São Paulo e custear meus estudos.

Resumo

Eduardo Mendes Lopes. **LKML5Ws: Linux Mailing List Dataset**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

O kernel Linux é um dos projetos de Software Livre mais complexos e influentes da atualidade, sendo desenvolvido de forma colaborativa há mais de três décadas por meio de um modelo baseado em revisões públicas realizadas em listas de discussão por e-mail. Embora sistemas de controle de versão registrem as alterações finais incorporadas à base de código, uma parcela significativa do esforço envolvido no processo de desenvolvimento, incluindo revisões, testes, debates e rejeições de contribuições, permanece documentada exclusivamente nessas listas de discussão. Diante desse cenário, este trabalho tem como objetivo principal a construção de um *dataset* abrangente que possibilite a investigação dos processos sociais e técnicos que antecedem a aceitação de contribuições no kernel Linux.

Na primeira parte deste trabalho, apresenta-se uma contextualização do desenvolvimento do kernel Linux, bem como os principais softwares que dão suporte a esse processo. Na segunda parte, descreve-se a metodologia empregada para a coleta, extração e estruturação dos dados, assim como os fundamentos conceituais que orientam a obtenção de e-mails a partir do Kernel Lore Archive. Como resultado, é apresentado o LKML5Ws, um conjunto de dados com mais de 20 milhões de e-mails provenientes de 345 listas de discussão, totalizando mais de 200 GB de dados brutos, compactados em mais de 55 GB de arquivos no formato Parquet. Por fim, é apresentada uma análise exploratória com o LKML5Ws que demonstra seu potencial para revelar diferenças na dinâmica de revisão e teste entre distintos subsistemas do Kernel, evidenciando tendências divergentes na participação da comunidade ao longo do tempo, bem como propostas de usos futuros para o dataset.

Esse conjunto de dados oferece uma visão ampla do desenvolvimento do kernel ao explicitar: no que consiste cada contribuição (*what*), quando ela foi proposta (*when*), quem participou (*who*), para qual lista foi submetida (*where*) e por que se tornou, ou não, parte do código (*why*). Além disso, busca contribuir para ampliar a base empírica disponível à comunidade de Engenharia de Software, oferecendo uma nova perspectiva sobre os aspectos sociais e técnicos que moldam a evolução de um dos mais emblemáticos projetos de Software Livre.

Palavras-chave: Kernel Linux. Mailing Lists. Lore Archive. Software Livre.

Abstract

Eduardo Mendes Lopes. **LKML5Ws: Linux Mailing List Dataset**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

The Linux kernel is one of the most complex and influential Free Software projects today, having been developed collaboratively for more than three decades through a model based on public reviews conducted on email mailing lists. Although version control systems record the final changes incorporated into the codebase, a significant portion of the effort involved in the development process, including reviews, testing, discussions, and the rejection of contributions, remains documented exclusively in these mailing lists. In this context, the main objective of this work is to build a comprehensive dataset that enables the investigation of the social and technical processes that precede the acceptance of contributions into the Linux kernel.

In the first part of this work, we provide background on the development of the Linux kernel, as well as an overview of the main software tools that support this process. In the second part, we describe the methodology used for data collection, extraction, and structuring, along with the conceptual foundations that guide the retrieval of emails from the Kernel Lore Archive. As a result, we present LKML5Ws, a dataset containing more than 20 million emails from 345 mailing lists, totaling over 200 GB of raw data, compressed into more than 55 GB of files in Parquet format. Finally, we present an exploratory analysis using LKML5Ws that demonstrates its potential to reveal differences in review and testing dynamics across distinct kernel subsystems, highlighting divergent trends in community participation over time, as well as proposing future uses for the dataset.

This dataset provides a broad view of kernel development by making explicit what each contribution consists of (what), when it was proposed (when), who participated (who), which mailing list it was submitted to (where), and why it did or did not become part of the codebase (why). In addition, it aims to expand the empirical foundation available to the Software Engineering community by offering a new perspective on the social and technical aspects that shape the evolution of one of the most emblematic Free Software projects.

Keywords: Kernel Linux. Mailing Lists. Lore Archive. Free/Libre Software.

Lista de abreviaturas

SO	Sistema Operacional
FLOSS	*Free/Libre and Open Source Software*
LKML	*Linux Kernel Mailing List*
CC/Cc	*Carbon Copy*
rc	*Release Candidate*
LTS	*Long Term Support*
MTA	*Mail Transfer Agent*
DNS	*Domain Name System*
MDA	*Mail Delivery Agent*
POP3	*Post Office Protocol version 3*
IMAP	*Internet Message Access Protocol*
NNTP	*Network News Transfer Protocol*
Blob	*Binary Large Object*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
HTML	*Hypertext Markup Language*
SHA-1	*Secure Hash Algorithm 1*
URL	*Uniform Resource Locator*
UTF-8	*Unicode Transformation Format - 8-bit*
CSV	*Comma-separated values*

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Introdução

Os computadores constituem ferramentas essenciais para o funcionamento do mundo moderno. Os sistemas computacionais variam desde computadores pessoais, utilizados pelo público em geral para atividades cotidianas, até sistemas embarcados presentes em uma parcela significativa de automóveis, aeronaves e indústrias, nos quais desempenham tarefas críticas. Embora não seja um requisito estrito, os computadores são, em geral, compostos por duas partes interdependentes: hardware e software. O hardware corresponde aos componentes eletrônicos físicos do sistema, enquanto o software compreende o conjunto de programas e dados responsáveis por controlar e coordenar o funcionamento desse hardware.

O software pode assumir diferentes formas para instruir o hardware. Programas escritos diretamente em linguagem de máquina, que utilizam sequências de zeros e uns para codificar instruções específicas de um determinado processador, representam o nível mais baixo de abstração. Por outro lado, programas podem ser desenvolvidos sobre camadas adicionais de software, criando abstrações que reduzem a dependência das especificações do hardware e tornam o desenvolvimento mais legível, estruturado e produtivo. Nesse contexto, programas que interagem diretamente com o hardware são classificados como de baixo nível, enquanto aqueles que se apoiam em abstrações fornecidas por outros programas são considerados de alto nível.

Do ponto de vista dos programas de alto nível, que dependem de software de baixo nível para acessar recursos computacionais, o sistema operacional (SO) é um software que atua como intermediário entre esses dois níveis. Além disso, o SO é o componente responsável por gerenciar os recursos do computador, como processador, memória, dispositivos de entrada e saída, que são tipicamente limitados e precisam ser coordenados para evitar conflitos quando múltiplos programas os utilizam simultaneamente. Dessa forma, os sistemas operacionais são fundamentais para garantir eficiência, segurança e confiabilidade no uso dos recursos computacionais.

O principal componente de um sistema operacional é o seu *kernel*, responsável por encapsular suas funcionalidades centrais e por abstrair os detalhes de hardware por meio de componentes específicos, conhecidos como *drivers*. Um importante exemplo de *kernel* de sistema operacional é o Kernel Linux. O kernel Linux foi lançado oficialmente por Linus Torvalds em 5 de outubro de 1991, na versão 0.0.2, inspirado nos sistemas operacionais UNIX e MINIX. Como um *kernel* isolado e unicamente não constitui um sistema operacional completo, o Linux foi combinado com os utilitários desenvolvidos pelo Projeto GNU,[1]

[1] O Projeto GNU foi anunciado por Richard Stallman em 27 de setembro de 1983, para fornecer uma coleção

resultando no sistema operacional GNU/Linux, classificado como Software Livre e de Código Aberto (Free/Libre and Open Source Software – FLOSS).[2] Esse modelo garante aos usuários a liberdade de obter, executar, estudar, modificar e redistribuir o software.

A criação do GNU/Linux representou uma grande ruptura no desenvolvimento de sistemas operacionais. Embora o projeto GNU já disponibilizasse praticamente todos os componentes necessários para a construção de um SO FLOSS, as tentativas anteriores de desenvolver um *kernel* próprio não haviam obtido sucesso. Assim, com o surgimento do Kernel Linux se preencheu essa lacuna e possibilitou a consolidação de um ecossistema completo, amplamente adotado nas décadas seguintes.

Atualmente, existem inúmeras variações do GNU/Linux, adaptadas para operar de forma otimizada em diferentes contextos e plataformas. O ecossistema GNU/Linux desempenha um papel central na infraestrutura computacional global, sendo amplamente utilizado em servidores, dispositivos de rede que compõem o núcleo da Internet, sistemas embarcados e ambientes educacionais. O acesso irrestrito ao código-fonte torna o GNU/Linux uma referência prática e amplamente utilizada no ensino e na pesquisa em sistemas operacionais.

No que se refere especificamente ao kernel Linux, o projeto vem sendo desenvolvido de forma colaborativa há mais de 30 anos. Com o passar do tempo, o projeto torna-se progressivamente maior e mais complexo. Além disso, o volume de contribuições ao projeto também cresce rapidamente, de modo que uma única pessoa ou grupo não consegue compreender e manter toda a base de código. Como solução, o modelo de contribuição do kernel Linux emprega um modelo de cadeia de comando de forma a dividir a responsabilidade de manutenção do projeto em porções menores, chamadas de subsistemas. Cada subsistema possui um ou mais mantenedores responsáveis por decidir quais mudanças (*patches*) são aceitas no subsistema correspondente. Assim, essas mudanças vão subindo na hierarquia de processos até que Linus Torvalds as incorpore em um lançamento oficial do Linux.

Esse modelo de desenvolvimento distribuído é fortemente sustentado pelo sistema de controle de versão Git. O Git foi criado por Linus Torvalds em 2005 como resposta direta às limitações das ferramentas existentes à época para lidar com o volume, a descentralização e a velocidade do desenvolvimento do kernel Linux. Desde então, o Git tornou-se um componente essencial do fluxo de trabalho do projeto, permitindo o versionamento eficiente do código, a manutenção de múltiplas árvores de desenvolvimento e a integração segura de contribuições oriundas de milhares de desenvolvedores distribuídos globalmente.

Apesar da existência do Git para o versionamento do código-fonte e plataformas centralizadoras como GitHub e GitLab, o e-mail é o principal meio utilizado para propagar mudanças entre contribuidores, mantenedores e demais participantes do processo de revisão. As listas de discussão desempenham um papel fundamental nesse ecossistema, atuando como o registro público e permanente das interações técnicas e sociais do desenvolvimento do kernel Linux.

---

completa de softwares livres para a sociedade, incluindo um sistema operacional completo.

[2] Neste trabalho, a sigla "FLOSS" é usada para representar "Free Software", "Open Source Software"(OSS), e "Free/Open Source Software"(FOSS)

Uma forma de acesso a esse histórico de comunicação é viabilizado por meio dos arquivos do *Lore*, que agregam e mantêm atualizados os registros de todas as listas de discussão relacionadas ao desenvolvimento do kernel Linux. Esses arquivos abrangem desde listas de alto tráfego e longa duração, como a *Linux Kernel Mailing List* (LKML), até listas de menor atividade ou já desativadas, como a `linux-hotplug@vger.kernel.org`. No total, esses arquivos públicos compreendem dezenas de milhões de mensagens.

Nesse sentido, apesar de haver *commits* que descrevem e explicam explicitamente as mudanças de cada *patch* nos históricos do Git, uma parcela significativa dos comentários que viabilizaram a aceitação de contribuições, críticas que resultaram na rejeição de *patches* e discussões conceituais que influenciaram decisões arquiteturais raramente são capturados no histórico de *commits* do Git. Dessa forma, as listas de discussão constituem uma fonte inestimável para compreender não apenas o que foi modificado no kernel, mas por que e como essas decisões foram tomadas.

Neste contexto, este trabalho apresenta o dataset **LKML5Ws**, uma coleção abrangente de contribuições extraídas de e-mails enviados às listas de discussão do kernel Linux. Para cada contribuição, o conjunto de dados disponibiliza: (*i*) a modificação de código associada (**what**); (*ii*) a data de submissão (**when**); (*iii*) os papéis dos colaboradores auxiliares identificados nos trailers (**who**); (*iv*) a lista de discussão de destino (**where**); e (*v*) a justificativa apresentada pelo autor no corpo do e-mail (**why**).

Por fim, este Trabalho de Conclusão de Curso pode ser estruturado logicamente em três seções principais. A primeira seção, composta por sete capítulos, apresenta a contextualização do processo de contribuição para o kernel Linux, abordando seus conceitos fundamentais, fluxos de trabalho e softwares utilizados pela comunidade, bem como a relevância desses softwares para a extração dos dados. A segunda seção descreve detalhadamente a metodologia de coleta, interpretação, estruturação e armazenamento dos dados, tudo em um formato de arquivo colunar. Por último, a terceira seção, contendo três capítulos, apresenta os resultados obtidos, discute possíveis análises viabilizadas pelo conjunto de dados, relaciona o trabalho com pesquisas existentes na literatura e aponta limitações e direções para trabalhos futuros.

Capítulo 1

O Processo de Desenvolvimento e Contribuição para o Kernel Linux

O modelo de desenvolvimento do kernel Linux difere bastante dos fluxos de trabalho existentes em projetos modernos que utilizam plataformas centralizadas de colaboração, como GitHub ou GitLab. Em essência, o kernel Linux caracteriza-se como um projeto distribuído e descentralizado, estruturado a partir de uma hierarquia bem definida de mantenedores e de um processo de revisão de código mediado pela troca de mensagens por e-mail.

O fluxo de contribuição tem início no ambiente de desenvolvimento do próprio programador. Para que um *patch* seja considerado para inclusão, é necessário seguir um conjunto de diretrizes de padronização que abrangem tanto boas práticas de codificação(Linux Documentation, 2025d) quanto o princípio da atomicidade. Esse princípio estabelece que cada *commit* deve representar uma única modificação lógica e atômica. Por exemplo, a correção de um defeito em um *driver* de *Wi-Fi* e a alteração de uma variável no subsistema *Bluetooth* devem ser submetidas como dois *patches* independentes. Além do mais, ao final da mensagem de *commit*, o desenvolvedor deve obrigatoriamente incluir a linha `Signed-off-by`, contendo seu nome e endereço de e-mail, certificando a autoria e o direito de licenciamento do código, conforme definido pelo *Developer Certificate of Origin* (DCO)(Linux Documentation, 2020).

Em contraste com o modelo tradicional de *pull requests*, a submissão de *patches* no kernel Linux ocorre via e-mail. Para isso, o desenvolvedor utiliza *scripts* auxiliares disponibilizados na própria árvore de código-fonte, como o `get_maintainer.pl`,[1] para identificar a lista de discussão apropriada e os mantenedores responsáveis pelo subsistema que está sendo modificado. Por exemplo, ao propor uma modificação em um *driver* USB, o *script* pode indicar o envio do *patch* para a lista `linux-usb@vger.kernel.org`, com cópia (CC) para o mantenedor Greg Kroah-Hartman. O envio deve ser realizado estritamente em formato texto plano, contendo a mensagem de *commit* e o *diff* correspondente das alterações propostas. Quando múltiplos *patches* são enviados simultaneamente, formando um *patchset*,

[1] código: https://archive.softwareheritage.org/browse/content/sha1_git:4414194bedcfd747bd24199b5de9ccf04bf6d227/?origin_url=https://github.com/torvalds/linux&path=scripts/get_maintainer.pl

torna-se necessária a inclusão de uma mensagem introdutória denominada *cover letter*, cuja finalidade é contextualizar o objetivo geral das mudanças.

Após a recepção do e-mail pela lista de discussão, inicia-se o processo de revisão propriamente dito. Inicialmente, ocorre a revisão por pares, na qual desenvolvedores da comunidade examinam o código, sugerem melhorias e apontam eventuais falhas. Por fim, os mantenedores do subsistema realizam a avaliação final. Durante esse processo, são adicionadas ao *commit* diversas tags que sinalizam o estado da revisão, tais como `Reviewed-by`, `Acked-by` e `Tested-by`, as quais serão detalhadas em seção posterior. É importante destacar que raramente um *patch* é aceito em sua primeira versão (v1); na maioria dos casos, o autor precisa submeter múltiplas iterações (v2, v3, etc.), incorporando as sugestões e correções apontadas pela comunidade.

Uma vez aprovado, o *patch* inicia sua progressão na hierarquia de desenvolvimento. O mantenedor do subsistema aplica a alteração na *branch* do subsistema e, periodicamente, o conjunto de mudanças acumuladas nessas *branches* é integrado em uma *branch* intermediária denominada `linux-next`(Linux Documentation, 2025a). O propósito dessa etapa é identificar e resolver conflitos de integração entre diferentes subsistemas antes que as alterações alcancem a *branch* principal.

O ciclo de desenvolvimento de uma nova versão do *kernel* é iniciado imediatamente após o lançamento de uma versão estável anterior, com a abertura da chamada *merge window*. Durante este período, que dura aproximadamente duas semanas, acontece o fluxo de aceitação de código por Linus Torvalds, ele recebe e processa milhares de solicitações de *pull requests* dos mantenedores de subsistemas. O critério para essa fase é que o código submetido deve ser considerado "estável"e já ter passado por testes prévios em árvores de integração (como a `linux-next`). Segundo a documentação oficial(Linux Documentation, 2025b), mudanças que não estiverem maduras ou que não foram integradas a tempo para a janela de *merge* devem aguardar o próximo ciclo de desenvolvimento.

Ao final dessas duas semanas, Linus Torvalds declara o fechamento da *merge window* e publica a primeira versão candidata, denominada -rc1 (*Release Candidate 1*). A partir deste momento, o foco do desenvolvimento muda: se deixa de aceitar novas funcionalidades para priorizar a estabilidade do sistema. O objetivo principal dessa fase passa a ser a identificação e correção de regressões.[2]

O processo segue com lançamentos de novas versões candidatas (-rc2, -rc3, etc.). O ciclo de estabilização dura, em média, entre sete e dez semanas. A decisão de publicar a versão final e estável cabe a Linus Torvalds, quando o volume de correções críticas diminui a um nível que indique maturidade do código.

Uma vez publicada a versão estável na árvore *Mainline*, a responsabilidade por sua manutenção é mudada. Enquanto Linus inicia o próximo ciclo de desenvolvimento, uma equipe dedicada à manutenção de versões estáveis (*Stable Team*) assume o suporte da versão recém-lançada. O processo de manutenção estável consiste em correções de segurança e de erros críticos que foram descobertos na árvore principal. Essas versões são numeradas com um terceiro dígito (ex: 6.1.1, 6.1.2) e são as bases utilizadas por distribuições como

[2] defeitos introduzidos por mudanças recentes que fazem com que o sistema pare de funcionar como funcionava anteriormente.

Fedora, Ubuntu e Debian para fornecer um sistema robusto e seguro aos seus usuários finais. Algumas dessas versões são selecionadas para suporte de longo prazo (*LTS - Long Term Support*), garantindo atualizações de segurança por vários anos.

Capítulo 2

Infraestrutura de Comunicação e Arquitetura de E-mail no Linux

Para compreender o ecossistema de desenvolvimento do kernel Linux, é fundamental analisar não apenas o processo de escrita e submissão de código, mas também o funcionamento das listas de discussão (*mailing lists*) e da infraestrutura de e-mails em sistemas Unix/Linux. Nesse contexto, uma *mailing list* atua, essencialmente, como um mecanismo de redistribuição de mensagens, funcionando como um "refletor" ou multiplexador de comunicações entre seus participantes.

No contexto do servidor `vger.kernel.org`, responsável por hospedar as principais listas associadas ao projeto do Kernel Linux, o gerenciamento dessas listas é realizado, predominantemente o sistema mlmmj.[1] Substituindo o antigo software Majordomo.[2] Para realizar uma inscrição hoje, o usuário deve enviar um e-mail, preferencialmente vazio, para o endereço da lista desejada seguido da extensão `+subscribe`(Linux Documentation, 2025f), como no exemplo `linux-kernel+subscribe@vger.kernel.org`.

Ao receber essa solicitação, o servidor ignora o conteúdo do e-mail e inicia um processo de confirmação automática, enviando uma mensagem de retorno para validar a identidade do remetente e evitar cadastros maliciosos ou acidentais. A inscrição só é efetivada após o usuário responder a esse e-mail de confirmação, momento em que seu endereço é registrado na base de dados e ele passa a receber as mensagens enviadas à lista. Esse modelo de interação baseada em endereços também simplifica o cancelamento da participação, que segue a mesma lógica ao utilizar o sufixo `+unsubscribe`, garantindo um gerenciamento mais seguro e eficiente para os colaboradores do projeto.

O tráfego dessas mensagens depende da interação coordenada entre diferentes agentes e protocolos, cada qual desempenhando funções específicas na arquitetura de e-mails em sistemas Unix/Linux.

[1] https://codeberg.org/mlmmj/mlmmj

[2] https://subspace.kernel.org/vger.kernel.org.html\#what-happened-to-majordomo

2.1 Agentes de Transporte e o Protocolo SMTP

O envio e o roteamento de mensagens através da Internet são executados pelo protocolo SMTP (*Simple Mail Transfer Protocol*).[3] O software responsável por implementar esse protocolo e encaminhar as mensagens até seus destinos é denominado MTA (*Mail Transfer Agent*) (Dent, 2003). No ambiente Linux, exemplos conhecidos de MTAs incluem Postfix, Exim e Sendmail. O MTA atua como um servidor de e-mail: ele recebe a mensagem do usuário ou de outro servidor, resolve o endereço de destino por meio do DNS e realiza o transporte da mensagem através da rede até o servidor destinatário.

2.2 Agentes de Entrega e Armazenamento

Uma vez que a mensagem alcança o servidor de destino, o MTA delega a etapa final do processo ao MDA (*Mail Delivery Agent*),[4] como os softwares Dovecot ou Procmail. A função do MDA consiste em armazenar a mensagem em um diretório local do usuário. No ecossistema Linux, o formato de armazenamento mais difundido é o *Maildir*. Diferente de formatos mais antigos, como o *mbox*, que concatenam as mensagens que chegam em um único arquivo, o *Maildir* armazena cada e-mail em um arquivo de texto individual. Essa abordagem garante atomicidade nas operações de escrita e reduz o risco de corrupção de dados em cenários concorrentes. A estrutura básica do *Maildir*(Unix Documentation, 2025) é organizada nos seguintes diretórios:

- `/Maildir/tmp/`: armazena mensagens em processo de escrita ou entrega pelo MDA;
- `/Maildir/new/`: contém mensagens entregues com sucesso, mas ainda não acessadas pelo usuário;
- `/Maildir/cur/`: reúne mensagens que já foram visualizadas ou processadas por um cliente de e-mail.

2.3 Protocolos de Acesso e Leitura (IMAP versus NNTP)

Para que o usuário final possa acessar as mensagens armazenadas, são disponibilizados protocolos de leitura, dentre os quais mais conhecidos são o POP3[5] e o IMAP.[6] Este último é o mais utilizado em *in-boxes* pessoais, pois permite a sincronização do estado das mensagens entre servidor e cliente.

Já o protocolo NNTP (*Network News Transfer Protocol*)[7] é particularmente eficiente na

[3] https://datatracker.ietf.org/doc/html/rfc5321

[4] https://dl.acm.org/doi/pdf/10.17487/RFC5068

[5] https://www.ietf.org/rfc/rfc1939.txt

[6] https://datatracker.ietf.org/doc/html/rfc3501

[7] https://datatracker.ietf.org/doc/html/rfc3977

distribuição de grandes volumes de mensagens organizadas em estruturas hierárquicas de discussão (*threads*). Ele opera como um fluxo de notícias, possibilitando que os clientes obtenham rapidamente apenas os cabeçalhos de milhares de mensagens. Dessa forma, a estrutura completa das discussões pode ser reconstruída localmente, enquanto o conteúdo integral das mensagens é transferido apenas sob demanda.

Capítulo 3

Introdução sobre o kernel lore archive

O `lore.kernel.org` é a principal plataforma de arquivamento e interface web das listas de discussão oficiais do Kernel Linux e de seus subprojetos relacionados. Ele atua como o ponto centralizado de acesso aos arquivos de e-mails relacionados ao desenvolvimento do Kernel, desempenhando papel fundamental na organização das comunicações técnicas do projeto.

Sua principal função é assegurar que toda a comunicação do ecossistema, incluindo discussões técnicas, submissões de patches e processos de revisão de código, seja armazenada de forma permanente, clonável e facilmente pesquisável. Dessa maneira, o Kernel Lore Archive contribui diretamente para a transparência e rastreabilidade do processo de desenvolvimento do Kernel Linux.

Do ponto de vista técnico, o `lore.kernel.org` baseia-se no software public-inbox, cuja arquitetura e funcionamento serão abordados de forma mais aprofundada na seção subsequente. Essa infraestrutura é mantida e operada pela própria comunidade do Kernel Linux, de acordo com os princípios de desenvolvimento aberto que direcionam o projeto.

O objetivo central do `lore.kernel.org` é oferecer à comunidade múltiplas formas de acesso ao histórico das discussões técnicas. Esse acesso pode ocorrer por meio de uma interface web, pela clonagem direta dos repositórios que armazenam os e-mails ou ainda através do protocolo NNTP.

Sendo assim, o `lore.kernel.org` desempenha um papel crucial no ecossistema do Kernel Linux, ao manter um registro histórico completo das decisões técnicas, debates e da evolução do código-fonte ao longo do tempo. Trata-se da principal ferramenta utilizada pelos desenvolvedores para localizar *patches* recentes, revisar o histórico de modificações de componentes específicos e sincronizar espelhos (*mirrors*[1]) dos arquivos de listas de discussão.

[1] um mirror (espelho) é uma cópia exata ou fiel de um site, servidor ou conjunto de dados, hospedada em um servidor diferente do original. Essa técnica é usada para replicar dados em tempo real ou periodicamente, garantindo que a informação esteja disponível em múltiplos locais geográficos ou máquinas.

Capítulo 4

Arquitetura do Public-Inbox

O public-inbox[1] é o software por baixo do `lore.kernel.org`. Ele é um software de código aberto com o propósito é fornecer uma solução descentralizada para arquivamento de listas de discussão (*mailing lists*) para a comunidade de software livre e comunidades técnicas, ele utiliza o *git* como mecanismo principal para realizar esse armazenamento. Ele foi projetado para complementar ou substituir os sistemas tradicionais de listas de discussão, garantindo uma comunicação permanentemente arquivada, facilmente acessível e de fácil replicação.

O public-inbox é uma caixa de ferramentas que se utiliza de várias tecnologias para servir determinados propósitos.

4.1 Armazenamento e versionamento baseado em Git

O uso do git no public-inbox(Public-Inbox Documentation, 2025c) é uma forma de deixar o armazenamento e o versionamento fácil e imutável, pois ele transforma a lista de e-mail em repositório git aproveitando todas as ferramentas git padrão para guardar esse histórico. Cada e-mail é convertido em um objeto Blob (*Binary Large Object*),[2] que é a estrutura usada pelo git para armazenamento de dados de arquivos. Para que esse *blob* seja parte do histórico, ele precisa ser referenciado por um objeto *Tree* e, subsequentemente, por um objeto *commit*. Assim, o objeto *blob* armazena o conteúdo do e-mail, que se transforma em um objeto *commit*, assim esse *blob* é referenciado dentro do *Tree* do *commit* com um nome específico, que é determinado pelo `message-id` do e-mail.

O public-inbox, com versão v2(Public-Inbox Documentation, 2025d), também tem a funcionalidade de dividir os arquivos em múltiplos repositórios chamados de “épocas” (*epochs*). Em resumo, as épocas são múltiplos repositórios Git, divididos por tamanho, que juntos formam o arquivo completo da caixa de entrada. Como os repositórios das listas de discussão do kernel Linux podem crescer para centenas de gigabytes. O git tem um

[1] https://public-inbox.org/public-inbox-overview.html

[2] é um objeto usado para armazenar o conteúdo binário de um arquivo

dificuldade de lidar com esses repositórios muito extensos. Assim, o public-inbox faz a divisão do repositório em uma série de épocas, como `git/0.git`, `git/1.git`, `git/2.git`, e assim por diante, cuja segmentação é divida em aproximadamente um gigabytes. Essa funcionalidade permite limitar o crescimento do histórico em qualquer repositório único, tornando a clonagem e a manutenção mais eficientes.

Ao ser um "repositório Git",[3] todo o arquivo pode ser facilmente clonado, espelhado e transferido para novos servidores sem perda ou divisão do histórico, garantindo a descentralização.

4.2 Ingestão de mensagens

O software também oferece formas de capturar e injetar esses e-mails no arquivo Git. Diferente de um gerenciador de listas tradicional, que foca na distribuição para assinantes da lista, o public-inbox foca na preservação e acessibilidade das mensagens. Isso pode ser feito de forma passiva, através do `public-inbox-watch`,[4] que monitora diretórios *Maildir* ou servidores externos para espelhar mensagens, ou de forma ativa, atuando como um MDA(Public-Inbox Documentation, 2025b). Neste último caso, ele é integrado a um MTA (como o Postfix) para receber mensagens e alimentá-las instantaneamente no repositório Git, garantindo que esteja sempre sincronizado com o tráfego da lista.

4.3 Modelo de acesso de e-mail

O public-inbox oferece três formas de acesso aos e-mails armazenados nele, a primeira forma é por clonagem do repositório git, como já mencionado antes, a segunda é através de uma interface web subindo um servidor HTTP, permitindo consultas, visualização e navegação toda baseada em HTML (o que foi muito importante, pois para conseguir os e-mails em formato texto simples, esse trabalho utilizou essa funcionalidade, como irá ser especificado mais a frente) e a terceira forma é NNTP permite que os usuários leiam o arquivo através do *Network News Transfer Protocol*, o que significa que o arquivo pode ser acessado como um grupo de notícias.

4.4 Indexação e pesquisa

Para tornar os arquivos de listas de discussão facilmente acessíveis e pesquisáveis, o public-inbox utiliza um mecanismo de *full-text search*,[5] capaz de transformar um repositório Git essencialmente estático, eficiente para fins de armazenamento e distribuição, porém inadequado para consultas em larga escala, em um sistema de pesquisa rápido e otimizado. Essa funcionalidade é particularmente relevante para a visualização das

[3] diferentemente de repositório comum, quando se clona um repositório de e-mails, os dados dos e-mails estão contidos apenas nos arquivos `.git`.

[4] https://public-inbox.org/public-inbox-watch.html

[5] é um mecanismo que permite pesquisar palavras, termos e frases em grandes volumes de texto de forma rápida e inteligente.

mensagens por meio da interface HTTP, na qual a capacidade de busca eficiente é um requisito fundamental.

Embora o Git oferece, nativamente, mecanismos de pesquisa, como o comando `git grep`, tal abordagem é inviável quando aplicada em um repositório contendo milhões de arquivos, como ocorre nos arquivos das listas de discussão do Kernel Linux. Para contornar essa limitação, o public-inbox integra-se ao Xapian,[6] uma biblioteca de recuperação de informação baseada em modelos probabilísticos, projetada para lidar de forma eficiente com grandes volumes de dados textuais.

Durante o processo de indexação(Public-Inbox Documentation, 2025a), o public-inbox percorre os objetos do tipo *blob* armazenados no repositório Git e realiza o processamento necessário para popular o banco de dados do Xapian. Importante ressaltar que o Xapian não armazena o conteúdo integral das mensagens, uma vez que esse texto já se encontra preservado no repositório Git. Em vez disso, o sistema constrói um índice invertido, estrutura de dados que associa termos relevantes aos documentos nos quais eles ocorrem.

Nesse processo, o public-inbox decompõe cada e-mail em seus componentes fundamentais, incluindo cabeçalhos, como `From`, `To` e `Subject`, e o corpo da mensagem. Essas informações são então encaminhadas ao Xapian, que realiza a tokenização dos termos e aplica técnicas de normalização linguística, como a redução de palavras ao seu radical (*stemming*)(Xapian Documentation, 2025). Como resultado, variações morfológicas de um mesmo termo, como “patching”, “patched” e “patches”, são indexadas sob um único radical, por exemplo “patch”, permitindo que uma única consulta recupere todas as ocorrências semanticamente relacionadas.

Além disso, o Xapian possibilita a atribuição de prefixos contextuais aos termos indexados,[7] permitindo diferenciar a ocorrência de uma mesma palavra conforme sua posição na mensagem. Dessa forma, o public-inbox pode mapear termos encontrados no assunto, nos cabeçalhos ou no corpo do e-mail para prefixos distintos, assegurando que, por exemplo, a palavra “Linux” presente no campo `Subject` seja tratada de maneira diferente daquela localizada no corpo da mensagem. A tabela 4.1 explicita os prefixos utilizados e os respectivos componentes do e-mail aos quais estão associados.

Campo do E-mail	Prefixo Xapian (Convenção)	Exemplo de Termo Indexado
From (Autor)	`A` (Author)	Alinus (para "Linus")
Subject (Assunto)	`S` (Subject)	Skernel
Thread ID	`G` (Group/Thread)	G<msg-id>
Message-ID	`Q` (Unique ID)	Q<2023...@example.com>
Date	(Armazenado como Value)	Unix Timestamp
Body (Corpo)	(Sem prefixo)	kernel, bug, fix

Tabela 4.1: *Prefixos usados pelo Xapian para otimizar pesquisas.*

[6] https://xapian.org

[7] https://public-inbox.org/public-inbox-searchquery.html

Para cada mensagem indexada, o Xapian cria uma entrada denominada documento,[8] que armazena, entre outras informações, o identificador do *blob* correspondente no repositório Git, usualmente representado por seu hash SHA-1. Assim, quando uma consulta é realizada, o Xapian retorna os identificadores dos documentos que correspondem aos termos pesquisados. Em seguida, o public-inbox utiliza o `blob ID` associado para recuperar o conteúdo original diretamente do repositório Git e apresentá-lo ao usuário.

4.5 Estabilidade e portabilidade de links

Uma característica adicional do public-inbox é a garantia de persistência e estabilidade dos links para mensagens arquivadas. Os *permalinks*[9] são estruturados com base no cabeçalho `Message-ID` (definido pela RFC 5322[10]), que funciona como um identificador globalmente único para cada e-mail (Figura 4.1). Ao adotar o `Message-ID` na composição das URLs, o sistema assegura que as referências permaneçam válidas mesmo diante de migrações de infraestrutura ou mudanças de domínio. Como consequência, preserva-se a integridade das referências históricas, evitando a quebra de citações e facilitando a auditabilidade do processo de desenvolvimento ao longo do tempo, independentemente da evolução tecnológica dos servidores de hospedagem.

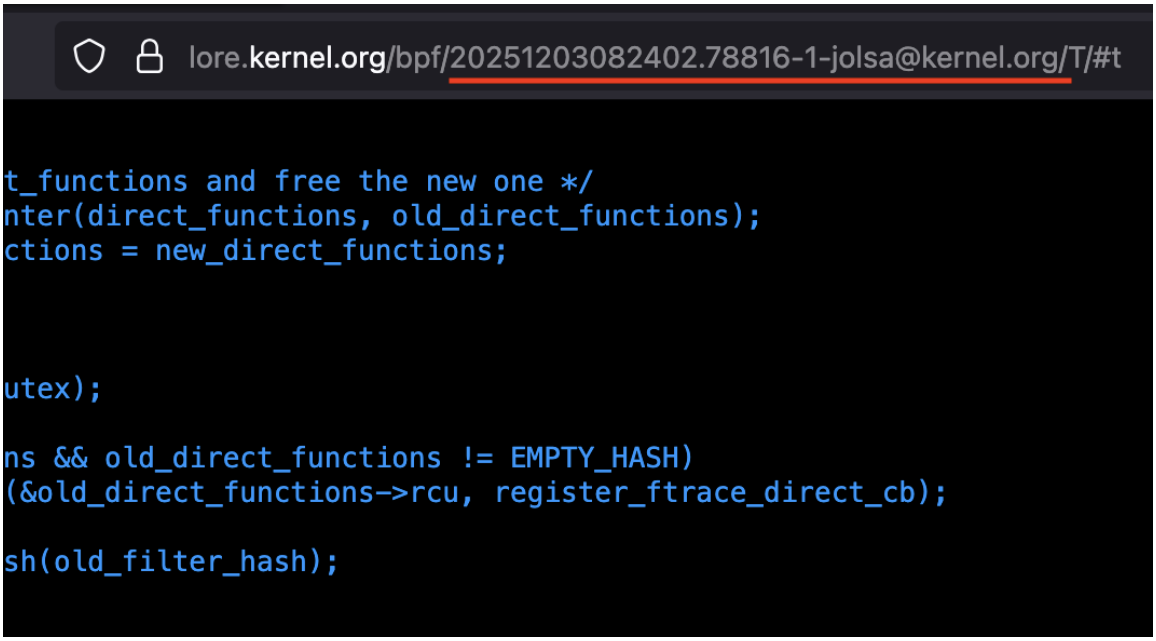

Figura 4.1: *Representação das URLs padronizadas no public-inbox.*

[8] https://xapian.org/docs/glossary.html

[9] URLs fixos e descritivos que direcionam para um conteúdo específico em um site (como um post, página ou artigo), projetados para não mudar.

[10] https://www.tech-invite.com/y50/tinv-ietf-rfc-5322.html

Capítulo 5

LEI (Local Email Interface)

O LEI (Local Email Interface)[1] é uma ferramenta desenvolvida no ecossistema do public-inbox e do `lore.kernel.org` com o objetivo de mitigar o problema da sobrecarga de e-mails enfrentados por desenvolvedores de software livre. Em projetos de grande escala, como o Kernel Linux, o volume diário de mensagens pode tornar inviável a assinatura integral de listas de discussão. Nesse contexto, o LEI atua como uma interface que permite aos desenvolvedores criar assinaturas virtuais baseadas em regras de busca, recebendo apenas o subconjunto de mensagens relevantes para o seu trabalho, sem a necessidade de se inscrever em listas completas.

Do ponto de vista de software, o LEI é um utilitário de linha de comando que converte consultas avançadas realizadas sobre os arquivos do public-inbox em *feeds* de e-mail personalizados, os quais são entregues diretamente na caixa de entrada do usuário, seja em formato *Maildir* local ou por meio de servidores IMAP remotos(Ryabitsev, 2021). Dessa forma, o fluxo de trabalho baseado em e-mail é preservado, ao mesmo tempo em que se reduz drasticamente o ruído de informação.

Entre suas funcionalidades, o LEI é capaz de realizar o download direto de mensagens armazenadas no `lore.kernel.org` ou em quaisquer outros servidores configurados na consulta. Para isso, ele utiliza a sintaxe de busca fornecida pelo Xapian (Figura 5.1), permitindo a formulação de consultas complexas que combinam múltiplos critérios, tais como caminhos de arquivos, nomes de funções, campos específicos do cabeçalho e termos presentes no corpo das mensagens.

```
(dfn:drivers/block/floppy.c OR dfhh:floppy_* OR s:floppy
 OR ((nq:bug OR nq:regression) AND nq:floppy))
AND rt:1.month.ago..
```

Figura 5.1: *Exemplo de query aceita pelo LEI.*

Neste exemplo[2] de utilização acima, o LEI pode formular uma consulta que selecione

[1] https://public-inbox.org/lei.html

[2] referência: https://people.kernel.org/monsieuricon/lore-lei-part-1-getting-started

mensagens que:

- referenciem arquivos específicos, por meio do prefixo "dfn";
- mencionem determinadas funções, utilizando o prefixo "dfhh";
- contenham o termo “floppy” no campo Subject, indicado pelo prefixo "s";
- mencionem termos como “bug” ou “regression” em conjunto com “floppy” no corpo da mensagem;
- estejam restritas a um intervalo temporal específico, como o último mês, por meio do operador `rt`;

Ao executar o comando `"lei q <consulta>"`, o LEI realiza a busca da mensagem correspondente com base na consulta definida e entrega as mensagens correspondentes em uma pasta local (*Maildir*) ou em uma caixa postal remota via IMAP.

De forma resumida, o LEI simplifica significativamente o acesso seletivo às listas de discussão relevantes, ao permitir que desenvolvedores tenham apenas os e-mails de interesse por meio da sintaxe de consultas do Xapian. Além disso, a ferramenta mantém o controle das mensagens já processadas, evitando o download de e-mails previamente entregues em execuções anteriores.

Capítulo 6

O software B4

O B4[1] é um utilitário de linha de comando muito utilizado pelos desenvolvedores do Kernel Linux para automatizar e simplificar o fluxo de trabalho de desenvolvimento. Conforme discutido anteriormente, o processo de desenvolvimento do Kernel Linux depende da submissão de código na forma de *patches* enviados por e-mail. Esse modelo envolve múltiplas versões de um mesmo *patch*, ciclos sucessivos de revisão, a inclusão de assinaturas e *trailers* de validação, bem como o acompanhamento de *threads* de discussão associadas a cada modificação proposta.

Nesse contexto, o B4 foi concebido com o objetivo de abstrair a complexidade desse fluxo de trabalho. A ferramenta substitui cadeias extensas de comandos manuais, frequentemente suscetíveis a erros, por operações mais simples, consistentes e padronizadas, reduzindo o esforço cognitivo do desenvolvedor e aumentando a confiabilidade do processo de manipulação de *patches*.

Para desenvolvedores responsáveis por receber, revisar e aplicar *patches* em suas *branches* Git, o B4 se mostra particularmente vantajoso. A ferramenta permite a recuperação automática de *threads* completas a partir dos arquivos de listas de discussão, a comparação entre diferentes versões de um mesmo *patch* (v1, v2, v3, etc.) e a extração estruturada das informações relevantes contidas nos e-mails. Entre essas informações destacam-se tanto os metadados quanto elementos fundamentais do corpo das mensagens, como os *trailers* de revisão e o conteúdo propriamente dito dos *diffs* associados aos *patches*.

Essa capacidade de coletar e organizar dados de forma consistente foi essencial para o desenvolvimento deste trabalho, uma vez que exemplifica formas consistentes de extrair os metadados das listas de discussão.

[1] https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/mricon/b4

Capítulo 7

O que são trailers

De modo geral, *trailers* são linhas padronizadas de metadados adicionadas ao final da mensagem de *commit* do Git ou ao corpo de um *patch* submetido por e-mail. Sua finalidade é registrar, de forma explícita e estruturada, o histórico de participação, responsabilidade e validação dos diferentes colaboradores envolvidos em um determinado *patch*.

No contexto do desenvolvimento do Kernel Linux, os *trailers* desempenham um papel central, pois fornecem um registro formal de autoria, revisão e aprovação das contribuições. Esse mecanismo é fundamental para sustentar o modelo de desenvolvimento distribuído adotado pelo projeto, bem como para garantir conformidade com o *Developer Certificate of Origin* (DCO), que estabelece as condições legais para a submissão e redistribuição do código.

Do ponto de vista sintático, os *trailers* seguem um padrão de pares Chave: Valor, normalmente no formato:

```
Nome-do-Trailer: Nome Completo do autor <email@exemplo.com>
```

Essa padronização permite que ferramentas automatizadas rastreiem e processem o fluxo de um *patch* ao longo das listas de discussão, desde sua submissão inicial até sua eventual integração ao repositório principal. Por convenção(Git Documentation, 2025), os *trailers* são posicionados ao final da mensagem de *commit*, após a descrição detalhada da alteração, e geralmente separados do restante do texto por uma linha em branco e pelo delimitador `"---"`, utilizado em *patches* enviados por e-mail.

Uma vez que o *patch* é aplicado, os *trailers* passam a integrar permanentemente os metadados do *commit*, tornando-se visíveis e auditáveis por meio de comandos como `git log`. Dessa forma, preserva-se um histórico das decisões técnicas e das responsabilidades associadas a cada modificação.

No desenvolvimento do Kernel Linux, os *trailers* mais relevantes(Linux Documentation, 2025e) estão diretamente associados à autoria, à responsabilidade técnica e à validação formal do código, os principais são:

- `Signed-off-by`: Trata-se do *trailer* mais essencial e, em praticamente todos os casos, obrigatório. Ele indica que o autor concorda explicitamente com os termos do DCO

e declara possuir o direito de submeter o código sob a licença do projeto, permitindo seu uso e redistribuição.

- `Acked-by`: Utilizado para sinalizar que um mantenedor ou revisor concorda com a alteração, ou com parte dela, ainda que não tenha realizado uma revisão aprofundada ou testes extensivos. É comumente usado por mantenedores de subsistemas para aprovar mudanças que impactam suas áreas de responsabilidade.
- `Reviewed-by`: Indica que o *patch* foi efetivamente revisado por um colaborador, que analisou o código de forma detalhada e considera que ele cumpre seu propósito, está correto do ponto de vista técnico e segue as diretrizes de codificação do Kernel Linux.
- `Tested-by`: Aponta que o *patch* foi testado com sucesso em um ambiente específico, fornecendo evidência prática de que a alteração funciona conforme esperado.
- `Suggested-by / Reported-by`: Esses *trailers* são utilizados para atribuir crédito a colaboradores que sugeriram a modificação ou relataram o defeito que motivou a criação do *patch*, reforçando a transparência e o reconhecimento das contribuições indiretas ao desenvolvimento.

Em conjunto, os *trailers* constituem um mecanismo para a rastreabilidade e a auditabilidade do Kernel Linux, permitindo que o histórico de cada contribuição seja compreendido de forma clara e verificável ao longo do tempo.

Capítulo 8

Metodologia de extração e armazenamento de emails

O objetivo central deste trabalho se resume na extração integral dos e-mails disponíveis na plataforma `lore.kernel.org`, com o propósito de disponibilizar essa massa de dados de forma pública em um formato tabular, adequado à realização de consultas e análises.

Para alcançar esse objetivo, definiu-se uma metodologia composta por quatro etapas principais. Inicialmente, buscou-se identificar um mecanismo eficiente para extrair e armazenar localmente os e-mails hospedados na plataforma *lore*. Em seguida, realizou-se à interpretação do conteúdo desses arquivos em formato textual bruto. A terceira etapa envolveu o desenvolvimento de um processo de *parsing* para transformar os dados extraídos em uma estrutura mais organizada, bem como a aplicação de tratamentos e normalizações sobre esses dados estruturados. Por fim, os dados resultantes foram organizados em um formato tabular, visando otimizar consultas e selecionar um formato de armazenamento apropriado para essa finalidade.

8.1 Extração de e-mails

Durante a investigação das possíveis abordagens para a extração e o armazenamento local dos e-mails arquivados no *lore*, foram identificadas diferentes estratégias. A primeira delas consistiu na extração direto via HTTPS na instância do `lore.kernel.org`, utilizando a ferramenta LEI (*Local Email Interface*), descrita anteriormente na seção 5. Essa ferramenta tem como objetivo simplificar a aquisição de e-mails por meio de comandos de terminal relativamente simples.

Entretanto, esse método mostrou-se pouco adequado para a extração em larga escala. A plataforma *lore* impõe limitações de segurança que interrompem *downloads* superiores a aproximadamente 100 MB, o que inviabiliza a obtenção completa de grandes volumes de dados por meio dessa abordagem. Em razão disso, essa estratégia foi descartada como método principal de extração direta. Ainda assim, a ferramenta LEI não foi abandonada, sendo reaproveitada em uma etapa posterior do processo.

Diante dessas limitações, optou-se por uma segunda abordagem, o *download* direto

dos repositórios de e-mails do Kernel Linux disponibilizados pela plataforma *lore* em formato Git. A própria documentação[1] do *lore* descreve como qualquer usuário pode clonar esses repositórios (Figura 8.1) e configurar um servidor local que espelha o conteúdo disponibilizado publicamente.

```
git clone --mirror https://lore.kernel.org/gfs2/0 gfs2/git/0.git

# If you have public-inbox 1.1+ installed, you may
# initialize and index your mirror using the following commands:
public-inbox-init -V2 --ng dev.linux.lists.gfs2 \
  gfs2 ./gfs2 https://lore.kernel.org/gfs2 \
  gfs2@lists.linux.dev
public-inbox-index ./gfs2
```

Figura 8.1: *Exemplo de instrução no Lore para realizar o mirror dos dados.*

Internamente, a plataforma *lore* é baseada no software public-inbox, conforme discutido na seção 4. O funcionamento desse sistema consiste, essencialmente, na extração dos e-mails armazenados nos repositórios Git e na criação de referências para cada mensagem em um banco de dados *full-text search*, com o objetivo de viabilizar consultas eficientes. Nesse contexto, comandos como `public-inbox-init` são responsáveis por configurar o servidor local com informações como o caminho do repositório Git, a versão do software utilizada e a URL de serviço dos e-mails, enquanto o comando `public-inbox-index` realiza a indexação das mensagens no banco de dados de busca.

Essa abordagem foi a escolhida neste trabalho para viabilizar a extração completa dos e-mails. Inicialmente, realizou-se o *download* do repositório de interesse. Em seguida, configuraram-se as informações essenciais do public-inbox, procedeu-se com a indexação das mensagens e, por fim, iniciou-se um servidor executando em *localhost*. A partir desse ambiente local, a ferramenta LEI foi novamente utilizada, desta vez não para acessar diretamente o servidor remoto do *lore*, mas sim o servidor public-inbox localmente configurado. Essa estratégia permitiu a extração integral dos e-mails desejados, os quais foram armazenados no diretório `Maildir/cur` em formato *plain text*, viabilizando as etapas subsequentes de processamento e análise.

8.2 Interpretação dos dados contidos nos e-mails

Após a extração e o armazenamento local de todos os e-mails, tornou-se necessário compreender a estrutura interna dessas mensagens e definir uma estratégia para extrair informações relevantes a partir de arquivos em formato de texto simples. Para isso, foi fundamental analisar o padrão de estruturação de e-mails no formato `.eml`[2] extensão comumente utilizada para representar mensagens individuais contendo, em um único arquivo, todo o conteúdo do e-mail, bem como lidar com os diferentes esquemas de

[1] fonte: https://lore.kernel.org/gfs2/_/text/mirror/

[2] é um padrão de arquivo de texto simples usado para salvar e-mails individuais, contendo todo o conteúdo (cabeçalho, corpo, anexos, formatação) de uma mensagem, seguindo as normas RFC 822/5322.

codificação (*encoding*) presentes nas mensagens, de modo a garantir a conversão adequada para o padrão UTF-8.

A interpretação dos arquivos no formato `.eml` parte do entendimento de que esses arquivos podem ser logicamente divididos em duas seções principais.[3] A primeira corresponde ao cabeçalho (*header*), que contém informações essenciais da mensagem, tais como remetente (`From`), destinatário (`To`), assunto (`subject`), identificadores únicos (`Message-ID`), referências a mensagens anteriores no caso de respostas (`In-Reply-To`), além de metadados técnicos, como o esquema de codificação utilizado (*charset*). A segunda seção corresponde ao corpo da mensagem (*body*), que, no contexto de submissão de *patches*, foco deste trabalho, foi subdividida em três partes conceituais: (*i*) a mensagem textual do contribuidor, na qual o autor descreve a motivação e o contexto da alteração; (*ii*) a seção de *trailers*, cuja relevância é significativa por permitir rastrear revisões, aprovações e interações entre desenvolvedores; e (*iii*) o próprio *patch*, que contém o código-fonte proposto para integração.

Diferentemente do cabeçalho, que segue uma estrutura relativamente rígida baseada em pares chave–valor, o corpo do e-mail não possui um formato estritamente padronizado. Por se tratar de uma mensagem livre, não há delimitadores universais que definam, de forma explícita, o início e o fim de cada uma de suas seções. Essa ausência de estrutura formal representou um desafio substancial para a extração automática e consistente das informações contidas no corpo das mensagens.

Inicialmente, buscou-se desenvolver algoritmos de *parsing* baseados na estrutura conhecida dos arquivos `.eml`. Nessa abordagem, o cabeçalho é interpretado como um conjunto de campos no formato `Chave: Valor`, enquanto o corpo da mensagem é identificado a partir da primeira quebra de linha em branco que separa o cabeçalho do conteúdo textual. A partir desse padrão, tentou-se construir um algoritmo capaz de abranger o maior número possível de variações de e-mails, com o objetivo de estruturar adequadamente as informações extraídas.

Entretanto, essa estratégia mostrou-se limitada diante da grande diversidade de formatos e estilos presentes nos e-mails arquivados no *lore*, o que dificultou a construção de um *parser* universal e robusto. Diante dessa complexidade, optou-se pela utilização da biblioteca Email,[4] nativa da linguagem Python, que oferece suporte completo para o *parsing* e a estruturação dessas mensagens. Essa biblioteca interpreta o conteúdo do e-mail e o encapsula em uma estrutura de dados do tipo EmailMessage, abstraindo grande parte das particularidades sintáticas do formato `.eml`. Para evitar erros decorrentes de decodificação prematura, os arquivos são inicialmente lidos em formato binário, essa abordagem é crucial porque as mensagens de e-mail no *lore* frequentemente utilizam diferentes esquemas de codificação (*encodings*)[5] em uma mesma estrutura. Um único e-mail pode apresentar cabeçalhos em US-ASCII, um corpo de mensagem em UTF-8 e fragmentos de *patches* ou anexos em ISO-8859-1, permitindo que a biblioteca consiga fazer a interpretação corretamente do conteúdo.

---

[3] fonte: https://datatracker.ietf.org/doc/html/rfc822

[4] https://docs.python.org/3/library/email.html

[5] https://en.wikipedia.org/wiki/Character_encoding

Outro desafio relevante enfrentado durante o desenvolvimento do *parser* foi a identificação e o tratamento adequado dos diferentes esquemas de codificação presentes nas mensagens. Como as contribuições ao Kernel Linux provêm de desenvolvedores distribuídos globalmente, é comum que tanto o corpo quanto determinados campos do cabeçalho utilizem codificações distintas do UTF-8, como o padrão ISO-8859-1. Esse problema foi igualmente mitigado por meio da biblioteca Email do Python, que identifica automaticamente o campo *chaset*[6] (Figura 8.2) especificado nos metadados da mensagem e realiza a decodificação apropriada para UTF-8, garantindo a uniformização do texto para as etapas subsequentes de processamento e análise.

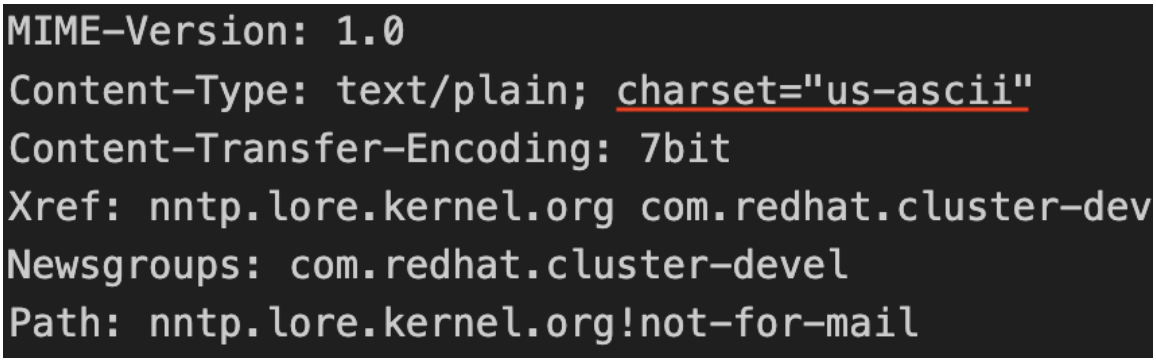

Figura 8.2: *Exemplo do campo chaset preenchido.*

8.3 Parser dos dados

Conforme discutido na seção anterior, a utilização da biblioteca Email da linguagem Python foi fundamental para a correta interpretação e estruturação dos cabeçalhos das mensagens, bem como para a identificação precisa do corpo (*body*) e o tratamento adequado da codificação (*enconding*). No entanto, para os objetivos deste trabalho, tornou-se também relevante a extração de informações específicas contidas no corpo do e-mail, em especial os *trailers* e o código propriamente dito, normalmente apresentados na forma de um `git diff`.

De modo geral, os e-mails que contêm submissões de *patches* seguem um formato relativamente padronizado(Linux Documentation, 2025c). Inicialmente, a mensagem apresenta um texto introdutório no qual o desenvolvedor descreve as alterações propostas e as motivações para sua implementação. Em seguida, encontra-se a seção de *trailers*, normalmente iniciada por um campo `Signed-off-by`, que referencia o autor do commit e formaliza a concordância com o *Developer Certificate of Origin*. Por fim, na parte final do corpo do e-mail contém o *patch* em si (Figura 8.3), geralmente precedido por um delimitador padrão, como uma linha iniciada por `"---"`.

Com base nessa estrutura, buscou-se selecionar e extrair os dados que fossem relevantes tanto no contexto de mensagens de e-mails quanto no contexto específico do desenvolvimento do Kernel Linux. Assim, para a definição do esquema de dados do *dataset*, foram utilizados campos do cabeçalho presentes em praticamente todos os e-mails, tais

[6] RFC que define esse padrão: https://www.ietf.org/rfc/rfc2045.txt

```
56   The stacktrace map can be easily full, which will lead to failure in
57   obtaining the stack. In addition to increasing the size of the map,
58   another solution is to delete the stack_id after looking it up from
59   the user, so extend the existing bpf_map_lookup_and_delete_elem()
60   functionality to stacktrace map types.
61
62   Signed-off-by: Tao Chen <chen.dylane@linux.dev>
63   ---
64    include/linux/bpf.h   |  2 +-
65    kernel/bpf/stackmap.c | 16 ++++++++++++++--
66    kernel/bpf/syscall.c  |  8 ++++++---
67    3 files changed, 20 insertions(+), 6 deletions(-)
68
69   diff --git a/include/linux/bpf.h b/include/linux/bpf.h
70   index 8f6e87f0f3a..9d6f7671ba1 100644
71   --- a/include/linux/bpf.h
72   +++ b/include/linux/bpf.h
```

Figura 8.3: *Exemplo de e-mail que segue um padrão bem definido de body.*

como `From`, `To`, `Cc`, `Subject` e `Date` (Figura 8.4). Esses campos foram representados como *strings*, com exceção do campo `Date`, que foi convertido para o tipo Datetime, visando facilitar consultas temporais.

No que diz respeito à identificação única das mensagens e às relações de encadeamento entre e-mails, foram definidos três campos adicionais: `message-id`, que representa o identificador único da mensagem; `in-reply-to`, que indica o e-mail ao qual a mensagem atual responde; e references, que aponta para o e-mail ou conjunto de *patches* ao qual a discussão está associada. Conforme explicado na seção 1, é comum que desenvolvedores submetam conjuntos de *patches* por meio de um e-mail inicial que fornece uma visão geral das alterações, seguido por mensagens individuais contendo cada *patch* (*patchset*) (Figura 8.5). Nesse contexto, o campo references desempenha um papel central, pois permite vincular não apenas os e-mails de resposta a um *patch* específico, mas também ao e-mail introdutório que contextualiza a modificação como um todo.

Com o objetivo de otimizar consultas e análises posteriores, como já foi dito, o corpo do e-mail foi subdividido em três campos distintos. O primeiro, denominado `raw_body`, armazena o conteúdo completo do corpo da mensagem. O segundo, chamado `code`, contém exclusivamente o código do *patch*, isto é, o `git diff` associado à modificação proposta. Por fim, o campo *trailers* que foi definido como uma lista de estruturas (*structs*), nas quais cada elemento armazena, o tipo do *trailer* (por exemplo, `Signed-off-by`, `Tested-by`, `Reviewed-by`) e as informações do desenvolvedor associado, incluindo nome e endereço de e-mail (Figura 8.6).

No que se refere ao processo de extração dessas informações, inicialmente optou-se pela utilização de expressões regulares (regex[7]), para identificar padrões específicos ao longo

[7] regex é uma sequência de caracteres que define um padrão de busca para encontrar, validar ou manipular

```
From: Tao Chen <chen.dylane@linux.dev>
To: ast@kernel.org,
  daniel@iogearbox.net,
  john.fastabend@gmail.com,
  andrii@kernel.org,
  martin.lau@linux.dev,
  eddyz87@gmail.com,
  song@kernel.org,
  yonghong.song@linux.dev,
  kpsingh@kernel.org,
  sdf@fomichev.me,
  haoluo@google.com,
  jolsa@kernel.org
Cc: bpf@vger.kernel.org,
  linux-kernel@vger.kernel.org,
  Tao Chen <chen.dylane@linux.dev>
Subject: [PATCH bpf-next v5 2/3] selftests/bpf: Refactor stacktrace_map
Date: Wed, 24 Sep 2025 00:58:48 +0800
Message-ID: <20250923165849.1524622-2-chen.dylane@linux.dev>
In-Reply-To: <20250923165849.1524622-1-chen.dylane@linux.dev>
References: <20250923165849.1524622-1-chen.dylane@linux.dev>
```

Figura 8.4: *Exemplo de um header de e-mail.*

```
Thread overview: 14+ messages (download: mbox.gz follow: Atom feed
-- links below jump to the message on this page --
2025-12-03  8:23 [PATCHv4 bpf-next 0/9] ftrace,bpf: Use single direct ops for bpf trampolines Jiri Olsa
2025-12-03  8:23 ` [PATCHv4 bpf-next 1/9] ftrace,bpf: Remove FTRACE_OPS_FL_JMP ftrace_ops flag Jiri Olsa
2025-12-03  9:15   ` Menglong Dong
2025-12-03 20:23     ` Jiri Olsa
2025-12-03  8:23 ` [PATCHv4 bpf-next 2/9] ftrace: Make alloc_and_copy_ftrace_hash direct friendly Jiri Olsa
2025-12-03  8:23 ` [PATCHv4 bpf-next 3/9] ftrace: Export some of hash related functions Jiri Olsa
2025-12-03  8:23 ` [PATCHv4 bpf-next 4/9] ftrace: Add update_ftrace_direct_add function Jiri Olsa
2025-12-03  8:47   ` bot+bpf-ci
2025-12-03 20:25     ` Jiri Olsa
2025-12-03  8:23 ` [PATCHv4 bpf-next 5/9] ftrace: Add update_ftrace_direct_del function Jiri Olsa
2025-12-03  8:23 ` [PATCHv4 bpf-next 6/9] ftrace: Add update_ftrace_direct_mod function Jiri Olsa
2025-12-03  8:24 ` [PATCHv4 bpf-next 7/9] bpf: Add trampoline ip hash table Jiri Olsa
2025-12-03  8:24 ` [PATCHv4 bpf-next 8/9] ftrace: Factor ftrace_ops ops_func interface Jiri Olsa
2025-12-03  8:24 ` [PATCHv4 bpf-next 9/9] bpf,x86: Use single ftrace_ops for direct calls Jiri Olsa
```

Figura 8.5: *Ilustração de uma hierarquia de e-mails exibida pela plataforma Lore. O e-mail mais acima é uma mensagem de apresentação de mudanças, seguindo temos na ordem de indentação os patchs em si com as trocas de mensagens logo abaixo.*

do corpo do e-mail. Contudo, essa abordagem mostrou-se limitada, uma vez que, apesar da existência de um formato predominante, há uma grande quantidade de mensagens que apresentam variações sutis, como diferenças na posição dos elementos, uso de caracteres não padronizados ou pequenas alterações na formatação (Figura 8.7). Considerando que o objetivo deste trabalho é catalogar e estruturar o maior número possível de e-mails, tornou-se evidente que um conjunto restrito de expressões regulares não seria suficiente para lidar com a diversidade presente nos dados do *lore*.

Diante dessa limitação, optou-se por estudar o funcionamento do software B4, que já

textos.

Coluna	Tipo	Descrição
from	string	Endereço de e-mail do remetente
to	Lista de String	Lista de e-mails para o envio
cc	Lista de String	Lista de e-mails de copia
subject	String	Assunto do e-mail
date	Datetime	Data e horário que o autor enviou o e-mail
message-id	String	Identificador único do e-mail
in-reply-to	String	message-id que o e-mail está respondendo
references	String	message-id de outros e-mails referenciados pelo atual
raw_body	String	Corpo do e-mail
code	String	Código em formato git diff
trailers	Lista de Structs	Struct que guarda o tipo da assinatura e o nome e e-mail do contribuídor

Figura 8.6: *Esquema de dados do dataset.*

disponibiliza mecanismos consolidados para a extração de *trailers* e de código a partir de e-mails de *patch*. A partir da análise de seu código-fonte[8] e da replicação parcial de sua lógica, foi possível aprimorar significativamente as funções de extração desenvolvidas neste trabalho. Essas melhorias incluíram a aplicação de etapas adicionais de padronização e normalização do texto, o uso de expressões regulares mais especializadas para tratar *corner cases*, bem como a substituição e limpeza de caracteres que dificultavam a identificação correta dos elementos, seguindo estratégias semelhantes às empregadas pelo B4.

8.4 Organização dos dados em formato colunar

Após o processo de tratamento e padronização dos dados, buscou-se uma forma eficiente de disponibilizar o *dataset* de maneira pública. O uso do formato CSV[9] foi descartado devido

[8] https://archive.softwareheritage.org/browse/content/sha1_git:3d774f70c026a84a380798e7d1133a5686d4b371/?origin_url=https://github.com/mricon/b4&path=src/b4/__init__.py

[9] é um formato de arquivo de texto simples usado para armazenar dados tabulares (como planilhas ou bancos de dados), onde cada linha representa um registro e os campos (colunas) são separados por vírgulas, facilitando a troca de dados entre diferentes programas.

```
Those revisions listed above that are new to this repository have
not appeared on any other notification email; so we list those
revisions in full, below.

- Log -----------------------------------------------------------------
commit a2699239ed1ba3537865b5dcbeb160bf3d5ecfc9
Author: David Teigland <teigland@redhat.com>
Date:   Thu Jul 10 13:45:50 2008 -0500

    fenced/dlm_controld/gfs_controld: ccs/cman setup

    Consistently set up and clean up ccs and cman.

    Signed-off-by: David Teigland <teigland@redhat.com>

-----------------------------------------------------------------------

Summary of changes:
 fence/fenced/fd.h                |    1 +
 fence/fenced/main.c              |    5 +-
 fence/fenced/member_cman.c       |    6 +-
 group/dlm_controld/action.c      |   69 ++++++++++++++++-----------
 group/dlm_controld/config.c      |  101 ++++++++++++++++++++++++++++++++++++
 group/dlm_controld/config.h      |    8 +-
 group/dlm_controld/dlm_daemon.h  |   14 +++--
 group/dlm_controld/group.c       |    5 ++
 group/dlm_controld/main.c        |   67 ++++++++++++++++++++++++
 group/dlm_controld/member_cman.c |   38 +++++++++++---
```

Figura 8.7: *Exemplo de e-mail que não segue o padrão proposto.*

ao elevado volume de dados coletados, o que tornaria tanto o armazenamento quanto o processamento pouco eficientes. Diante desse cenário, optou-se pela utilização de um formato de armazenamento colunar, projetado especificamente para análise e manipulação de grandes volumes de dados.

Diferentemente de formatos orientados a linhas, como CSV ou planilhas eletrônicas, nos quais os dados são armazenados registro a registro, os formatos colunares organizam as informações por coluna. Em outras palavras, todos os valores de um mesmo atributo, como `From`, `To`, `Cc`, entre outros, são armazenados de forma contígua. Essa organização favorece significativamente a compressão dos dados, uma vez que valores pertencentes à mesma coluna tendem a apresentar alta similaridade, o que potencializa a eficiência dos algoritmos de compressão. Além disso, formatos colunares armazenam metadados estatísticos, como valores mínimos e máximos por bloco, o que permite a leitura seletiva dos dados e reduz o volume de informações que precisa ser carregado durante a execução de consultas analíticas.

Entre os formatos colunares mais amplamente utilizados destacam-se Parquet e ORC, ambos projetos *open source* mantidos pela Apache Software Foundation. Para este trabalho,

optou-se pelo uso do formato Parquet por dois motivos principais. Primeiramente, trata-se de um formato amplamente adotado pela indústria e pela comunidade acadêmica, o que resulta em maior disponibilidade de documentação e ferramentas de suporte. Em segundo lugar, o Parquet apresenta excelente desempenho para leitura de dados altamente comprimidos(Ivanov e Pergolesi, 2020), característica alinhada com o volume e a natureza do *dataset* gerado neste projeto.

Além disso, foi adotada a estratégia de *Hive Partitioning* (ou particionamento Hive), que consiste em uma convenção de organização dos arquivos em diretórios hierárquicos, baseada nos valores de uma ou mais colunas do conjunto de dados. Nesse modelo, os arquivos Parquet são armazenados em uma estrutura de diretórios que reflete diretamente os valores das colunas de particionamento.

Essa abordagem contribui de forma significativa para a eficiência das consultas, pois permite que os mecanismos de processamento de dados realizem *partition pruning*,[10] isto é, acessem apenas os subconjuntos relevantes dos dados, evitando a leitura desnecessária de arquivos que não atendem aos critérios da consulta (Figura 8.8).

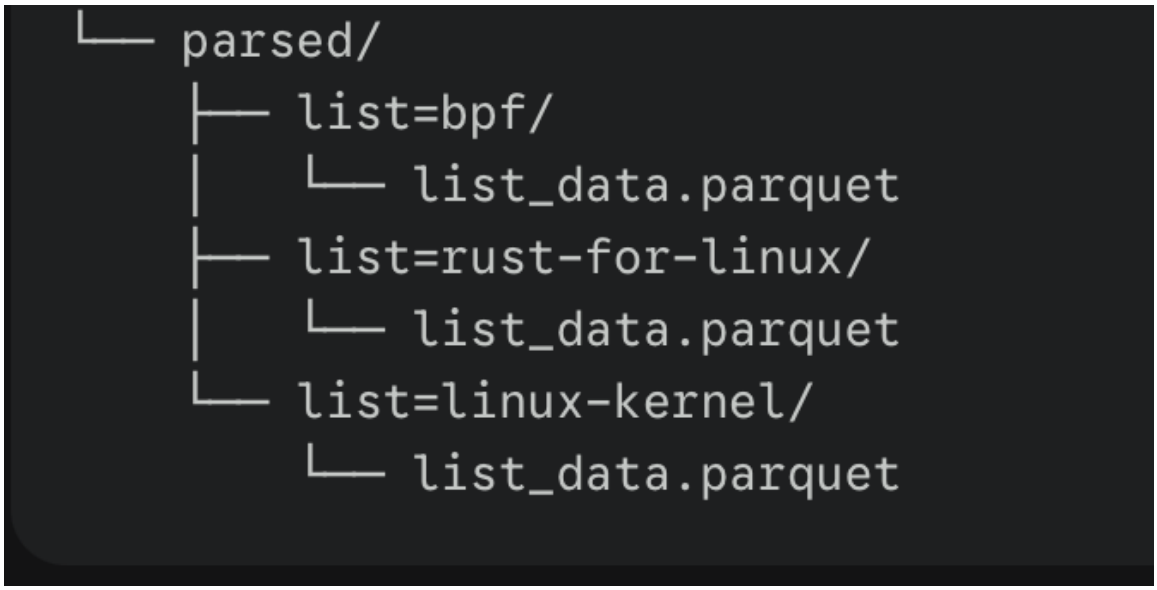

Figura 8.8: *Ilustração da organização do Hive partition de acordo com as listas de discussão.*

Como consequência, esse tipo de organização viabiliza consultas analíticas mais eficientes, como, por exemplo (Programa 8.1):

Assim, o motor de busca não precisa percorrer todos os dados, ele apenas irá na partição que interessa.

[10] https://docs.oracle.com/en/database/oracle/oracle-database/21/vldbg/partition-pruning.html\#GUID-E677C85E-C5E3-4927-B3DF-684007A7B05D

Programa 8.1 Exemplo de query feita em Python carregando o arquivo Parquet.

```
import polars as pl

dataset = pl.scan_parquet("parsed/list=*/list_data.parquet")
ctx = pl.SQLContext(emails=dataset)

# Note que 'list' se torna uma coluna disponível por causa das pastas
query = """ SELECT * FROM emails WHERE list = 'bpf' AND date >= '2023-01-01'
    AND date <= '2023-12-31' """

resultado = ctx.execute(query).collect() print(resultado)
```

Capítulo 9

Como executar o projeto

Para executar o projeto, é necessário inicialmente realizar o *clone* do repositório do **código-fonte**.[1] Além disso, o ambiente no qual o projeto será executado deve possuir os pacotes **Make** e **Docker** previamente instalados e corretamente configurados.

Após a instalação das dependências, o próximo passo consiste na configuração do arquivo `config.yml` (Figura 9.1), localizado na raiz do projeto. Esse arquivo é responsável por definir os parâmetros de execução, incluindo a lista de discussão do Kernel Linux da qual os e-mails serão extraídos, bem como os filtros aplicados para a recuperação das mensagens. Esses filtros seguem a mesma lógica e sintaxe das queries utilizadas pela ferramenta LEI, conforme descrito na seção 5 deste trabalho.

config.yml 375 B

```yaml
# nome da pasta que quer salvar
# destination: "~/Mail/lore"

# nome do modulo, digite all para pesquisar em todos
repository: "bpf"
# repository: "gfs2"

# filtro para fazer a pesquisa no lore, exemplo:
#   (dfn:drivers/block/floppy.c OR dfhh:floppy_* OR s:floppy
#   OR ((nq:bug OR nq:regression) AND nq:floppy))
#   AND rt:1.month.ago..
filter_query: >
  rt:1.years.ago..

```

Figura 9.1: *Exemplo de configuração do arquivo config.yml.*

Com o arquivo de configuração devidamente ajustado, deve-se executar, a partir da raiz do projeto, o seguinte comando para a construção da imagem *Docker*:

[1] link do repositório: https://gitlab.com/dudiszzz/tcc

```
make build
```

Esse comando é responsável por preparar o ambiente de execução, incluindo a instalação das dependências necessárias e a configuração dos serviços utilizados ao longo do fluxo de extração e processamento dos dados.

Em seguida, o projeto pode ser executado por meio de dois modos distintos. O modo padrão, sem interação com o usuário, pode ser iniciado com o comando:

```
make run
```

Alternadamente, é possível executar o projeto em modo interativo utilizando o comando:

```
make run-it
```

Nesse modo, é apresentado um menu interativo (Figura 9.2) que permite a execução individual das diferentes etapas do fluxo de extração, processamento e armazenamento dos dados, possibilitando maior controle e flexibilidade durante a execução.

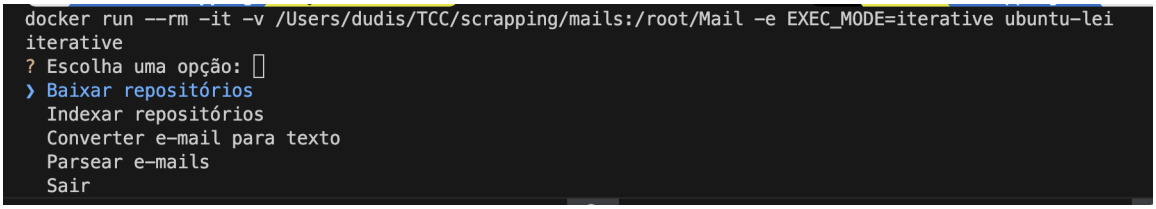

Figura 9.2: *Menu interativo do projeto.*

Por fim, o projeto disponibiliza um conjunto de testes unitários que podem ser executados com o comando:

```
make test
```

Para a execução dos testes, é necessário que o interpretador da linguagem Python esteja instalado localmente na máquina, uma vez que essa etapa não é executada dentro de um container *Docker*.

Capítulo 10

Resultados

O conjunto de dados resultante deste trabalho possui grande escala. Ao todo, foram coletados mais de 200 GB de dados brutos de e-mails, correspondentes a mais de 20 milhões de mensagens, que foram posteriormente interpretadas (*parsed*) e compactadas em aproximadamente 55 GB de arquivos em formato Parquet. O *dataset* cobre 345 listas de discussão, das quais cerca de 50% contêm mais de 13.000 e-mails, enquanto o quartil superior (25%) reúne mais de 50.000 mensagens por lista, evidenciando a elevada atividade de determinados subsistemas do kernel.

Esse volume expressivo de dados possibilita a realização de análises comparativas detalhadas entre subsistemas do Kernel Linux, bem como investigações aprofundadas sobre características fundamentais do seu processo de desenvolvimento e manutenção. Em particular, ele permite observar padrões de interação, revisão e suporte da comunidade que não são capturados por dados tradicionais de controle de versão.

Ao longo da última década, a comunidade do Kernel Linux tem manifestado preocupações recorrentes acerca de possíveis limitações de escalabilidade do modelo atual (*maintainership*) do projeto (Corbet, 2013; Dean, 2020; Edge, 2018; Vetter, 2017). Estudos recentes, baseados em revisões de literatura, sugerem que essas preocupações não se restringem a percepções individuais de colaboradores, mas podem refletir um problema estrutural mais profundo no modelo de desenvolvimento do Linux (Pinheiro e Meirelles, 2024; Tadokoro *et al.*, 2025; Wen, 2021).

Tanto este trabalho quanto os esforços anteriores de pesquisa buscam investigar empiricamente se há evidências que sustentem tais alegações de insustentabilidade, frequentemente associadas ao fenômeno conhecido como *maintainer overload*. Nesse contexto, foi recentemente desenvolvido o DUKS (*Dashboard for Unified Kernel Statistics*) (R. Passos *et al.*, 2025), com o objetivo de apoiar análises quantitativas e temporais do desenvolvimento do kernel. O *dataset* apresentado neste trabalho, denominado LKML5Ws, é particularmente relevante para esse tipo de investigação, pois permite capturar a quantidade significativa de esforço humano, interação social e discussão técnica que precede a aceitação de uma contribuição, aspectos que não são visíveis apenas por meio dos dados de *commits* armazenados no Git.

Como exemplo de aplicação do *dataset*, é possível analisar a atuação dos contribuidores

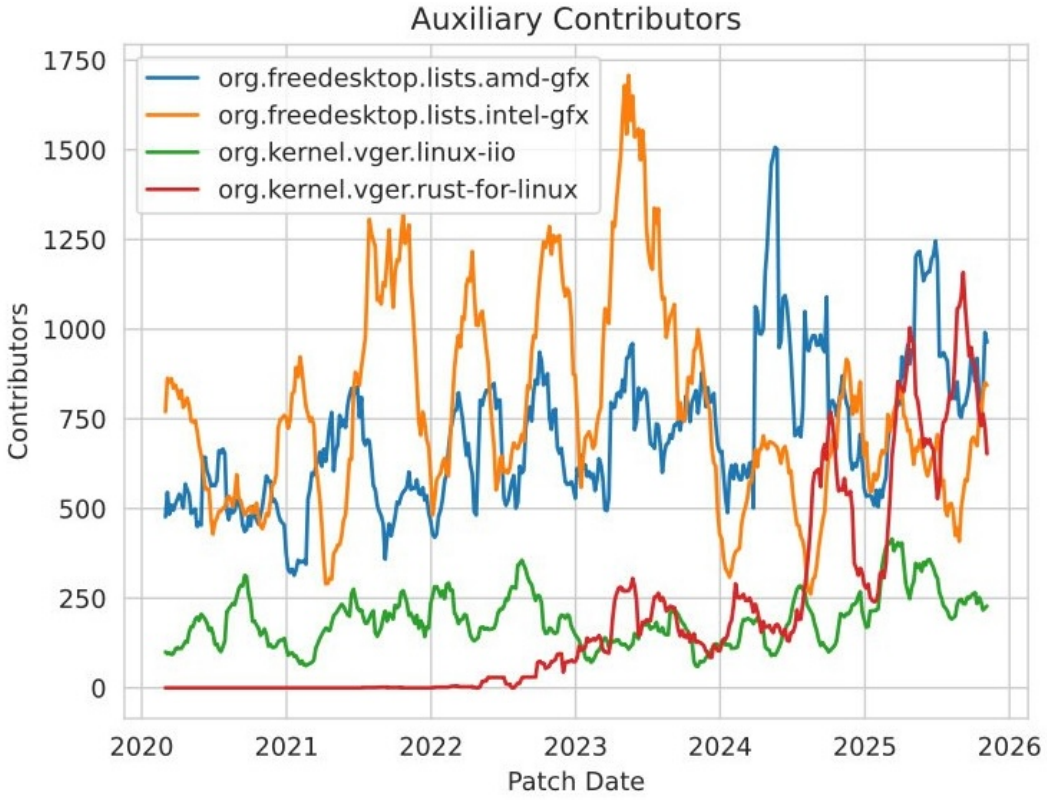

Figura 10.1: *Gráfico ilustrativo da variação do número de testadores e revisores ao longo do tempo.*

de apoio, como revisores e testadores, por meio da leitura da coluna *trailers*, que registra explicitamente *tags* como `Reviewed-by` e `Tested-by`. Além disso, a presença de múltiplas listas de discussão no conjunto de dados permite comparações entre subsistemas distintos do kernel.

Ao comparar as listas `linux-iio`, `amd-gfx`, `intel-gfx` e `rust-for-linux`, utilizando uma *sliding window* de dois meses ao longo dos últimos cinco anos, observa-se um comportamento heterogêneo na evolução da participação comunitária. Apesar das diferenças no número absoluto de colaboradores em cada lista, a figura 10.1 evidencia tendências divergentes no crescimento ou declínio do número de revisores e testadores desde 2020.

Os resultados indicam uma tendência de crescimento consistente na participação de revisores e testadores no subsistema `amd-gfx`, enquanto o subsistema `intel-gfx` apresenta uma tendência oposta. Para o subsistema `linux-iio`, os dados sugerem relativa estabilidade, com pouca variação na participação da comunidade ao longo do período analisado. Em contrapartida, a lista `rust-for-linux`, mais recente, demonstra uma rápida atração de colaboradores, refletindo o interesse crescente da comunidade na adoção da linguagem Rust no Kernel Linux.

Esses resultados ilustram como o *dataset* pode orientar pesquisadores na identificação de potenciais gargalos (*bottlenecks*) no processo de desenvolvimento do kernel. No caso específico do subsistema `linux-iio`, por exemplo, pode ser necessário aprofundar a análise

utilizando métricas complementares, como o número de *patches* submetidos, rejeitados e aprovados, a fim de avaliar se a aparente estabilidade na participação de revisores e testadores representa um risco à sustentabilidade do subsistema a médio ou longo prazo.

Capítulo 11

Trabalhos relacionados

Em consequência da incrível adoção e popularidade que o projeto do Kernel Linux foi conquistando ao longo dos anos, vários pesquisadores de engenharia de software fizeram estudos relacionados ao Kernel e seu modelo de desenvolvimento. Desde a caracterização do desenvolvimento de Software Livre (Balaguer *et al.*, 2017; Dias *et al.*, 2021) até servir como um estudo de caso ideal para ferramentas de análise de software (Occhipinti *et al.*, 2023; Suvorov *et al.*, 2012), o *kernel* tem servido amplamente à comunidade de Engenharia de Software. Além disso, o uso de listas de e-mail (*mailing lists*) como o principal meio de comunicação em projetos de Software Livre também tem sido foco de trabalhos anteriores (Guzzi *et al.*, 2013; Schneider *et al.*, 2016).

E por consequência, a comunidade de mineração de repositórios de software (*mining software repositories*) tem contribuído com *datasets* que apoiam estudos sobre projetos de Software Livre e o próprio kernel do Linux.

No sentido de caracterizar e compreender os colaboradores de Software Livre, Robles et al. (Robles *et al.*, 2014) realizaram uma pesquisa com mais de dois mil desenvolvedores para capturar informações demográficas, formações educacionais e profissionais e preferências pessoais de desenvolvimento.

Nas pesquisas de Passos, Czarnecki (L. Passos e Czarnecki, 2014) e German et al. (German *et al.*, 2015) criaram grandes conjuntos de dados que mostram a evolução da base de código do kernel do Linux. Enquanto Passos e Czarnecki trazem uma perspectiva única orientada a recursos (*feature oriented*), German et al. se concentram em agregar os dados do Git de múltiplas árvores em um repositório unificado, que eles chamam de super-repositório Git do Linux.

No entanto, nem Passos, Czarnecki ou German et al. exploram as listas de e-mail do Linux. O trabalho de Xu e Zhou (Xu e Zhou, 2018) é particularmente relevante para esse projeto, pois os autores criam um conjunto de dados com mais de 660 mil *patches* enviados por e-mail e armazenados no site Linux Patchwork(referenciar) e os *commits* Git aos quais esses *patches* se relacionam. Como o trabalho restringiu-se a apenas o arquivo Patchwork, isso limitou o conjunto de dados dos autores a menos de 120 listas de e-mail e a coleta de 9 anos de e-mails, enquanto esse trabalho se preocupou em coletar dados de mais de 15 anos de desenvolvimento.

Por fim, vale a pena ressaltar que o trabalho de Xu e Zhou não analisa informações relacionadas aos *trailers* e inclui apenas *threads* que contêm *patches*, enquanto nesse trabalho também incluí-se e-mails de discussão (*discussion threads*).

Capítulo 12

Conclusão

Embora as motivações iniciais deste trabalho fossem especificamente voltadas à criação de um conjunto de dados capaz de contribuir para investigações sobre gargalos decorrentes da sobrecarga de mantenedores, o *dataset* criado demonstra potencial para apoiar uma ampla gama de estudos futuros. Do ponto de vista de escala, a expressiva quantidade de e-mails coletados fornece uma boa base para análises aprofundadas dos padrões de comunicação entre desenvolvedores, bem como entre desenvolvedores, mantenedores e revisores.

Além disso, trabalhos futuros podem, por exemplo, investigar a prevalência e os impactos de comentários grosseiros e rudes nas listas de discussão do kernel Linux(Ehsani *et al.*, 2024; Gachechiladze *et al.*, 2017), buscando responder a questões como: existe correlação entre a ocorrência de interações rudes e a escassez de mantenedores em determinadas listas? A incivilidade tem aumentado ou diminuído ao longo do tempo? Além do mais, o conjunto de dados possibilita identificar quais tipos de problemas são mais frequentemente apontados durante o processo de revisão de código(Gonçalves *et al.*, 2025; Rahman *et al.*, 2025), como duplicação de código, complexidade excessiva ou questões de legibilidade.

Uma das contribuições mais relevantes do conjunto de dados LKML5Ws decorre da estrutura adotada para o corpo de cada e-mail, segmentado em três partes: uma seção de *trailers*, uma seção de código e o conteúdo inteiro da mensagem. Essa organização permite que estudos futuros explorem comparações entre métricas extraídas de diferentes *branches* e subsistemas do kernel, com o objetivo de analisar os impactos dos diferentes modelos de manutenção adotados ao longo da evolução do projeto.

De forma mais específica, o *dataset* viabiliza investigações como: quais subsistemas apresentam o menor número de revisores por *patch* submetido? Quais exibem a maior razão entre linhas de código modificadas e o número de mantenedores ativos? O comprimento médio da descrição dos *patches* varia de maneira significativa entre diferentes árvores do kernel?

Apesar de suas contribuições, o LKML5Ws apresenta algumas limitações. Em particular, determinados e-mails podem aparecer duplicados quando uma mesma mensagem é enviada para múltiplas listas de discussão hospedadas no `lore.kernel.org`. Esse aspecto pode ser tratado em extensões futuras do conjunto de dados. Conforme descrito na publicação do

DUKS(R. Passos *et al.*, 2025), trabalho que serviu de inspiração para este projeto, pretende-se integrar as informações extraídas das listas de discussão com dados do histórico de versões coletadas a partir do Software Heritage(Pietri *et al.*, 2019).

Essa integração poderá explorar o modelo de indexação baseado em *hash* adotado pelo Software Heritage, no qual as alterações de código são indexadas por meio de um ID exclusivo que usa um *hash* da própria alteração de código. Esse mecanismo impede o armazenamento de conteúdo duplicado e, quando aplicado ao corpo completo dos e-mails, permitirá a eliminação de mensagens redundantes no conjunto de dados LKML5Ws.

Além disso, extensões futuras podem explorar a flexibilidade da abordagem de coleta adotada neste trabalho para incluir mensagens provenientes de outros grandes projetos de Software Livre. Projetos do ecossistema GNU (como Emacs e GCC) e da Apache (como HTTPD e Superset) disponibilizam arquivos de listas de discussão no formato MBOX para seus diversos subprojetos. A incorporação desses dados, especialmente os mais antigos, é particularmente relevante, considerando que apenas cerca de 25% dos e-mails atualmente presentes no conjunto de dados foram enviados antes de 2014.

Por fim, espera-se que este trabalho de conclusão de curso, ao apresentar um conjunto de dados com mais de 20 milhões de e-mails distribuídos em 345 listas de discussão hospedadas no arquivo Kernel Lore, ofereça uma visão ampla e abrangente do processo de desenvolvimento do kernel Linux ao longo de sua história. Ao evidenciar no que consiste cada contribuição (**what**), quando ela foi inicialmente concebida (**when**), quem participou de sua revisão e aprovação (**who**), para qual lista foi submetida (**where**) e por que se tornou, ou não, parte do código do kernel (**why**), o **LKML5Ws** tem o potencial de alimentar pesquisas futuras sobre os aspectos técnicos e sociais que caracterizam um dos projetos de Software Livre mais representativos da atualidade.

Referências

[Balaguer *et al.* 2017] Federico Balaguer *et al.* "Assessing code authorship: the case of the linux kernel". In: *Open Source Systems: Towards Robust Practices*. Springer-Charm, 2017, pp. 151–163 (citado na pg. 41).

[Corbet 2013] Jonathan Corbet. *On saying "no"*. Out. de 2013. URL: lwn.net/Articles/571995/ (acesso em 05/11/2025) (citado na pg. 37).

[Dean 2020] Sam Dean. *The maintainer's paradox: balancing project and community*. Dez. de 2020. URL: https://www.linuxfoundation.org/blog/blog/the-maintainers-paradox-balancing-project-and-community (acesso em 05/11/2025) (citado na pg. 37).

[Dent 2003] Kyle D. Dent. *Postfix: The Definitive Guide: A Secure and Easy-to-Use MTA for UNIX*. Sebastopol, CA: O'Reilly Media, Inc., 2003. ISBN: 9781449378790 (citado na pg. 10).

[Dias *et al.* 2021] Edson Dias *et al.* "What makes a great maintainer of open source projects?" In: *International Conference on Software Engineering (ICSE)*. IEEE, mai. de 2021, pp. 982–994. DOI: 10.1109/ICSE43902.2021.00093 (citado na pg. 41).

[Edge 2018] Jake Edge. *Too many lords, not enough stewards*. Jan. de 2018. URL: lwn.net/Articles/745817/ (acesso em 05/11/2025) (citado na pg. 37).

[Ehsani *et al.* 2024] Ramtin Ehsani, Mia Mohammad Imran, Robert Zita, Kostadin Damevski e Preetha Chatterjee. "Incivility in open source projects: a comprehensive annotated dataset of locked github issue threads". In: *Mining Software Repositories Conference (MSR)*. ACM, 2024, pp. 515–519. ISBN: 9798400705878. DOI: 10.1145/3643991.3644887 (citado na pg. 43).

[Gachechiladze *et al.* 2017] Daviti Gachechiladze, Filippo Lanubile, Nicole Novielli e Alexander Serebrenik. "Anger and its direction in collaborative software development". In: *International Conference on Software Engineering: New Ideas and Emerging Results Track (ICSE-NIER)*. IEEE, 2017, pp. 11–14. ISBN: 9781538626757. DOI: 10.1109/ICSE-NIER.2017.18 (citado na pg. 43).

[German *et al.* 2015] Daniel M. German, Bram Adams e Ahmed E. Hassan. "A dataset of the activity of the git super-repository of linux in 2012". In: *Mining Software Repositories Conference (MSR)*. IEEE, 2015, pp. 470–473. ISBN: 9780769555942 (citado na pg. 41).

[Git Documentation 2025] Git Documentation. *git-interpret-trailers Documentation*. Documentação oficial do Git sobre manipulação de trailers em mensagens de commit. 2025. URL: https://git-scm.com/docs/git-interpret-trailers (acesso em 20/12/2025) (citado na pg. 23).

[Gonçalves *et al.* 2025] Pavlína Wurzel Gonçalves, Pooja Rani, Margaret-Anne Storey, Diomidis Spinellis e Alberto Bacchelli. "Code review comprehension: reviewing strategies seen through code comprehension theories". In: *International Conference on Program Comprehension (ICPC)*. 2025, pp. 589–601. DOI: 10.1109/ICPC66645.2025.00068 (citado na pg. 43).

[Guzzi *et al.* 2013] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger e Arie van Deursen. "Communication in open source software development mailing lists". In: *Mining Software Repositories Conference (MSR)*. IEEE, mai. de 2013, pp. 277–286. ISBN: 978-1-4673-2936-1 (citado na pg. 41).

[Ivanov e Pergolesi 2020] Todor Ivanov e Max-Petre Pergolesi. "The impact of columnar file formats on sql-on-hadoop engine performance: a study on orc and parquet". *Concurrency and Computation: Practice and Experience* 32.3 (2020), e5523. DOI: 10.1002/cpe.5523. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5523 (citado na pg. 33).

[Linux Documentation 2020] Linux Documentation. *Submitting patches: Sign your work - the Developer's Certificate of Origin*. Documentação oficial do Kernel Linux, versão 5.7. 2020. URL: https://www.kernel.org/doc/html/v5.7/process/submitting-patches.html#sign-your-work-the-developer-s-certificate-of-origin (acesso em 20/12/2025) (citado na pg. 5).

[Linux Documentation 2025a] Linux Documentation. *How the development process works: Next trees*. Documentação oficial do processo de desenvolvimento (linux-next). 2025. URL: https://www.kernel.org/doc/html/latest/process/2.Process.html#next-trees (acesso em 20/12/2025) (citado na pg. 6).

[Linux Documentation 2025b] Linux Documentation. *How the development process works: The Big Picture*. Documentação oficial sobre o fluxo de desenvolvimento do Kernel. 2025. URL: https://www.kernel.org/doc/html/latest/process/2.Process.html#the-big-picture (acesso em 20/12/2025) (citado na pg. 6).

[Linux Documentation 2025c] Linux Documentation. *How to Get Your Change Into the Linux Kernel (Submitting Patches)*. Guia oficial completo sobre o processo de submissão e revisão de patches. 2025. URL: https://www.kernel.org/doc/Documentation/process/submitting-patches.rst (acesso em 20/12/2025) (citado na pg. 28).

[Linux Documentation 2025d] Linux Documentation. *Linux kernel coding style*. Versão da documentação oficial do processo de desenvolvimento do kernel. 2025. URL: https://www.kernel.org/doc/Documentation/process/coding-style.rst (acesso em 20/12/2025) (citado na pg. 5).

[Linux Documentation 2025e] Linux Documentation. *Submitting patches: Using Reported-by, Tested-by, Reviewed-by, Suggested-by and Fixes*. Documentação oficial sobre atribuição de créditos e metadados em patches do Kernel. 2025. URL: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#using-reported-by-tested-by-reviewed-by-suggested-by-and-fixes (acesso em 20/12/2025) (citado na pg. 23).

[Linux Documentation 2025f] Linux Documentation. *Subscribing to Linux kernel mailing lists*. Instruções oficiais para participação nas listas de discussão do Kernel. 2025. URL: https://subspace.kernel.org/subscribing.html (acesso em 20/12/2025) (citado na pg. 9).

[Occhipinti *et al.* 2023] Gianlorenzo Occhipinti, Csaba Nagy, Roberto Minelli e Michele Lanza. "Syn: ultra-scale software evolution comprehension". In: *International Conference on Program Comprehension (ICPC)*. IEEE, mai. de 2023, pp. 69–73. DOI: 10.1109/ICPC58990.2023.00020 (citado na pg. 41).

[L. Passos e Czarnecki 2014] Leonardo Passos e Krzysztof Czarnecki. "A dataset of feature additions and feature removals from the linux kernel". In: *Mining Software Repositories Conference (MSR)*. ACM, 2014, pp. 376–379. ISBN: 9781450328630. DOI: 10.1145/2597073.2597124 (citado na pg. 41).

[R. Passos *et al.* 2025] Rafael Passos, Arthur Pilone, David Tadokoro e Paulo Meirelles. "Streamlining analyses on the linux kernel with duks". In: *Software Visualization Conference(VISSOFT)*. IEEE, 2025, pp. 125–128. DOI: 10.1109/VISSOFT67405.2025.00025 (citado nas pgs. 37, 44).

[Pietri *et al.* 2019] Antoine Pietri, Diomidis Spinellis e Stefano Zacchiroli. "The software heritage graph dataset: public software development under one roof". In: *Mining Software Repositories Conference (MSR)*. IEEE, mai. de 2019, pp. 138–142. ISBN: 978-1-7281-3412-3 (citado na pg. 44).

[Pinheiro e Meirelles 2024] Eduardo Pinheiro e Paulo Meirelles. "Understanding group maintainership model in the linux kernel development". In: *Software Visualization, Maintenance and Evolution Conference (VEM)*. SBC, 2024, pp. 113–124 (citado na pg. 37).

[Public-Inbox Documentation 2025a] Public-Inbox Documentation. *public-inbox-index - create and update search indices for public-inbox*. Documentação sobre a ferramenta de indexação e motor de busca do public-inbox. 2025. URL: https://public-inbox.org/public-inbox-index.html (acesso em 15/11/2025) (citado na pg. 17).

[Public-Inbox Documentation 2025b] Public-Inbox Documentation. *public-inbox-mda - mail delivery agent for public-inbox*. Documentação do agente de entrega de e-mail do sistema public-inbox. 2025. URL: https://public-inbox.org/public-inbox-mda.html (acesso em 15/11/2025) (citado na pg. 16).

[Public-Inbox Documentation 2025c] Public-Inbox Documentation. *public-inbox-v1-format - public-inbox v1 repository format*. Documentação técnica sobre o formato de repositório e indexação de mensagens. 2025. URL: https://public-inbox.org/public-inbox-v1-format.html (acesso em 15/11/2025) (citado na pg. 15).

[Public-Inbox Documentation 2025d] Public-Inbox Documentation. *public-inbox-v2-format - public-inbox v2 repository format*. Documentação técnica sobre o formato de armazenamento escalável para arquivos de mensagens. 2025. URL: https://public-inbox.org/public-inbox-v2-format.html (acesso em 15/11/2025) (citado na pg. 15).

[Rahman *et al.* 2025] Md Shamimur Rahman, Zadia Codabux e Chanchal K. Roy. "Investigating the understandability of review comments on code change requests". In: *Mining Software Repositories Conference (MSR)*. IEEE, 2025, pp. 539–551. DOI: 10.1109/MSR66628.2025.00087 (citado na pg. 43).

[Robles *et al.* 2014] Gregorio Robles, Laura Arjona Reina, Alexander Serebrenik, Bogdan Vasilescu e Jesús M. González-Barahona. "Floss 2013: a survey dataset about free software contributors: challenges for curating, sharing, and combining". In: *Mining Software Repositories Conference (MSR)*. ACM, mai. de 2014, pp. 396–399. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597129 (citado na pg. 41).

[Ryabitsev 2021] Konstantin Ryabitsev. *lore + lei, part 2: Now with IMAP*. Blog técnico sobre infraestrutura e ferramentas do Kernel Linux. Jan. de 2021. URL: https://people.kernel.org/monsieuricon/lore-lei-part-2-now-with-imap (acesso em 20/12/2025) (citado na pg. 19).

[Schneider *et al.* 2016] Daniel Schneider, Scott Spurlock e Megan Squire. "Differentiating communication styles of leaders on the linux kernel mailing list". In: *12th International Symposium on Open Collaboration*. ACM, ago. de 2016, pp. 1–10. ISBN: 978-1-4503-4451-7. (Acesso em 20/04/2025) (citado na pg. 41).

[Suvorov *et al.* 2012] Roman Suvorov, Meiyappan Nagappan, Ahmed E. Hassan, Ying Zou e Bram Adams. "An empirical study of build system migrations in practice: case studies on kde and the linux kernel". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 160–169. DOI: 10.1109/ICSM.2012.6405267 (citado na pg. 41).

[Tadokoro *et al.* 2025] David Tadokoro, Rodrigo Siqueira e Paulo Meirelles. "Can the linux kernel sustain 30 more years of growth? toward mitigating bottlenecks in its development model". In: *Brazilian Symposium on Software Engineering (SBES)*. SBC, 2025, pp. 845–851. DOI: 10.5753/sbes.2025.11607 (citado na pg. 37).

[Unix Documentation 2025] Unix Documentation. *maildir(5) - Linux manual page*. Documentação técnica sobre o formato de armazenamento de e-mail Maildir. 2025. URL: https://www.unix.com/man_page/linux/5/maildir/ (acesso em 20/12/2025) (citado na pg. 10).

[Vetter 2017] Daniel Vetter. *Maintainers don't scale*. Jan. de 2017. URL: blog.ffwll.ch/2017/01/maintainers-dont-scale.html (acesso em 05/11/2025) (citado na pg. 37).

[Wen 2021] Melissa Shihfan Ribeiro Wen. "What happens when the bazaar grows: a comprehensive study on the contemporary Linux kernel development model". Tese de dout. São Paulo, Brasil: University of São Paulo, 2021 (citado na pg. 37).

[Xapian Documentation 2025] Xapian Documentation. *Xapian Documentation: Stemming*. Documentação sobre algoritmos de radicalização para motores de busca. 2025. URL: https://xapian.org/docs/stemming.html (acesso em 15/11/2025) (citado na pg. 17).

[Xu e Zhou 2018] Yulin Xu e Minghui Zhou. "A multi-level dataset of linux kernel patchwork". In: *Mining Software Repositories Conference (MSR)*. IEEE, mai. de 2018, pp. 54–57. URL: https://ieeexplore.ieee.org/document/8595178/ (citado na pg. 41).