UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Lucas Moura Veloso

Extração de Data Provenance Por Meios Não-Intrusivos Para LIMS

Belo Horizonte
2024

Lucas Moura Veloso

Extração de Data Provenance Por Meios Não-Intrusivos Para LIMS

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Sergio Vale Aguiar Campos

Belo Horizonte
2024

Veloso, Lucas Moura.

V443e Extração de data provenance por meios não-intrusivos para LIMS [recurso eletrônico] / Lucas Moura Veloso. – 2024.

1 recurso online (83 f. il.) : pdf.

Orientador: Sérgio Vale Aguiar Campos.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência Da Computação.
Referências: f. 67-69.

1. Computação - Teses. 2. Bioinformática - Teses. I. Campos, Sérgio Vale Aguiar. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*93(043)

Ficha catalográfica elaborada por Célio Resende Diniz, bibliotecário CRB 6/2403 - Universidade Federal de Minas Gerais – ICEx.

UNIVERSIDADE FEDERAL DE MINAS GERAIS

EXTRAÇÃO DE LINHAGEM DE DADOS POR MEIOS NÃO-INSTRUSIVOS PARA LIMS

LUCAS MOURA VELOSO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Sérgio Vale Aguiar Campos - Orientador

Departamento de Ciência da Computação - UFMG

Prof. Adriano César Machado Pereira

Departamento de Ciência da Computação - UFMG

Prof. Mark Alan Junho Song

Departamento de Ciência da Computação - PUC-MG

Belo Horizonte, 28 de maio de 2024.

Documento assinado eletronicamente por **Sergio Vale Aguiar Campos**, **Professor do Magistério Superior**, em 27/11/2024, às 11:31, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Mark Alan Junho Song**, **Usuário Externo**, em

27/11/2024, às 14:41, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Adriano Cesar Machado Pereira**, **Professor do Magistério Superior**, em 27/11/2024, às 21:32, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3765225** e o código CRC **77943517**.

Referência: Processo nº 23072.269503/2024-35 SEI nº 3765225

Para Luísa.

Acknowledgments

À toda equipe do LUAR UFMG, em especial o meu orientador Dr. Sergio Vale Aguiar Campos, a Dra. Alessandra Faria Campos, e a equipe de desenvolvimento do Flux por todo o apoio e fundamentação que me foi dado.

À minha família e amigos pelo suporte durante este período.

Aos meus gatos pela companhia na escrita deste trabalho.

"*O que pode deter o coração determinado e a vontade decidida do homem?*"
(Mary Wollstonecraft Shelley)

Resumo

Ao criarmos sistemas de informação biomédicos e laboratoriais, a confiabilidade e rastreio da informação são requisitos essenciais, porém em muitos projetos estes são desenvolvidos fortemente acoplados com o funcionamento do sistema, e não necessariamente seguindo padrões para o bom funcionamento de uma cadeia de auditoria e rastreio.

O conceito de *Data Provenance* vem para formalizar e estruturar as informações a serem coletadas, ajudando a construir todo o lastro necessário para os trabalhos feitos com apoio destes sistemas.

O objetivo deste trabalho é desenvolver um sistema para extração de *Data Provenance* a partir de alterações em *workflows* de *Laboratory Information Management Systems* (LIMS), de modo a ter os ganhos de uma cadeia de informações concisa e confiável. Esta cadeia deverá ser extraida de forma não-intrusiva, ou seja, evitando ao máximo impactos na experiência do usuário e no tempo de resposta do sistema.

Palavras-chave: Linhagem de Dados; Auditoria de Software; Data Provenance; Data Pedigree.

Abstract

It is essential for biomedical and lab research software to be trustworthy when it comes to the data that they generate and handle, but in many projects the data reliability is developed from scratch, strongly coupled to the inner workings of the system, and not necessarily following the best practices for creating a healthy audit trail.

The concept of data provenance formalizes and structures the information that should be collected, adding to the trust of the research and development that is done with these systems.

The goal of this work is the development of a system to automatically generate data provenance information from the usage of workflows in Laboratory Information Management Systems (LIMS), to generate a trusted chain of data. The data extraction should be done in non-intrusive fashion, avoiding impacts in the user experience and in the system's response time.

Keywords: Data Provenance; Data Pedigree.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Neste capítulo trabalharemos alguns conceitos que serão necessários para o entendimento dos textos a seguir e contextualizaremos o leitor sobre o problema a ser resolvido, ao final discutiremos os objetivos que buscamos ao iniciar este trabalho.

1.1 Motivação

Sistemas laboratoriais são essenciais para a produção científica biomédica e seu desenvolvimento e melhorias constantes são de grande importância para o avanço destas e das demais áreas que possam se beneficiar deles.

Para um sistema poder ser utilizado seguramente para registro e manutenção em um laboratório, este deve possuir as seguintes características:

- Confiabilidade: o que existe registrado no sistema deve representar a verdade ocorrida durante o experimento;
- Reprodutibilidade: os dados coletados devem permitir que os experimentos realizados sejam reproduzíveis;
- Rastreabilidade: o sistema deve registrar informações sobre todos os passos, agentes, e materiais envolvidos em um experimento;
- Acessibilidade: os dados devem ser disponibilizados de forma legível e facilmente auditáveis;
- Confidencialidade: o sistema deve disponibilizar os seus dados apenas para usuários qualificados e com os níveis de acessos corretos.

Alguns exemplos de dados que devem ser registrados para um experimento cumprir os requisitos acima são:

- Rastreamento de lotes de insumos utilizados;
- Operações técnicas realizadas em cada passo do experimento;
- Data, hora, e local de execução de cada atividade;
- Profissionais e supervisores envolvidos em cada passo;
- Condições laboratoriais no momento da execução do experimento;
- Técnicas utilizadas na preparação e manuseio dos materiais.

Criar um sistema com estas características é uma tarefa complexa por ser necessária a construção de cadeias de informação precisas sobre cada atividade de fluxo de trabalho, as quais são de difícil identificação em sua maioria, e que nem sempre são executadas da mesma forma ou pelos mesmos profissionais, além de exigir iterações com profissionais multidisciplinares capacitados nas práticas laboratoriais que serão implementadas.

1.2 Conceituação

1.2.1 Trilha de histórico

No contexto deste trabalho, o termo trilha é utilizado para representar informações históricas de algum dado presente no sistema, este histórico pode ser representado de diversas formas, mas ele sempre deve mostrar como o dado transita entre estados, e, se possível, quais as operações e usuários envolvidos nessas alterações.

1.2.2 Auditoria de software

Alterações nos dados gerenciados por um *software* são vitais, muitas vezes adentrando a esfera jurídica, quando o assunto é confiabilidade e rastreabilidade do sistema.

Visando registrar o que acontece com os dados de um sistema, o campo da auditoria de *software* foi criado para:

> "*In relation to software, auditing can be defined as the location, detection and identification of all software within the subject installation. Primarily, we must ask ourselves the baseline questions of what, where and who?*" [10]

Tradicionalmente, as trilhas de auditoria de *software* são formadas de registros de alterações de uma determinada entidade, indicando quais dados foram alterados ao longo do tempo, e metadados adicionais, como, por exemplo, usuários envolvidos nestas alterações, localização geográfica de onde a requisição de alteração foi efetuada, data e hora da requisição, como exemplificado na figura 1.5.

O estado da arte atualmente consiste em modelos de auditoria onde os dados são obtidos, processados, validados, e, caso seja necessário, usuários são alertados sobre testes que falharam, tudo isso automatizadamente, como detalhado no estudo referenciado em [11].

1.2.3 Data Provenance

O termo *Data Provenance*, também denominado *Data Lineage* ou *Data Pedrigree*, indica a capacidade de acompanhar os dados e suas modificações. Existem algumas definições diferentes para *Data Provenance*, para este trabalho usaremos a definição da W3C:

> "*Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness.*" [17]

De forma simples, *Data Provenance* visa responder às perguntas de como e porque um dado gerado por algum usuário de sistema e alterado por este ou outros usuários chegou no seu estado atual.

Estas informações adicionais podem ser descritas de diversas formas e, temporalmente, padrões de metadados vêm sendo propostos para criar um consenso entre sistemas produtores e consumidores destes dados.

Propomos utilizar do conceito de *Data Provenance* para agregar confiabilidade aos sistemas, trazer trilhas de rastreabilidade e reprodutibilidade para os experimentos realizados, além de possibilitar a exibição destas informações de forma segura e legível.

1.2.4 Sistemas de Informação Biomédicos e LIMS

Sistemas de Informação Biomédicos (SIB) são sistemas computacionais cujo propósito é modelar, armazenar, recuperar e analisar dados biomédicos. Estes dados são especiais pelas características intrínsecas de alta complexidade e pelas suas características legais de privacidade e confidencialidade.

A sigla LIMS significa *Laboratory Information Management System*, que pode ser definido como:

> "Um LIMS (*Laboratory Information Management System*) é um método usado por um laboratório para gerenciar seus dados e para propagar os resultados para áreas específicas." [18]

Ainda segundo Hinton [18], um LIMS deve acompanhar todas as etapas de alteração de um dado, desde o momento em que ele é criado até o momento em que ele é divulgado ou arquivado, o que já demonstra a sinergia deste conceito com o de *Data Provenance* (ver seção 1.2.3).

Por definição, um LIMS não necessariamente é um sistema computacional, mas para efeitos práticos deste trabalho consideraremos que os exemplos de LIMS citados são.

As funcionalidades essenciais para os LIMS modernos são as que permitem a reprodução dos procedimentos seguidos no laboratório, além de facilidades para o dia a dia dos profissionais que utilizarão o sistema. Algumas destas funcionalidades estão listadas a seguir:

- Mecanismos para minimizar a entrada de dados incorreta pelo usuário;
- Mecanismos de validação dos dados inseridos no sistema;
- Mecanismos de recuperação de dados e consultas;
- Cálculos automáticos;
- Controle de acesso;
- Integração com outros sistemas do laboratório;
- Integração com os equipamentos do laboratório.

É fundamental que um LIMS produza e armazene dados de forma confiável, segura e de fácil acesso para usuários credenciados, já que alguma falha nestas características pode levar um experimento a ser um fracasso e inviabilizar o uso do sistema em ambientes produtivos.

Outro requisito dos LIMS modernos é que sejam auditáveis por terceiros, de modo a garantir a transparência do sistema. Estas trilhas de auditoria, porém, não podem ser elaboradas de forma simples, por exporem dados extremamente sensíveis ou dados críticos e confidenciais para a operação dos experimentos. Os dados disponibilizados pelos módulos de auditoria devem ser anonimizados sempre que possível, e para isso os metadados do sistema devem seguir padrões rígidos.

Estes sistemas podem ser usados por diversos laboratórios e diversas categorias de usuários, alguns exemplos de possíveis usuários finais estão citados abaixo para fins de ilustração:

- Laboratórios de Análises Clínicas;
- Laboratórios de Pesquisa Médica/Aplicada;
- Laboratórios de Pesquisa básica;
- Gestores da Qualidade;
- Clínicas/Consultórios Médicos.

1.3 Trabalhos Relacionados

Nesta seção iremos discutir alguns trabalhos relevantes para a produção deste.

1.3.1 TRANSFoRm

O projeto TRANSFoRm é um *Learning Health System* (um sistema de informação médica evolutivo, onde os usuários são tanto produtores quanto consumidores da informação, de modo a proverem acesso mais rapidamente a avanços da medicina) desenvolvido por instituições de diversos países da Europa pelo programa EU FP7, que visa criar uma infraestrutura e fomentar o conhecimento e uso de LHS nos países participantes.

Este projeto traz *Data Provenance* desde a sua concepção, como exemplificado nos trabalhos [7], [6], e [5], para habilitar o registro e gerenciamento de informação de forma segura e confiável entre os vários subsistemas que compõem a arquitetura do mesmo.

O projeto engloba diversas iniciativas, sendo a mais relevante para o trabalho um sistema de auxílio de diagnósticos médicos que implementa os conceitos de *Data Provenance*, tanto para melhorar o próprio sistema, quanto para manter um registro auditável de tudo que foi sugerido e de como estas sugestões foram utilizadas no tratamento dos pacientes.

Apesar de similares, este trabalho e o projeto TRANSFoRm se diferenciam em alguns pontos-chave, sendo estes:

- Uso dos *templates*, o projeto TRANSFoRM utiliza *templates* complexos para lidar com interações consecutivas de um usuário em uma mesma funcionalidade, que requerem que o usuário realize todas as interações de uma só vez, o que não foi considerado neste trabalho;

- Eventos registrados, neste trabalho coletamos dados sempre que houver uma mudança persistida no banco de dados, uma abordagem diferente da tomada pelo projeto TRANSFoRm, que os coleta somente quando certos eventos acontecem no sistema (como, por exemplo, uma sugestão de diagnóstico é exibida ao médico);

- Usos adicionais, neste trabalho usamos o sistema de *Data Provenance* para troca de mensagens e para habilitar a edição de *workflows* em tempo real, usos estes que não tem similaridades com nenhum que TRANSFoRm apresenta em seus estudos.

1.3.2 Sistemas de Informação Baseados em Workflows

Sistemas de informação baseados em *workflows* modelam e gerenciam fluxos de trabalho, humanos ou automatizados, para documentar a execução de um dado conjunto de processos.

XML Process Definition Language, ou XPDL, é um padrão desenvolvido pela *Workflow Management Coalition* (WfMC) [2] para estruturar dados que definem e representam processos reais, visando formalizar estas modelagens para as informações poderem ser intercambiáveis entre aplicações diversas que sejam compatíveis com o modelo.

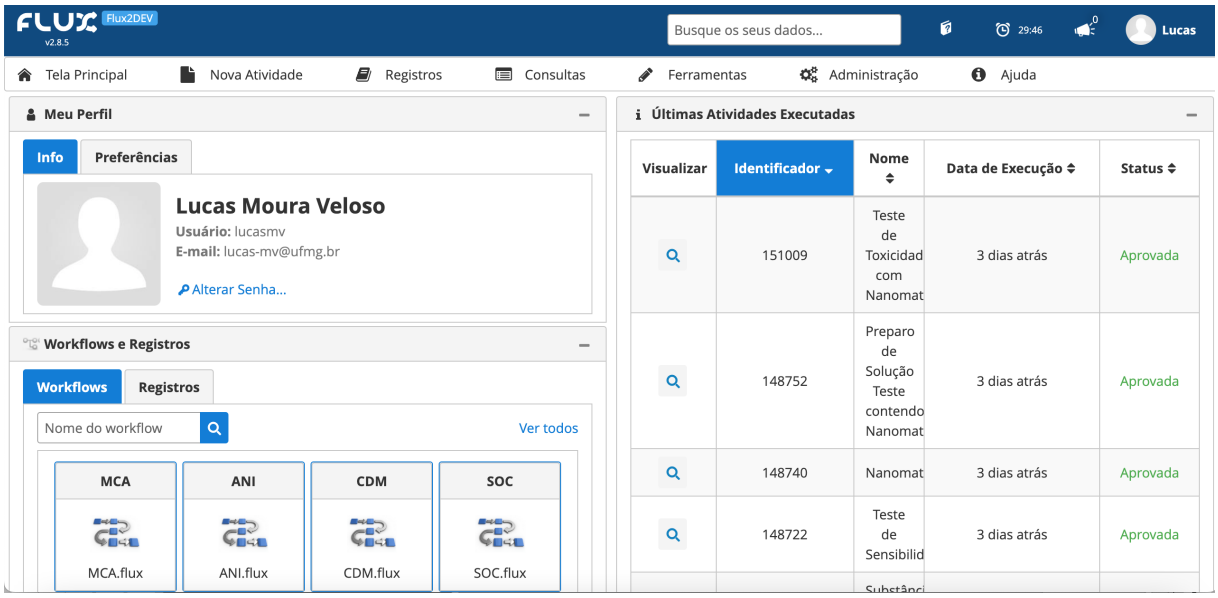
Historicamente, o padrão XPDL foi guiado com o intuito de sintetizar em dados os diagramas de *Business Process Management* [25], porém o formato também se mostrou útil para o trabalho com *workflows* em SIBs, como no exemplo do Flux e LIMS.

O padrão XPDL define várias entidades, porém para efeitos práticos sumarizaremos em quatro: *workflows*, processos, atividades, e transições, que serão descritos abaixo e mais formalmente nas referências [25] e [24].

- *Workflow* define um fluxo de trabalho completo e podemos entender ele como todo o conjunto que o XPDL representa
- Processos são agrupamentos lógicos de atividades, ou um conjunto de atividades que devem ser executados em um dado fluxo de trabalho
- Atividades são uma unidade básica de trabalho em um fluxo de negócio, são a representação computacional de algum trabalho que precisa ser executado
- Transições são conexões entre atividades que podem ou não estar atreladas com condições de execução das atividades

Um exemplo de sistema de informação baseado em *workflow* que já foi validado em ambientes médicos e laboratoriais (ver trabalhos [9] e [8]) é o sistema Flux.

Figura 1.1: Tela principal do Flux.



Fonte: Captura de tela do sistema Flux realizada pelo autor em abril de 2023.

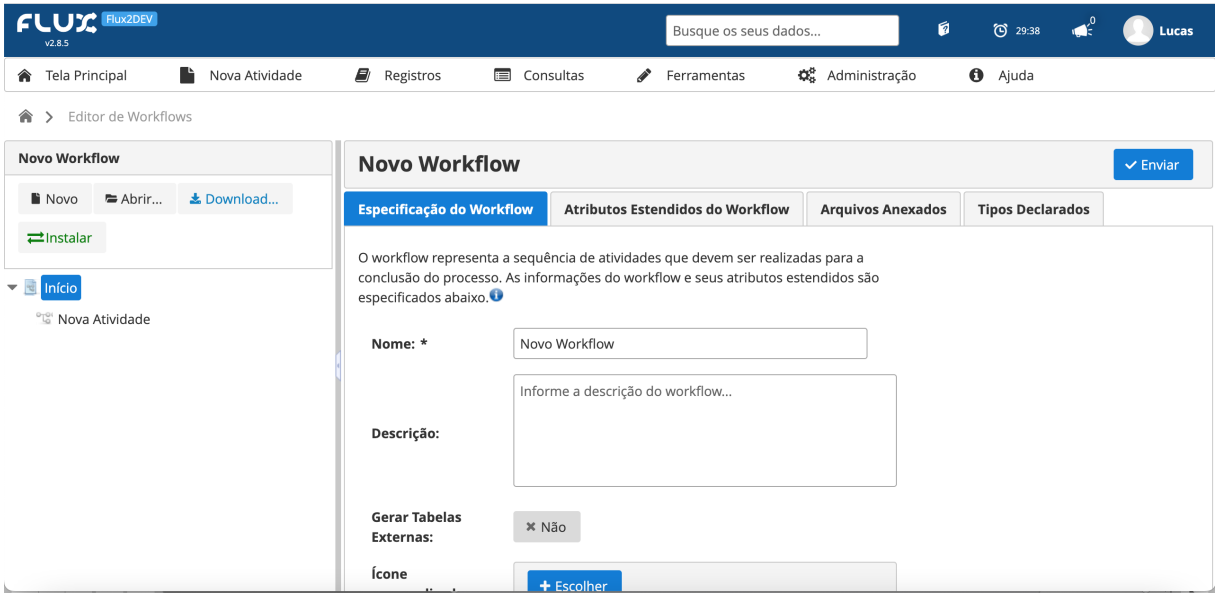
Flux, sistema descrito profundamente na referência [9], é um LIMS (ver seção 1.2.4) de uso geral disponibilizado como um sistema *web* originário do projeto de LIMS genérico e *open source* SIGLa (ver referência [22]).

Atualmente na sua versão 2, o Flux é um sistema completo e intuitivo, que contém diversas funcionalidades para além da construção e gerenciamento de *workflows*, como, por exemplo, *plug-ins* para execução de códigos de análise em R [13] e uma própria linguagem baseada em *JavaScript* para gerar informação a partir de um *workflow*, a FluxLang.

Além dos usos já citados, o sistema já foi utilizado para trabalhos com o INMETRO e para avaliação e triagem de pacientes com suspeita de COVID-19, mostrando ser extremamente versátil.

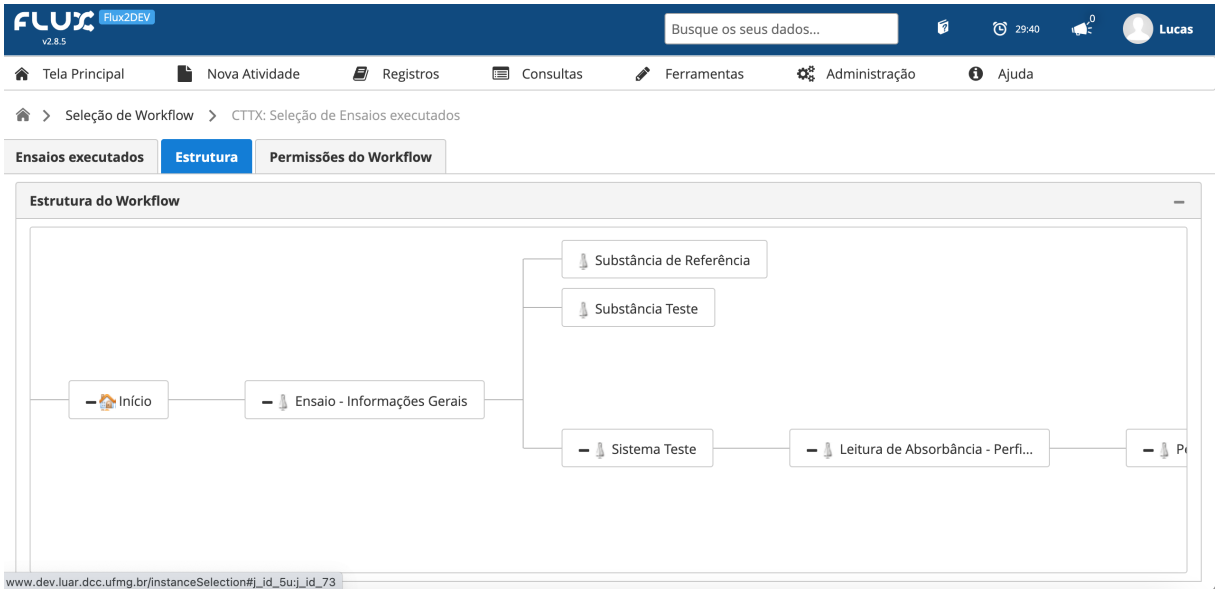
A figura 1.1 mostra a tela inicial do Flux, onde é mostrado um resumo de todas as operações recentes do sistema das quais o usuário possui acesso. Em seguida podemos criar e editar *workflows* na figura 1.2, e depois visualizá-los e executá-los na figura 1.3.

Figura 1.2: Tela de edição de *workflows* do Flux.



Fonte: Captura de tela do sistema Flux realizada pelo autor em abril de 2023.

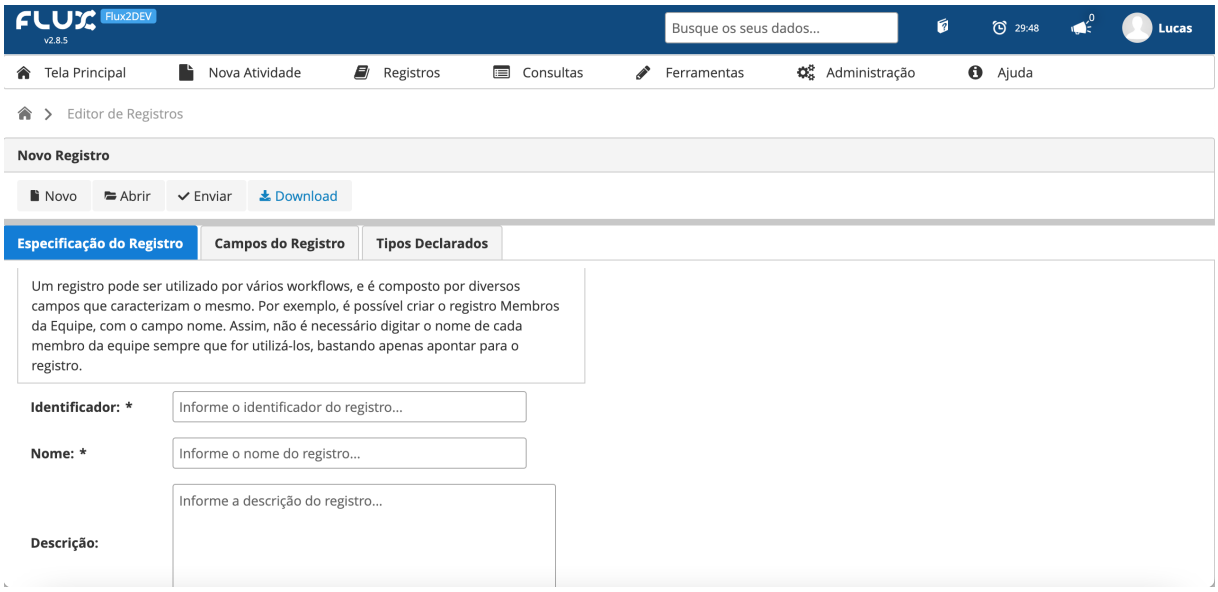
Figura 1.3: Tela de visualização de *workflows* do Flux.



Fonte: Captura de tela do sistema Flux realizada pelo autor em abril de 2023.

A edição e criação de registros do Flux é mostrada na figura 1.4. Registros são entidades fixas que podem ser referenciadas em diversas instâncias de um mesmo *workflow* de modo a facilitar o preenchimento do mesmo, por exemplo, se o atributo de uma atividade de um dado *workflow* for o CPF de uma pessoa responsável pelo sistema, podemos criar este CPF como um registro e apenas o referenciar nas execuções desta atividade.

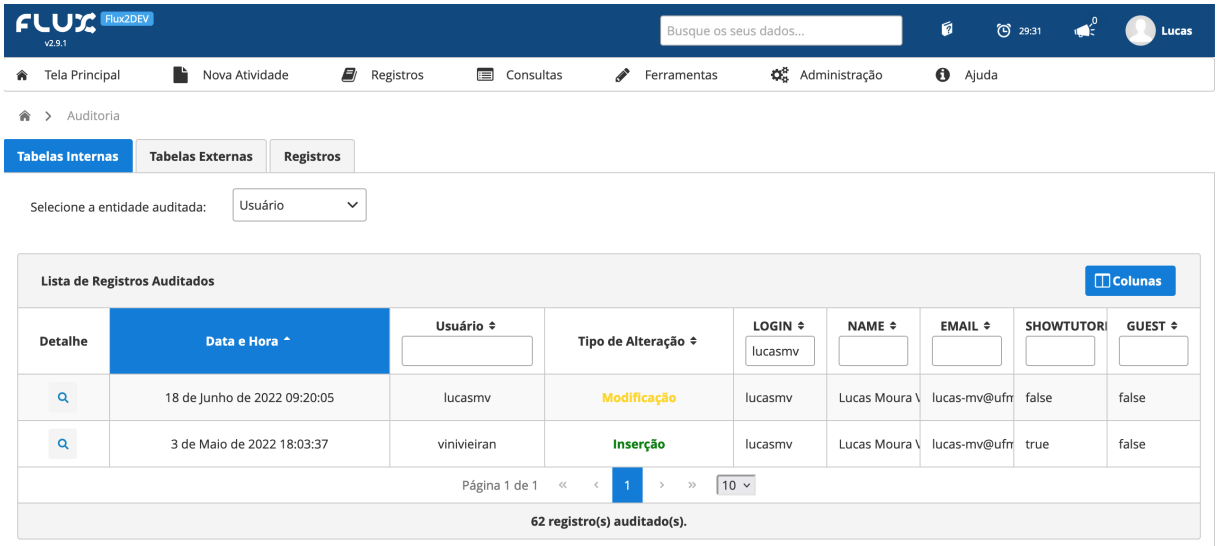
Figura 1.4: Tela de edição de registros do Flux.



Fonte: Captura de tela do sistema Flux realizada pelo autor em abril de 2023.

Apesar do Flux já incluir um módulo de auditoria antes deste trabalho, os dados coletados por este não possuem o contexto adicional que as trilhas de *Data Provenance* concedem, portanto, não permitem compreender os processos que levaram a cada alteração e nem como outras entidades foram envolvidas nessas alterações. Por este motivo faremos do Flux o objeto de integração e fonte de dados deste trabalho, sobre o qual construiremos toda a cadeia de *Data Provenance* necessária, baseando nas ideias e experiências apresentadas no projeto TRANSFoRm (ver seção 1.3.1).

Figura 1.5: Tela de auditoria do Flux.



Fonte: Captura de tela do sistema Flux realizada pelo autor em abril de 2023.

O projeto também prevê a inclusão de mensagens de texto que podem ser enviadas ao se realizar alguma alteração em um registro do banco de dados, de modo a adicionar

ainda mais contexto a essas mudanças. Estas mensagens deverão estar presentes nos metadados de *Data Provenance* quando estes forem extraídos.

1.3.3 Padrões para construção de trilhas de *Data Provenance*

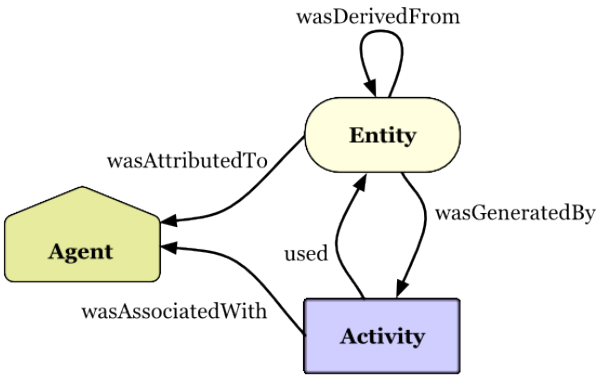
Dois padrões comuns para descrição dos metadados de *Data Provenance* são o Dublin Core [1] e o PROV [17], mas existem formatos criados para melhor descrever domínios específicos, como na norma ISO19115-2 [12] para dados geográficos.

Para este projeto focaremos no padrão PROV [17], tanto pela sua especificidade em tratar metadados de *Data Provenance*, quanto pela sinergia com sistemas baseados em *workflow*.

O padrão PROV consiste em uma série de recomendações do *World Wide Web Consortium* (W3C) para modelagem e armazenamento de dados de *Data Provenance* (ver seção 1.2.3). Por ser uma padronização de um modelo de dados, a ideia é que seja agnóstico à ferramenta que o implemente, tendo já sido desenvolvidas adaptações para XML (PROV-XML), adaptações específicas para o melhor entendimento humano (PROV-N), dentre outras exemplificadas em [17].

De forma básica, o padrão PROV visa colocar as informações em formatos de grafos orientados onde os vértices indicam objetos e as arestas indicam ações que conectam estes objetos.

Figura 1.6: Exemplo de uso do padrão PROV.



Fonte: [16].

A figura 1.6 exemplifica o uso do padrão PROV, onde podemos ver objetos sendo conectados entre si por meio das ações executadas por usuários ou processos.

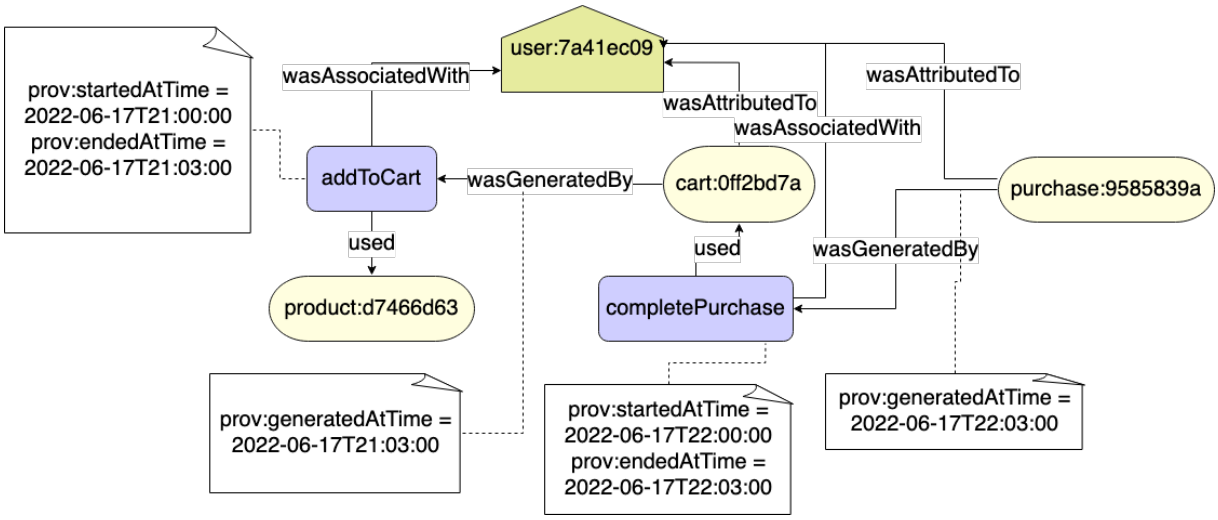
Existem três categorias de vértices em um grafo no padrão PROV, entidades, atividades, e agentes. Abaixo definiremos brevemente cada uma destas, além do papel das arestas, definições mais formais podem ser encontradas em [16] e [17].

- Entidades são quaisquer conceitos sobre os quais gostaríamos de traçar um rastro de *Data Provenance*, podem ser representações reais, ou apenas conceitos para os quais queremos tecer uma teia de registros.
- Atividades são os processos ou conjunto de processos que geram ou modificam entidades.
- Agentes são os realizadores de atividades, podem indicar usuários finais de algum sistema, ou sistemas intermediários.

As arestas do grafo indicam como uma entidade foi criada ou utilizada, e podem ter diversas propriedades conforme as necessidades do sistema. O padrão PROV contém uma lista de propriedades que podem compor os seus grafos descrita em [20]. É importante notar que essas propriedades sempre recorrem a verbos no passado, já que o intuito do modelo é prover informações sobre fatos já ocorridos.

O modelo também prevê anotações que podem ser inclusas em qualquer nível de um grafo, indicando metadados importantes como data da ocorrência, papel que algum agente possuía no sistema no momento da atividade, entre outros.

Figura 1.7: Exemplo de uso do padrão PROV para aplicações de *e-commerce*.



Fonte: Elaborado pelo autor.

A figura 1.7 mostra um exemplo de grafo de *Data Provenance* no padrão PROV contendo diversas entidades, atividades, agentes e anotações. A figura exemplifica como podemos aplicar o padrão PROV para aplicações de *e-commerce*, nela um usuário identificado como "7a41ec09" adiciona um produto "d7466d63" ao seu carrinho de compras, que recebe um identificador único "0ff2bd7a", e em seguida completa a compra criando o pedido "9585839a". Neste grafo também temos anotações associadas às atividades "addToCart" (adicionar ao carrinho) e "completePurchase" (completar compra) indicando seus respectivos horários de início e fim e anotações associadas às arestas identificadas

"wasGeneratedBy" (foi gerado por), indicando o momento em que uma dada entidade foi gerada no sistema.

Como o modelo é agnóstico à ferramenta de implementação ou ao sistema que o utiliza, é possível criar trilhas de *Data Provenance* muito completas que se estendem à fronteira de um sistema e tocam nos seus vizinhos, conforme exemplificado na figura 1.8. Nesta figura temos uma extensão para o grafo presente na figura 1.7 contendo dois subsistemas, uma aplicação de *e-commerce* e um sistema para gerenciamento de envios dos pedidos. A compra "9585839a" foi utilizada por um sistema automatizado de geração de pedidos "orderGeneratorSystem" para gerar o pedido "aa63a574" e este foi utilizado pelo sistema automatizado de notificação dos armazéns "warehouseNotifierSystem" através da atividade "notifyWarehouse" para notificar o armazém "5d666923" de um pedido que deve ser entregue para o usuário "7a41ec09".

Nos casos de gerarmos trilhas de *Data Provenance* entre diversos sistemas devemos nos manter atentos às necessidades de confidencialidade dos dados, como os nossos objetos de estudo (ver seção 1.2.4).

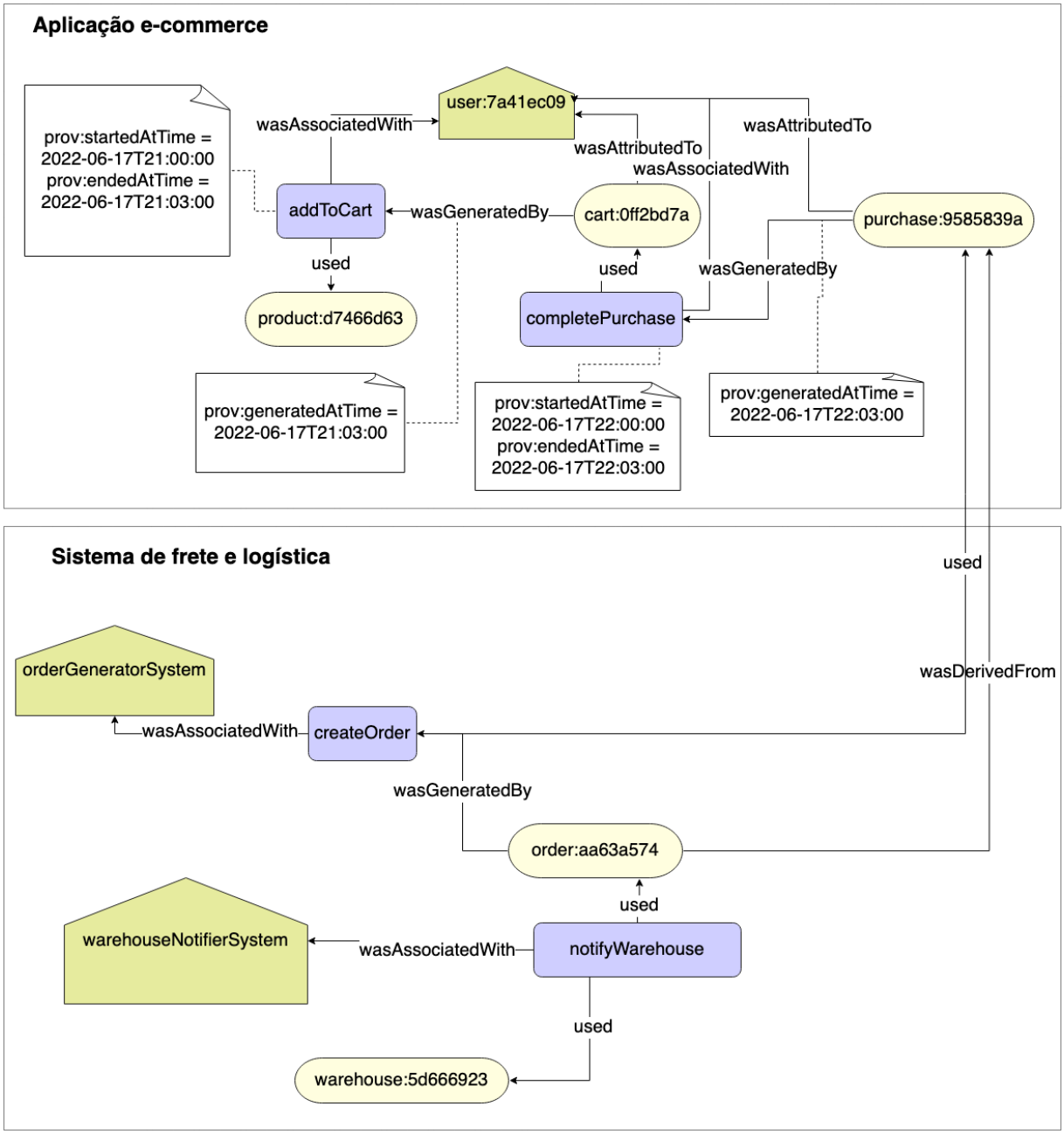
1.4 Objetivos do trabalho

O objetivo deste trabalho é a utilização de uma abordagem que possibilite a extração de trilhas de *Data Provenance* do sistema Flux a partir dos seus fluxos internos sem a necessidade de construção dessa funcionalidade diretamente nos módulos de interesse. Para tanto, as experiências relatadas no projeto TRANSFoRm 1.3.1 e os padrões para construção de trilhas de *Data Provenance* 1.3.3 serão muito valiosos.

Este trabalho tem como resultado principal melhorar a qualidade dos sistemas laboratoriais, trazendo os conceitos de *Data Provenance* para os principais pontos focais deste tipo de sistema:

- Confiabilidade, tudo que for executado no sistema será registrado nas cadeias de *Data Provenance* sem possibilidade de alteração externa ou maliciosa;
- Reprodutibilidade, os grafos compilados representarão todos os passos realizados nos *workflows*;
- Rastreabilidade, cada vértice dos grafos de *Data Provenance* deverá conter todas as informações necessárias para rastreio;
- Acessibilidade, os padrões de leitura e modelagem utilizados irão garantir que os dados sejam produzidos de forma acessível e que serão legíveis;

Figura 1.8: Exemplo de uso do padrão PROV atravessando as fronteiras entre sistemas.



Fonte: Elaborado pelo autor.

- Confidencialidade, os níveis de acesso para leitura das informações coletados serão mantidos.

Ao final deste trabalho teremos melhorias na forma de ler, interagir e compreender as alterações feitas no Flux pelos seus diversos perfis de usuários, e cadeias de *Data Provenance* entregues aos usuários de forma intuitiva e legível.

Também usamos as trilhas de *Data Provenance* para desenvolver duas funcionalidades adicionais no Flux que trouxeram melhorias na usabilidade do sistema:

- Envio de mensagens associadas às atividades: melhoria na operação do sistema tanto pelos usuários construtores de *workflow* quanto pelos usuários que os preenchem, além de contextualizar as informações auditáveis do sistema;
- Edição de *workflows* em tempo real: os usuários com permissão para tanto poderão realizar alterações na estrutura de um *workflow* sem afetar as instâncias preenchidas anteriormente.

É de interesse deste trabalho que o módulo produzido também seja genérico o suficiente para poder ser acoplado a outros ambientes, além dos SIBs e dos LIMS, de modo a viabilizar a inclusão de trilhas de *Data Provenance* de forma prática e rápida em diversos projetos.

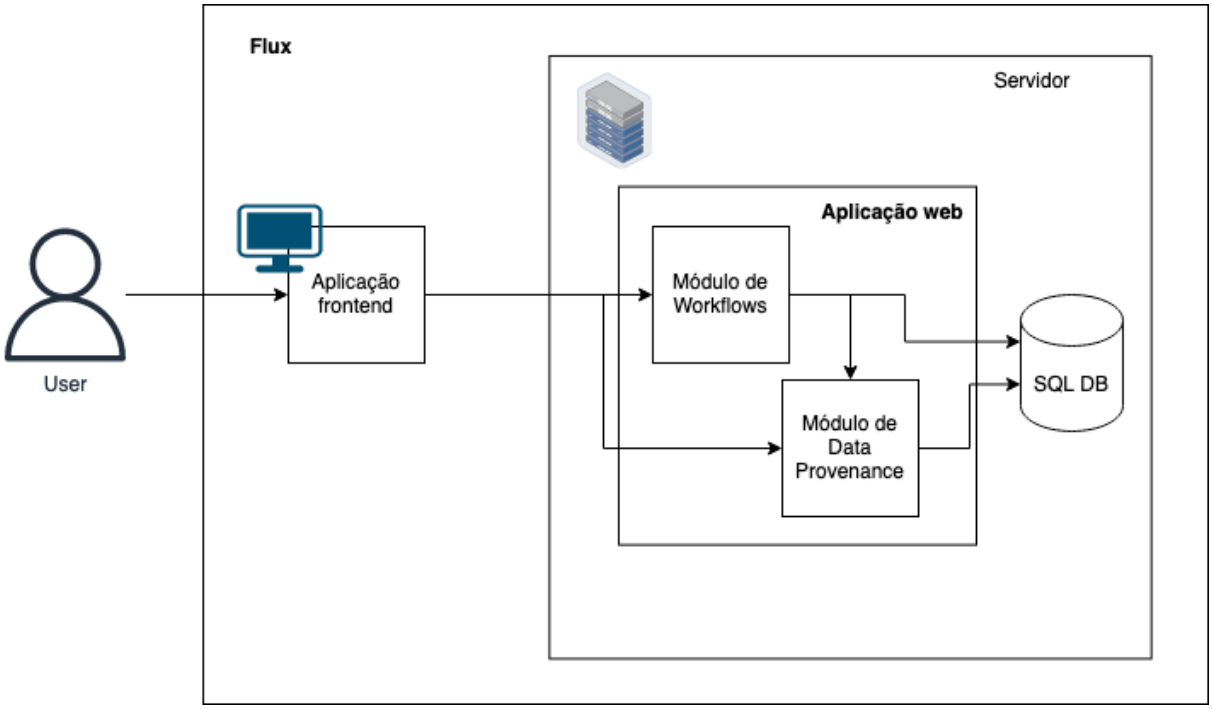
Capítulo 2

Metodologia e desenvolvimento

Neste capítulo abordaremos a metodologia aplicada e como o projeto foi desenvolvido, as decisões tomadas, e como elas afetam os resultados obtidos.

2.1 Especificação

Figura 2.1: Arquitetura básica do sistema proposto



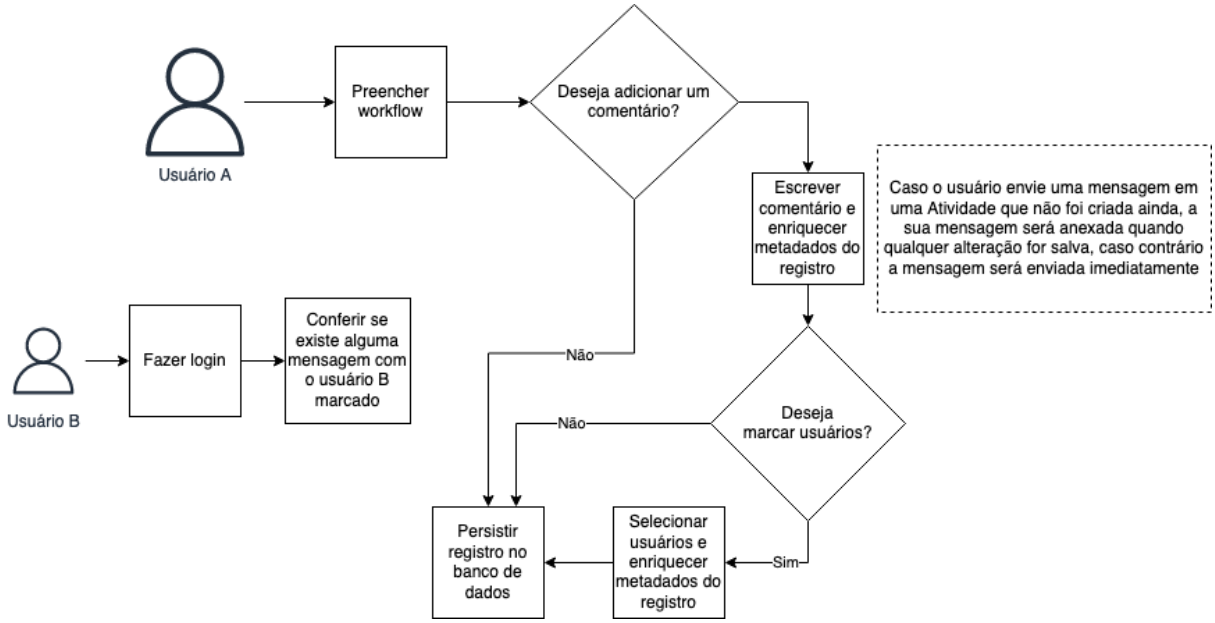
Fonte: Elaborado pelo autor.

A figura 2.1 mostra a arquitetura básica do sistema, ele foi desenvolvido como uma parte do sistema Flux e integrado aos seus módulos e ao seu *frontend* de modo que o usuário possa realizar consultas sobre as informações coletadas.

O sistema também conta com a funcionalidade de redigir mensagens de texto ao

realizar alguma alteração em um workflow, seja na sua estrutura ou no seu preenchimento, estas informações serão salvas como metadados de *Data Provenance* e permitem mencionar outros usuários do mesmo projeto para que estes fiquem cientes das alterações realizadas. A figura 2.2 indica como foi concebido o fluxo de adição de comentários e alerta de usuários.

Figura 2.2: Fluxograma da funcionalidade de comentários e menções de usuários.



Fonte: Elaborado pelo autor.

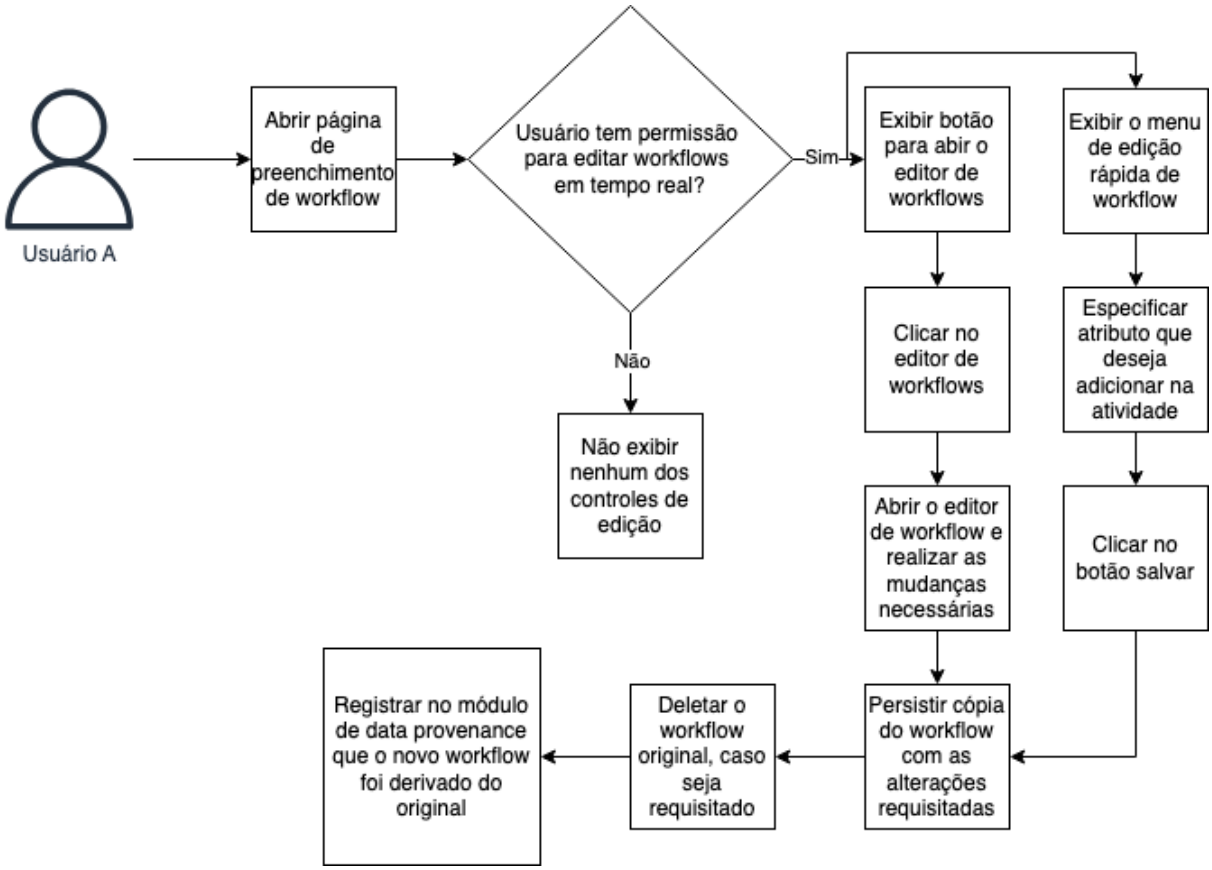
Como os registros de *Data Provenance* nos permitem rastrear interações entre entidades distintas, podemos então criar uma modalidade de edição de *workflows* em tempo real, onde o usuário poderá alterar um workflow durante a etapa do seu preenchimento, já que o sistema estará habilitado para nos indicar em que ponto a ramificação da estrutura do workflow ocorreu. A figura 2.3 indica o funcionamento básico deste fluxo proposto.

Para o modelo básico de *Data Provenance*, utilizaremos do padrão PROV discutido anteriormente e recorreremos ao padrão PROV-N [17] para disponibilizar as trilhas de informação de forma legível para usuários, também recorreremos às recomendações e experiências relatadas sobre a concepção da *Data Provenance* no sistema TRANSFoRm [5] para fundamentar este trabalho.

2.2 Experimentação

Para validar este sistema foi utilizada a versão mais recente do Flux, sendo que o módulo de *Data Provenance* deveria registrar informações necessariamente nos seguintes

Figura 2.3: Fluxograma da edição de *workflows* em tempo real.



Fonte: Elaborado pelo autor.

momentos:

- Criação ou edição de um *workflow*
- Preenchimento de uma atividade
- Criação de um registro do Flux

Com isso foi possível extrair os grafos de *Data Provenance* necessários para responder as seguintes perguntas essenciais:

- Como um determinado *workflow* chegou na forma em que se encontra hoje?
- Quais foram os responsáveis pela execução de uma determinada atividade?
- Como um determinado registro foi utilizado pelo sistema?

2.3 Módulo de Data Provenance

O módulo de *Data Provenance* é o núcleo deste projeto onde as operações de registro e leitura das trilhas de rastreio obtidas estão centralizadas. Para o módulo poder ser separado no futuro (ver seção 1.4), esta parte do sistema foi desenvolvida com o mínimo de dependências possíveis do Flux. A figura 2.4 e a seção 2.3.2 mostram como as dependências foram trabalhadas para que este objetivo fosse alcançado.

2.3.1 Funcionalidades básicas

De forma simplificada, o módulo contempla duas funcionalidades básicas, coletar dados de *Data Provenance* e reconstruir as trilhas de alguma entidade.

A coleta de dados se dá da seguinte forma, ao realizarmos uma operação no banco de dados, observamos qual foi o tipo de operação (criação, leitura, atualização, ou deleção), quais dados foram alterados, quem foi o usuário responsável pela alteração, em qual fluxo do sistema essa operação ocorreu, e depois persistimos estas informações no banco de dados.

Para a reconstrução da trilha, basta buscarmos nos dados salvos as alterações que envolvem a entidade a ser consultada e conectá-las uma a uma.

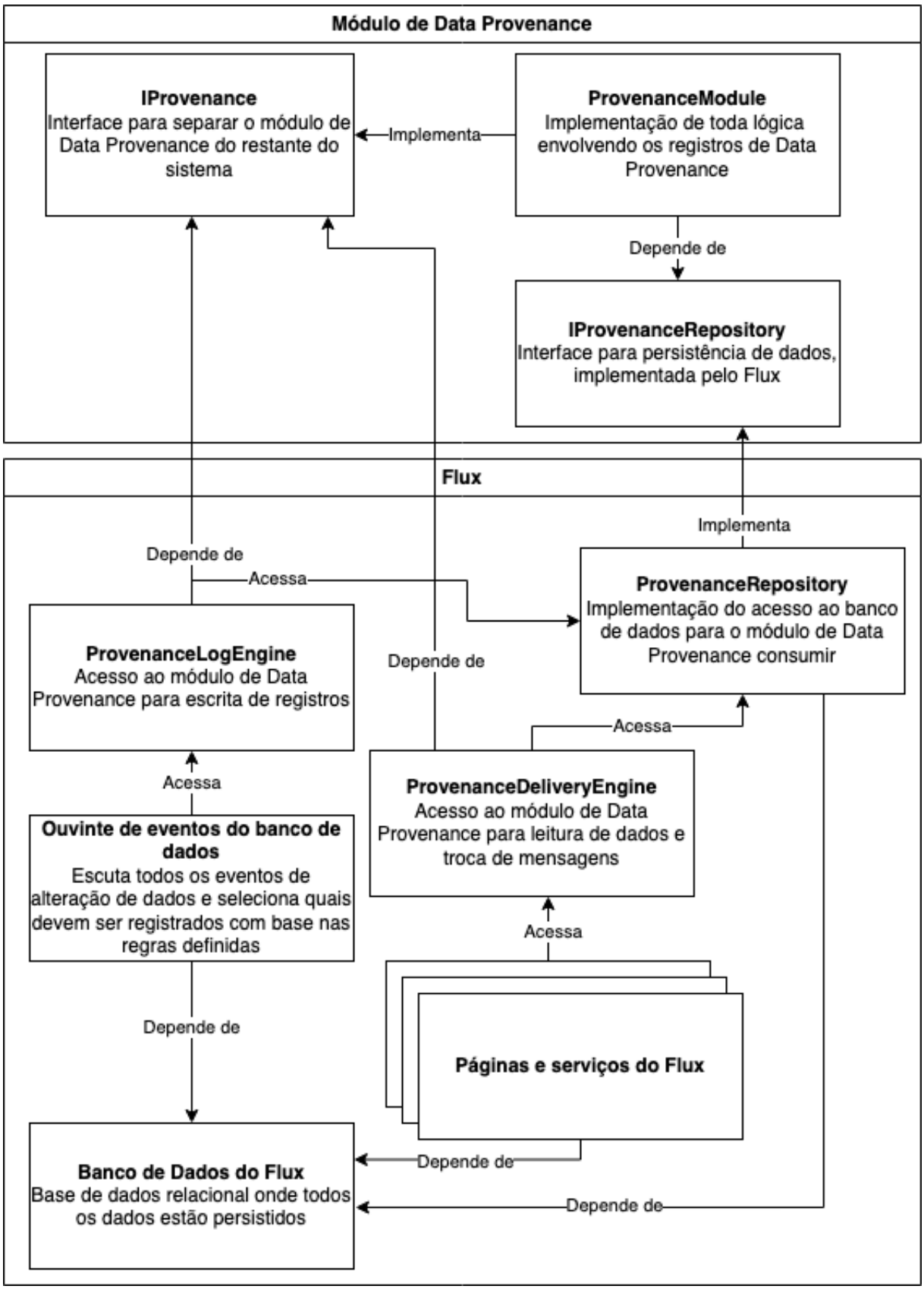
2.3.2 Padrões de arquitetura adotados

Como ilustrado na figura 2.4, podemos separar o sistema arquiteturalmente em duas partes, o módulo de *Data Provenance* e o sistema Flux, e estas se comunicam a partir da interface *IProvenance*, que por sua vez é implementada pelo próprio módulo de *Data Provenance*.

Para o módulo poder acessar o banco de dados relacional utilizado pelo Flux, foi utilizado o padrão *repository* (descrito em detalhes na referência [14]), onde operações de banco complexas, por exemplo, a inserção de um registro junto de todos os seus vértices e arestas, são abstraídas em uma única interface, neste caso chamada de *IProvenanceRepository*, para as funcionalidades necessárias serem facilmente acessadas pelo módulo.

Desta forma, construímos um sistema com suas fronteiras de responsabilidade cla-

Figura 2.4: Diagrama de dependências do sistema visando minimizar o acoplamento entre módulo de *Data Provenance* e os demais módulos do Flux.



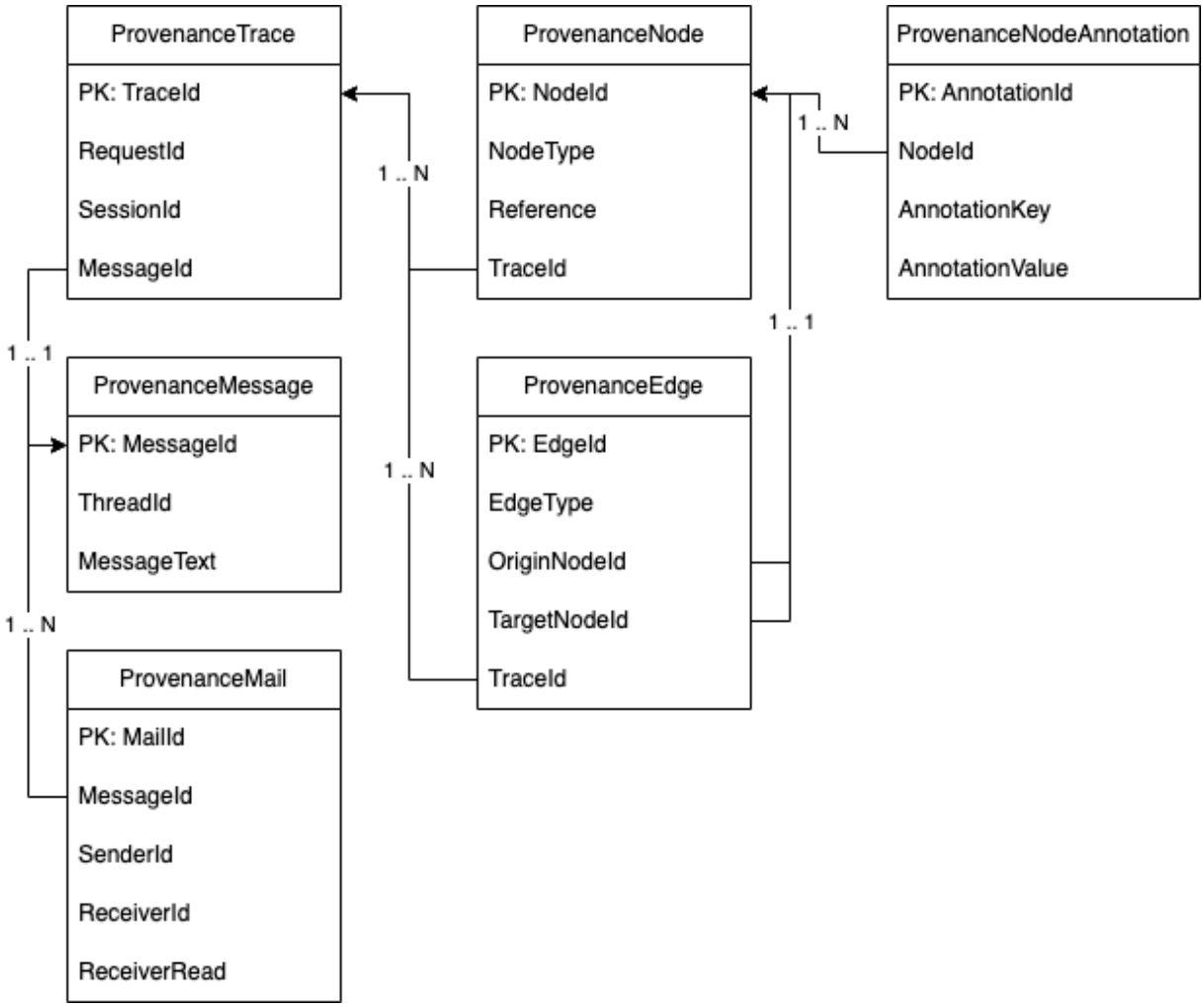
Fonte: Elaborado pelo autor.

ramente delimitadas, a lógica do Flux não interfere nas implementações da interface *IProvenance* e nem esta interfere nas implementações de *IProvenanceRepository* que regem o acesso ao banco de dados.

2.3.3 Estrutura de dados dos registros

Seguindo o padrão PROV [16], um registro de *Data Provenance* deve constituir um subgrafo, portanto, cada entidade do banco de dados envolvida em um registro foi modelada como um nó, representado pela tabela *ProvenanceNode*, e as arestas, representadas pela tabela *ProvenanceEdge*, indicam qual a relação entre estes nós, como exemplificado na figura 2.5.

Figura 2.5: Diagrama de classes indicando as relações entre as entidades do banco de dados do módulo de *Data Provenance*.



Fonte: Elaborado pelo autor.

Os campos *NodeType* e *EdgeType* representam a qual categoria de vértice um nó do grafo pertence, e qual tipo de alteração foi feita envolvendo estes elementos (ver seção 1.3.3), respectivamente, e seus possíveis valores estão descritos nas tabelas 2.1 e 2.2.

Tabela 2.1: Tipos de vértices representados em *NodeType*, referentes aos valores descritos em [16]

Valor	*Descrição*
Entity	Indica que um vértice do grafo representa uma entidade do sistema, referente a uma entrada específica de uma tabela do banco de dados.
Activity	Indica que um vértice do grafo representa uma função, rota *web*, ou serviço do sistema.
Agent	Indica que um vértice do grafo representa um usuário do sistema ou alguma rotina específica executada a partir de um gatilho pré-definido.

O campo *Reference* da tabela *ProvenanceNode* representa a referência à entidade pertencente ao sistema que utiliza o módulo de *Data Provenance*, no caso deste trabalho o sistema Flux. Para ser possível indicar corretamente qual a origem de um nó do grafo final, este campo é construído da seguinte forma:

- Caso $NodeType = Entity$,

  $$Reference = \text{Nome da tabela original} + \text{“ : ”} + \text{identificador da entidade}$$

  Por exemplo, no caso de uma entidade do tipo *Attribute* com identificador 14, teremos:

  $$Reference = \text{Attribute:14}$$

- Caso $NodeType = Activity$,

  $$Reference = \text{Identificador único de atividade dentro do sistema Flux}$$

  No caso do Flux optamos por usar a parte final da *URL* acessada como identificador da atividade, portanto, caso uma mudança seja realizada na página de editor de *workflows* por exemplo, teríamos:

  $$Reference = /workflowEditor$$

  No caso específico da tela de preenchimento das atividades de um *workflow* optamos por adicionar o identificador do *workflow* ao nome da atividade, já que grande parte do funcionamento do sistema e da coleta de dados ocorre nesta *URL*, por exemplo, ao preenchermos uma atividade do *workflow* com identificador 10 teríamos:

  $$Reference = /workflow\ [Workflow\ ID\ 10]$$

Tabela 2.2: Tipos de arestas representados em *EdgeType* referentes aos valores descritos em [20]

Valor	*Descrição*
WasAssociatedWith	Associação entre dois vértices.
WasGeneratedBy	Geração de um vértice, utilizado pelo Flux para indicar que uma entidade foi criada por algum usuário ou atividade.
WasAttributedTo	Atribuição de um vértice a uma entidade ou atividade.
WasDerivedFrom	Derivação, utilizado pelo Flux para indicar que um *workflow* foi gerado a partir de outro pela funcionalidade de edição de *workflows* em tempo real.
WasStartedBy	Indica que uma atividade foi iniciado por algum agente.
WasEndedBy	Indica que uma atividade foi finalizada por algum agente.
WasInformedBy	Indica que alguma atividade ou agente foram informados de algum valor, utilizado pelo Flux para representar que alguma entidade foi lida.
ActedOnBehalfOf	Indica qual foi o agente por trás de uma ação, utilizado pelo Flux para indicar qual foi o usuário responsável por uma alteração em uma entidade.
WasInfluencedBy	Representa que um vértice foi influenciado por outro, utilizado pelo Flux para representar que uma entidade sofreu uma atualização devido à influência de alguma atividade.
Used	Representa que um uma atividade ou agente usou alguma entidade.
WasRevisionOf	Indica que uma entidade é uma revisão de alguma outra, utilizado pelo Flux para indicar que houve uma atualização nos valores de uma entidade.
AttachedMessage	Utilizado quando um agente anexa uma mensagem a uma entidade sem realizar uma alteração na mesma, ver seção 2.5.3 para maiores detalhes.

- Caso $NodeType = Agent$,

  $Reference =$ "$Agent : UserId :$ " + Identificador do usuário + " : $Login :$ " + Login

  Por exemplo, para um usuário com *login admin* e identificador 57, teremos então:

  $Reference =$ Agent:UserId:57:Login:admin

A legibilidade dos dados foi considerada na construção do formato das referências, por isso o uso de *strings* fixas nos textos de atividades e agentes.

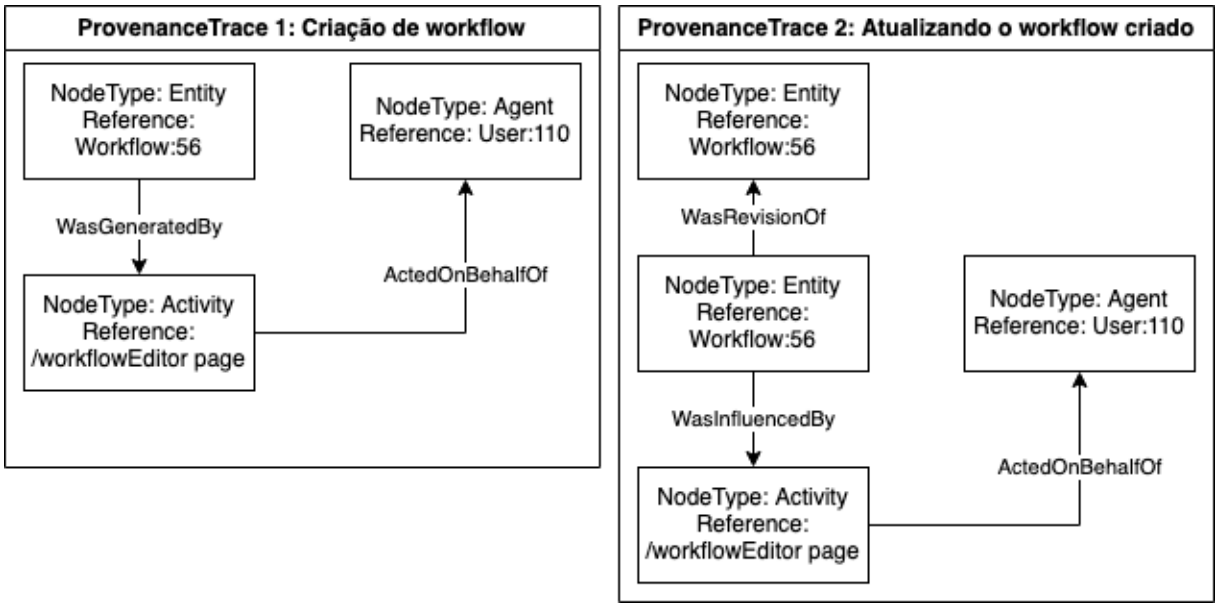
Desta forma conseguimos construir diversos nós que referenciam a mesma entidade do sistema original, podendo assim construir uma trilha de auditoria de todo o ciclo de vida da mesma, o que possibilita encontrar eficientemente todos os registros de *Data Provenance* associados. Para estas buscas podemos reconstruirmos a referência completa seguindo os padrões discutidos acima, ou buscar todos os registros relativos a uma parte

da referência, por exemplo, buscar todos os registros com *NodeType = Entity* e o texto *"Activity"* para encontrar todos os nós relacionados as atividades do Flux.

Visando ter uma rastreabilidade ainda maior dos registros, foi criada a tabela *ProvenanceTrace*, que integra todos os nós e arestas envolvidos em uma única operação, formando assim um subgrafo do grafo do ciclo de vida completo da entidade. A figura 2.6 exemplifica este conceito, onde temos um workflow sendo criado e depois atualizado, e todos os vértices e arestas que são registrados neste processo. Neste exemplo temos dois passos representados por dois *ProvenanceTrace* distintos:

- Em *ProvenanceTrace* 1, o *workflow* com identificador 56 é criado pelo usuário com identificador 110 na atividade referenciada pela página do Flux */workflowEditor*.
- Já no segundo momento em *ProvenanceTrace* 2, o mesmo workflow 56 é atualizado pelo mesmo usuário 110 na mesma página */workflowEditor*.

Figura 2.6: Exemplo de representação dos dados registrados pelo módulo de *Data Provenance*.



Fonte: Elaborado pelo autor.

Associada à tabela *ProvenanceNode*, temos a tabela *ProvenanceNodeAnnotation*, que persiste metadados referentes aos vértices que estão sendo registrados no formato chave-valor. Abaixo temos alguns dos metadados mais relevantes utilizados pelo Flux:

- Endereço IP e porta da requisição de origem.
- Dados de uma entidade no momento da inserção.
- Dados de uma entidade alterados durante uma atualização.

2.3.4 Interfaces de integração

Como dito na seção 2.3.2, todas as interações com o módulo desenvolvido passam pela interface pública *IProvenance*, seguindo o conceito de inversão de dependências (onde um sistema consome de uma interface e não de uma implementação específica, mais detalhes na referência [21]) e, seguindo este mesmo conceito, o sistema que utilizar o módulo deverá implementar a interface pública *IProvenanceRepository*.

Todos os requisitos definidos na seção 2.1 estão atendidos por esta interface, cujos principais métodos estão descritos na tabela 2.3.

Tabela 2.3: Principais métodos definidos na interface *IProvenance*

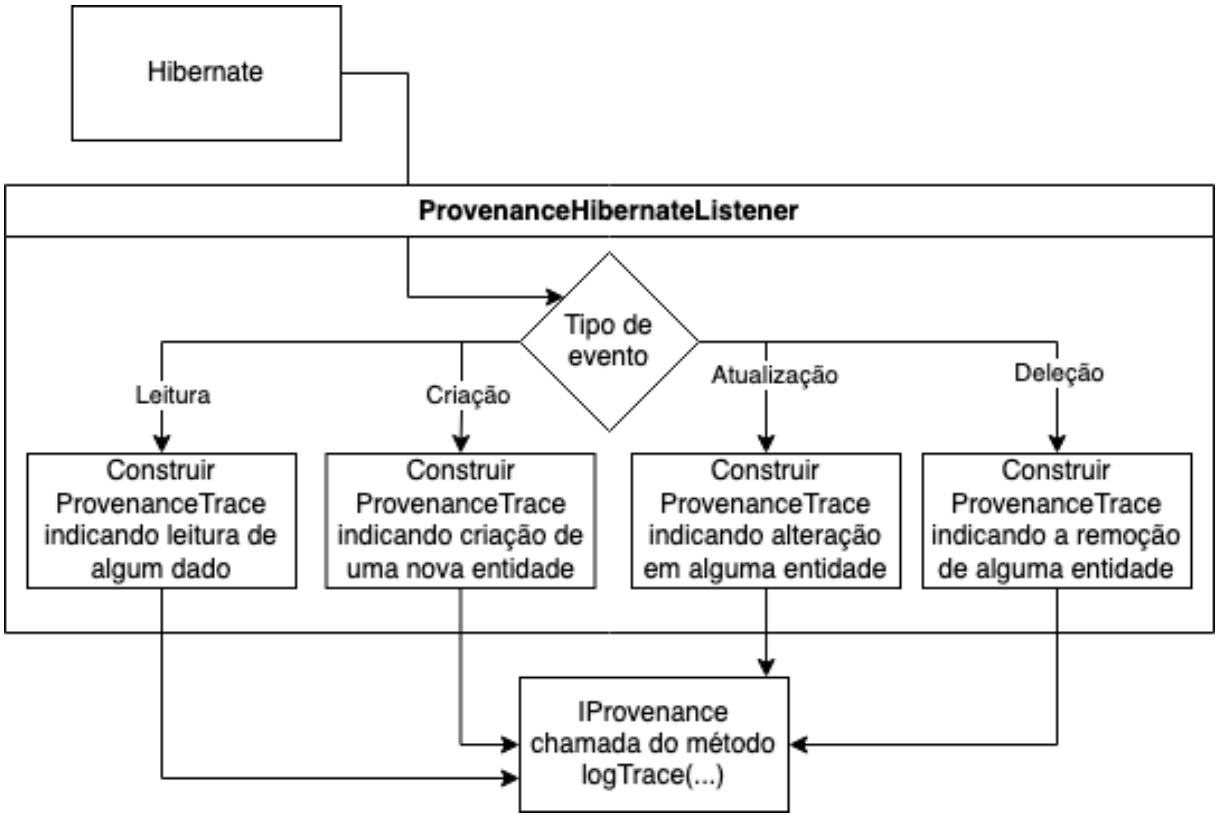
Método	*Descrição*
logTrace	Recebe um parâmetro do tipo *ProvenanceTrace* contendo todos os vértices e arestas internos e os persiste no banco de dados.
getTracesForReference	Obtém do banco de dados todos os subgrafos representados por objetos do tipo *ProvenanceTrace* que contém algum *ProvenanceNode* com o campo *Reference* igual à referência recebida como parâmetro.
saveMessageToTrace	Inicia uma conversa sobre o *ProvenanceTrace* identificado por *traceId* e marca os usuários identificados em *receivers* como os destinatários.
saveMessageToThread	Envia uma mensagem na conversa identificada por *threadId* e marca os usuários identificados em *receivers* como os destinatários.
getUserMessagesSent	Obtém todas as mensagens enviadas pelo usuário identificado por *userId*.
getUserMessagesReceived	Obtém todas as mensagens recebidas pelo usuário identificado por *userId*.
getThreadMessages	Obtém todas as mensagens na conversa identificada por *threadId*.

Os métodos para utilização da funcionalidade de troca de mensagens serão detalhados na seção 2.5.

2.4 Escrita de trilhas de *Data Provenance*

Para se conectar com o banco de dados, o Flux utiliza a biblioteca *Hibernate* [3], que implementa o padrão *observer* (um objeto dispara eventos e qualquer objeto que se interessar por estes eventos pode se inscrever para ser notificado quando eles ocorrerem [15]), e avisa aos observadores sempre que uma alteração for persistida no banco de dados, o que possibilitou rastrear todas as alterações que o Flux realiza de modo simples e unificado. A figura 2.7 ilustra este fluxo.

Figura 2.7: Arquitetura básica para receber e processar os eventos de alterações no banco de dados.



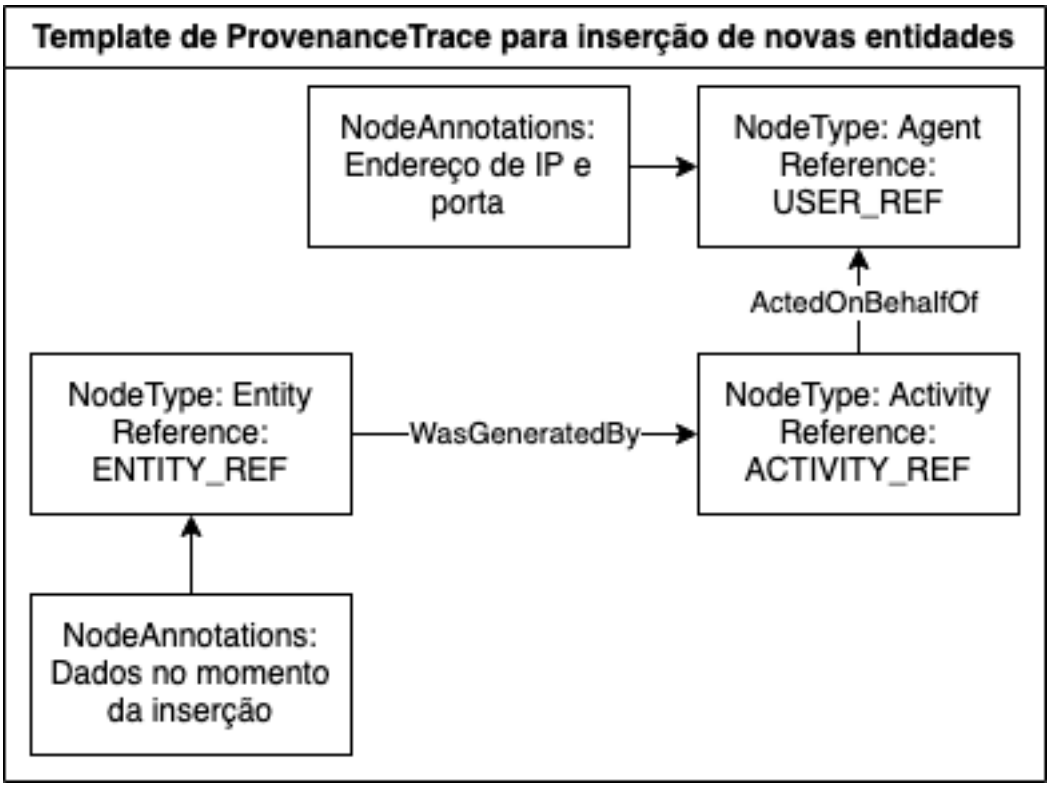
Fonte: Elaborado pelo autor.

Como podemos ver pela figura 2.7, os quatro tipos básicos de operações com dados estão contemplados nesta integração. Para cada um destes, um modelo de registros é enviado para o módulo, baseado na ideia de *templates* definida no estudo referenciado em [6]. Estes *templates* estão ilustrados nas figuras 2.8, 2.9, 2.10, 2.11, que usam as seguintes propriedades:

- *ENTITY_REF*: referência para a entidade que está sendo alterada.
- *USER_REF*: referência ao usuário que solicitou a alteração.

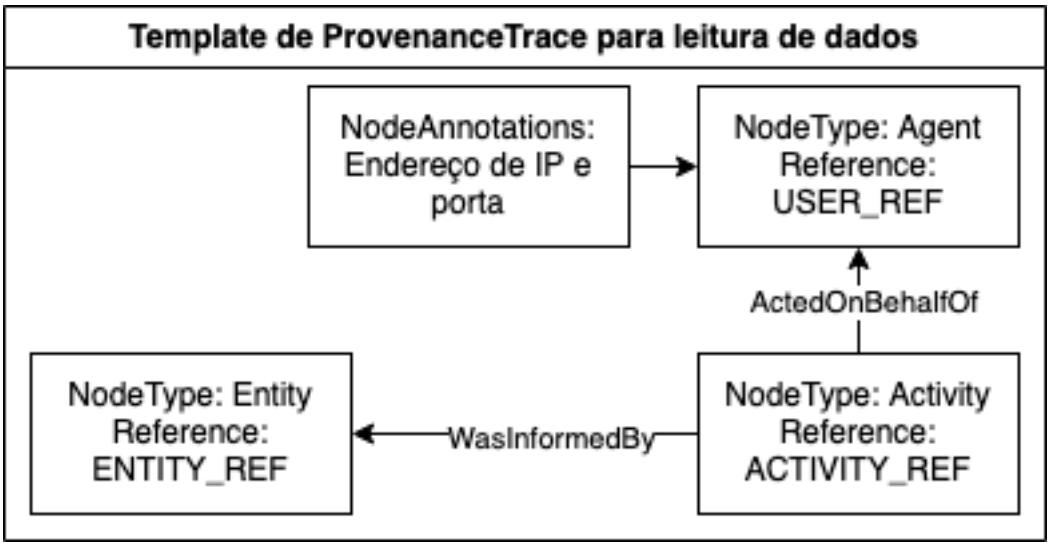
- *ACTIVITY_REF*: referência a atividade que está realizando a alteração.

Figura 2.8: *Template* para registro de inserções no banco de dados.



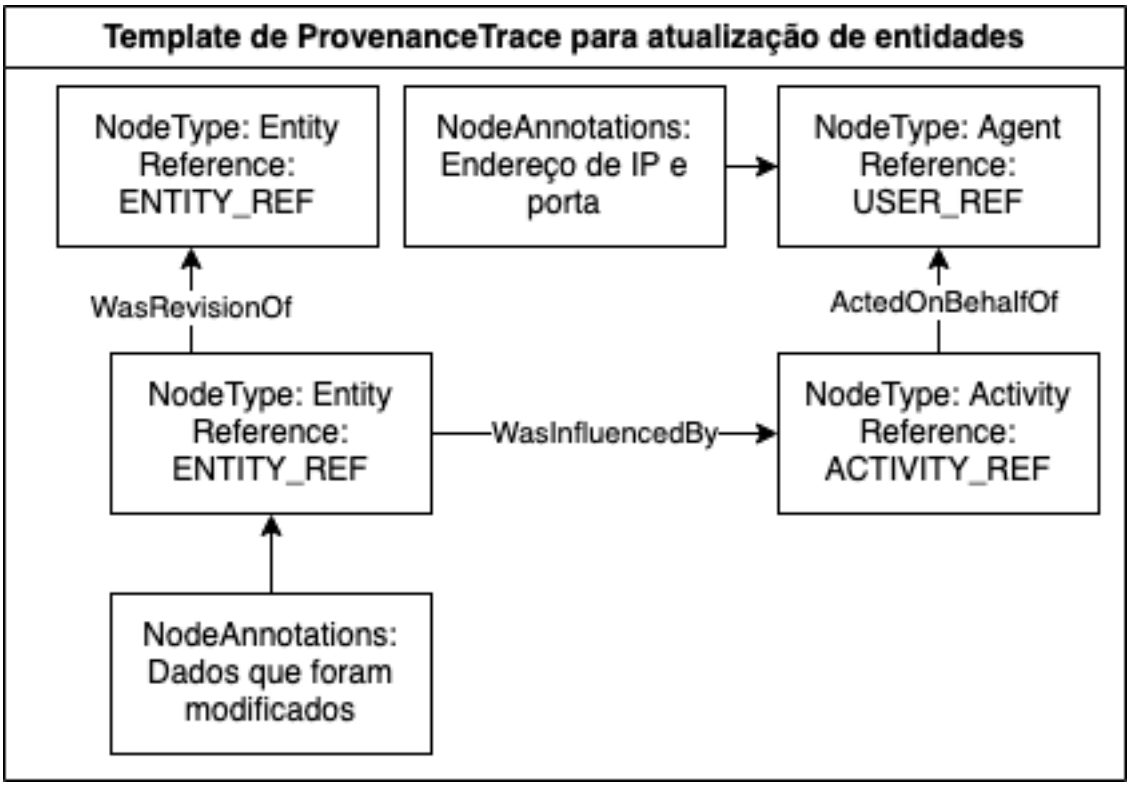
Fonte: Elaborado pelo autor.

Figura 2.9: *Template* para registro de leituras no banco de dados.



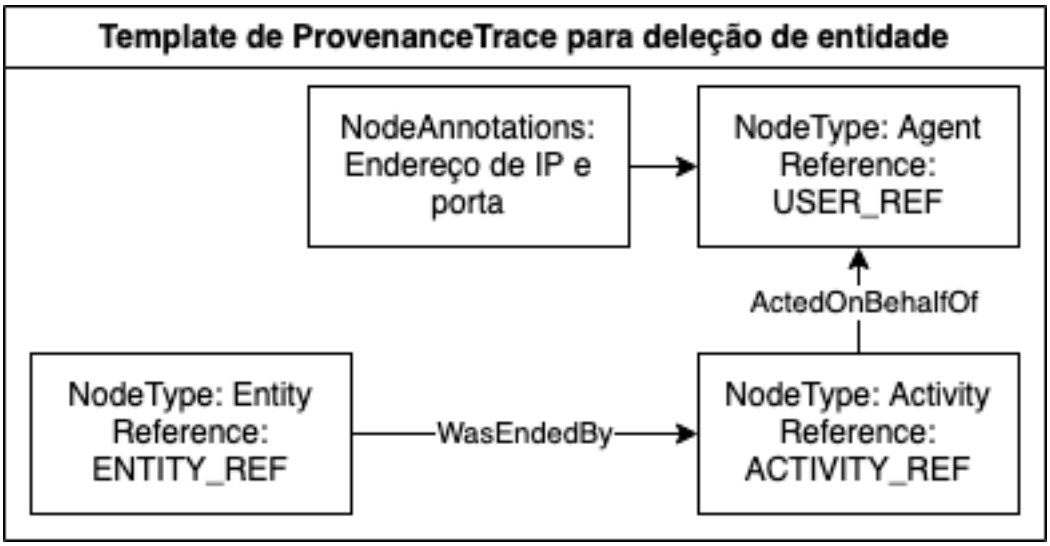
Fonte: Elaborado pelo autor.

Figura 2.10: *Template* para registro de atualizações no banco de dados.



Fonte: Elaborado pelo autor.

Figura 2.11: *Template* para registro de deleções no banco de dados.



Fonte: Elaborado pelo autor.

2.4.1 Gestão dos dados coletados

Como as trilhas de *Data Provenance* formatadas segundo os *templates* pré-estabelecidos podem gerar uma quantidade significativa de dados e necessitar de muito poder computacional para serem processadas, foi desenvolvida uma interface para que administradores do Flux possam controlar quais dados e operações devem ser rastreados.

Para cada uma das entidades disponíveis para auditoria, os administradores do sistema tem a opção de escolher quais das quatro operações básicas de dados (inserção, deleção, atualização, e leitura) devem ser consideradas.

2.5 Troca de mensagens

O módulo de *Data Provenance* também disponibiliza através da interface *IProvenance* (ver tabela 2.3) a funcionalidade de troca de mensagens a partir das trilhas capturadas, usuários podem iniciar uma conversa sobre uma mudança específica e marcar outros usuários para que eles sejam alertados sobre esta conversa.

2.5.1 Troca de mensagens

Como identificado no diagrama 2.5, as mensagens enviadas possuem o campo *ThreadId* que identifica a mensagem como pertencente a uma conversa, ou *thread*, que agrupa as mensagens cronologicamente e permite que os usuários discutam uma mudança realizada no sistema sem recorrer a sistemas externos como aplicativos de mensagens e *e-mails*.

Ao postar a primeira mensagem em um registro, o campo *ThreadId* é preenchido com uma chave única gerada aleatoriamente e todas as mensagens subsequentes enviadas nesta conversa compartilham deste mesmo identificador.

2.5.2 Registros como base para troca de mensagens

Para contextualizar a troca de mensagens e torná-la relevante ao sistema, foi decidido utilizar os subgrafos descritos nos objetos *ProvenanceTrace* como a base da troca de mensagens, desta forma uma mensagem, e consequentemente todas as mensagens enviadas nesta mesma conversa, está sempre associada a um registro de mudança no sistema, conforme exibido no diagrama 2.5.

2.5.3 Envio na tela de atividades

A troca de mensagens foi desenvolvida na tela de preenchimento de atividades de um *workflow*, desta forma a mensagem está naturalmente associada a uma atividade, facilitando a experiência de informar outros usuários sobre alguma dúvida ou decisão tomada e contextualizando todas as ações realizadas na mesma.

Foi criada uma aba onde o usuário tem a opção de redigir uma mensagem e marcar destinatários sem ter que sair do contexto do preenchimento da atividade, além de também exibir um resumo das outras conversas iniciadas referentes a atividade exibida.

Para o caso do usuário desejar enviar uma mensagem sem realizar uma alteração na atividade, foi criada uma funcionalidade onde o mesmo pode redigir uma mensagem e um registro de *Data Provenance* é criado utilizando o *EdgeType AttachedMessage* 2.2 indicando que o usuário relacionou esta mensagem com a atividade atualmente visualizada.

A interface visual para esta funcionalidade e as demais que envolvem a troca de mensagens podem ser encontradas na seção 3.1.2.

2.6 Edição de workflows em tempo real

Para que os *workflows* e atividades do Flux sejam consistentes entre as suas diversas instâncias e preenchimentos, a estrutura do *workflow* foi desenvolvida de forma que os mesmos atributos e atividades existam em todas as instâncias, trazendo confiabilidade nos campos disponibilizados para todas as instâncias, porém, em alguns casos de uso (por exemplo, quando os passos de um experimentos continuam a ser alterados enquanto está

sendo realizado, ou em que um resultado indica que passos adicionais devem ser executados e registrados no sistema), foi identificada a necessidade de alterar esta estrutura durante o preenchimento das atividades, de forma que essa alteração seja permanente ou não.

Para que este requisito fosse atingido, foi desenvolvida funcionalidade de edição de *workflows* em tempo real, onde usuários com permissão podem instalar um novo *workflow* com as mudanças realizadas e copiar as instâncias já existentes para a cópia, adicionando ou removendo atributos e atividades sempre que necessário. Também é possível apagar automaticamente o *workflow* original e as suas instâncias para a cópia sobrescrever o original, e não haja uma bifurcação no modo de preencher um *workflow*.

É importante ressaltar que o caso de uso de apagar o *workflow* original deve ser usado especificamente para casos em que o experimento continua sendo pensado no momento da execução do mesmo, e que essa funcionalidade requer uma permissão específica. Nos outros casos, a pessoa encarregada do laboratório seria a única com permissões para remover os *workflows* existentes e deve fazê-lo apenas quando não forem mais necessários.

Para executar essas ações, os usuários podem tanto usar o editor de *workflows* ou realizar uma edição rápida diretamente da tela de preenchimento de atividades, onde é permitido que adicionem um novo atributo na estrutura de uma atividade.

As interfaces visuais para esta funcionalidade podem ser encontradas na seção 3.1.4 e a sua sinergia com o módulo de *Data Provenance* será aprofundada na seção 3.2.4.

2.6.1 Integração com o módulo de Data Provenance

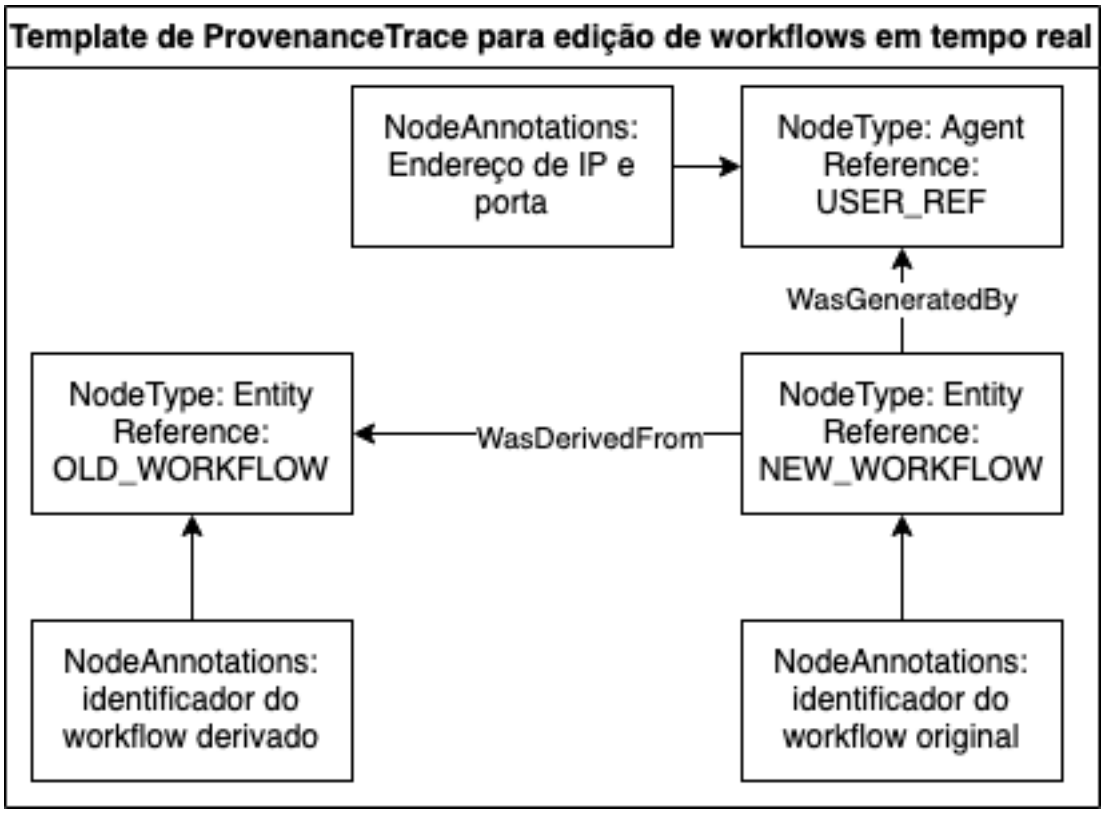
Como a cópia de um *workflow* com as suas propriedades alteradas é uma operação significativa, foi desenvolvido um *template* de *ProvenanceTrace* (ver seção 2.4) específico para ser registrado quando esta operação é realizada.

A figura 2.12 ilustra este *template* e recorre às seguintes propriedades:

- *USER_REF*: referência ao usuário que solicitou a alteração.
- *NEW_WORKFLOW*: referência ao novo *workflow* que foi gerado e editado no processo.
- *OLD_WORKFLOW*: referência ao *workflow* que foi copiado.

O Flux então passa a registrar um objeto com este *template* sempre que a operação de edição de *workflows* em tempo real é concluída.

Figura 2.12: *Template* para registro de edição de *workflows* em tempo real.



Fonte: Elaborado pelo autor.

2.6.2 Gestão de acesso

Para que esta operação não seja disponibilizada para todos os usuários, foi desenvolvida uma gestão de acessos específica, onde usuários administradores podem habilitar ou desabilitar outros usuários a executá-la. Desta forma, usuários que não possuem permissão nessas partes do sistema não conseguem ver as telas e nem os botões para interagir com estas ações.

A interface da gestão de acesso foi desenvolvida na tela de permissões já existente no Flux para os administradores terem uma visão geral e unificada das permissões de um dado usuário.

Capítulo 3

Resultados

Neste capítulo apresentaremos os resultados, tanto em relação às interfaces visuais desenvolvidas quanto às aplicações dos conceitos construídos ao longo deste trabalho.

Para leitores que se interessarem nos códigos desenvolvidos, estes se encontram nos apêndices A, B, C, e D.

3.1 Interfaces visuais desenvolvidas

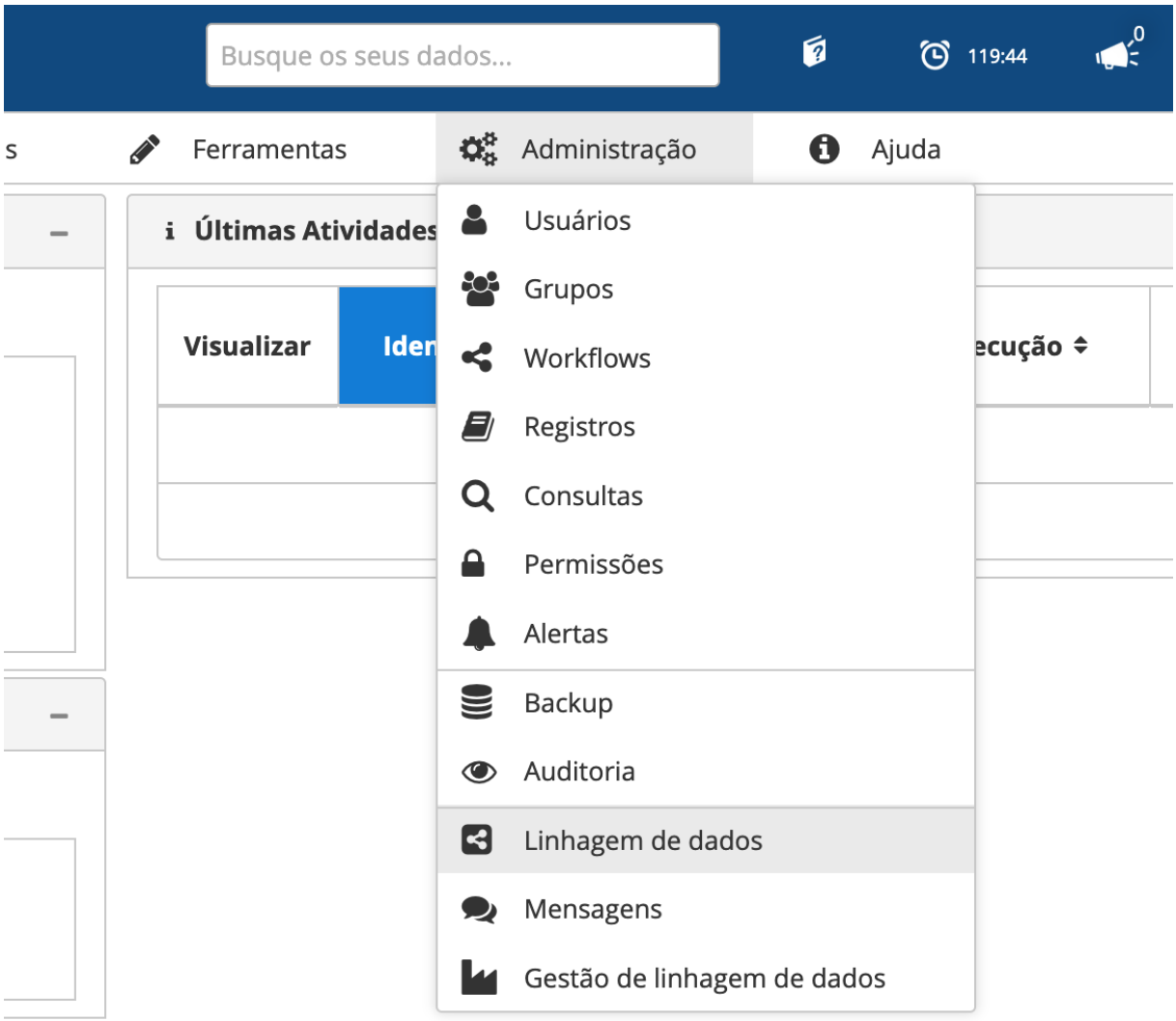
Nesta seção discutiremos as interfaces visuais gráficas desenvolvidas para atingir os objetivos do trabalho. Não discutiremos interfaces do sistema Flux que não foram criadas no contexto deste trabalho, exceto em algumas poucas exceções, já que estas estão detalhadas nos trabalhos referenciados em [9], [8], e [22].

3.1.1 Gerenciamento e permissões

Para acesso ao módulo de *Data Provenance* e aos sistemas de aprovisionamento, foram incluídos três novos itens nos menus de Administração do Flux, como mostrados na figura 3.1, são estes:

- Linhagem de dados, para acessar as visualizações de dados criadas para o módulo de *Data Provenance* (ver seção 3.1.3).
- Mensagens, para acessar a interface de troca de mensagens (ver seção 3.1.2).
- Gestão de linhagem de dados, para gerir quais entidades serão rastreadas.

Figura 3.1: Novos itens adicionados ao menu de Administração do Flux.

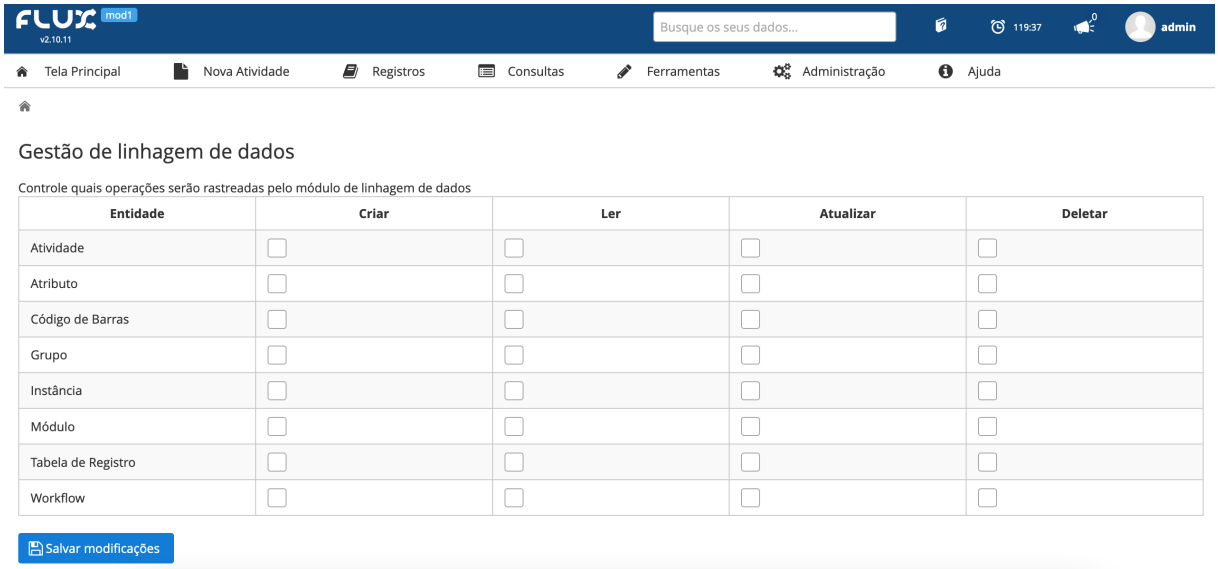


Fonte: Elaborado pelo autor.

Ao clicar no item "Gestão de linhagem de dados", o usuário é direcionado para uma tela onde pode escolher quais entidades e operações deseja que sejam rastreadas. O módulo possui a capacidade de rastrear as operações básicas de dados (criação, deleção, atualização e leitura) e cada uma delas pode ou não ser habilitada para cada tabela separadamente, como mostrado nas figuras 3.2 e 3.3. Apenas usuários administradores tem acesso a esta funcionalidade.

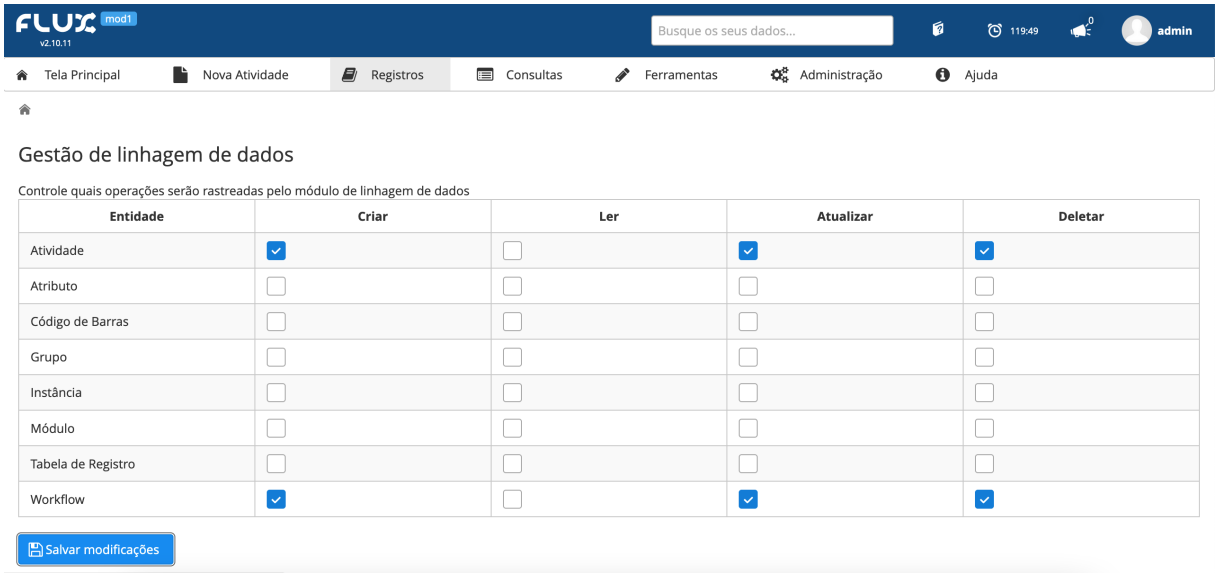
Para podermos controlar as permissões relacionadas à edição de *workflows* em tempo real (ver seção 2.6), foi desenvolvido uma adição à tela de "Permissões"que pode ser acessada pelo item de mesmo nome do menu de "Administração". Nesta tela, mostrada na figura 3.4, temos agora na parte inferior um local onde os usuários administradores podem selecionar algum outro usuário e indicar se eles podem alterar a estrutura de um

Figura 3.2: Tela de gestão de linhagem de dados sem nenhuma entidade preenchida.



Fonte: Elaborado pelo autor.

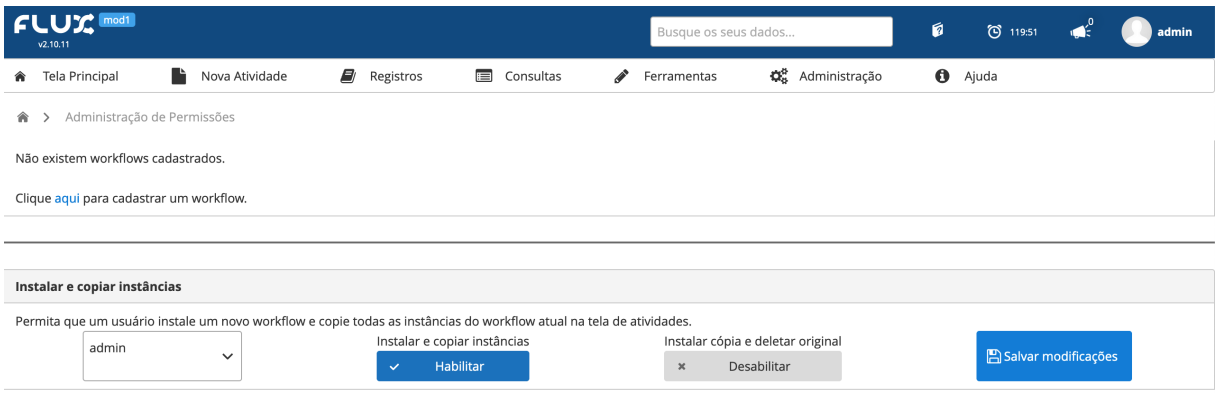
Figura 3.3: Tela de gestão de linhagem de dados com algumas entidades preenchidas.



Fonte: Elaborado pelo autor.

workflow existente fazendo uma cópia do mesmo (opção "Instalar e copiar instâncias") ou sobrescrevendo este (opção "Instalar cópia e deletar original").

Figura 3.4: Tela de permissões com as opções para habilitar a edição de *workflows* em tempo real.



Fonte: Elaborado pelo autor.

3.1.2 Troca de mensagens

Como o foco da funcionalidade de troca de mensagens no sistema Flux é a possibilidade dos usuários se comunicarem no momento de preenchimento de atividades de instâncias de *workflows*, foi desenvolvido uma interface especial para esta tela.

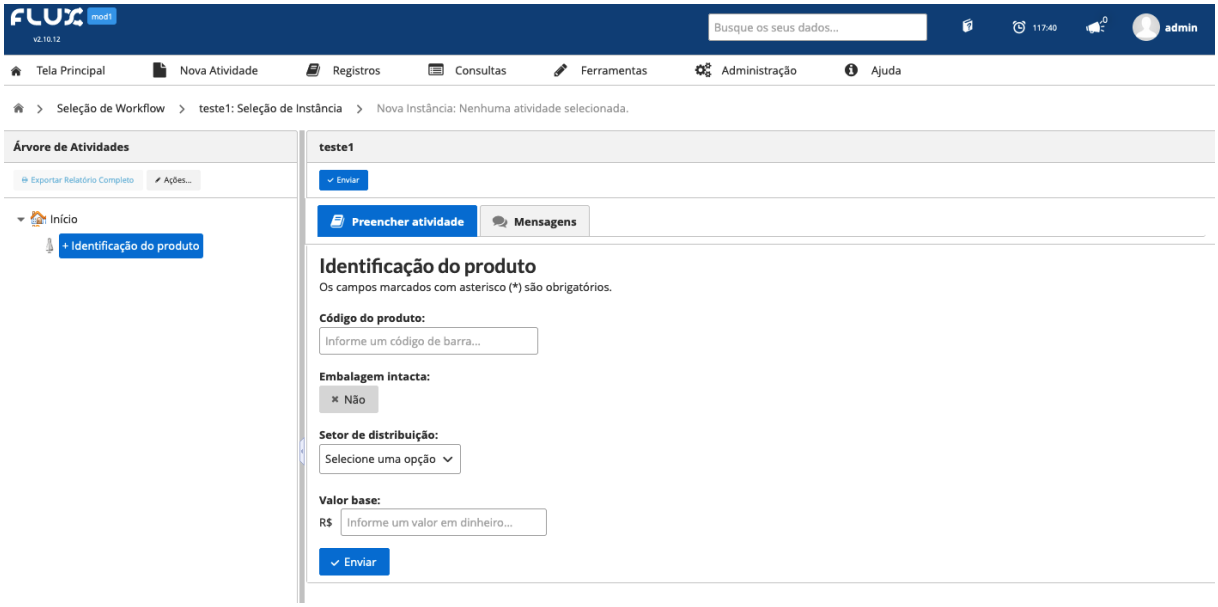
Como exibido nas figuras 3.5 e 3.6, foi adicionada uma aba onde o usuário pode preencher uma mensagem que será anexada à alteração salva na atividade.

Caso o usuário esteja trabalhando fora de uma instância de um *workflow*, ao preencher a mensagem e seus destinatários (ver figura 3.7) e clicar no botão "Enviar mensagem", o usuário receberá um aviso de que a mensagem será enviada ao salvar alguma alteração (figura 3.8), podendo então sair desta aba e ir realizar o trabalho necessário nesta tela. Caso contrário, a mensagem será anexada a um registro de *Data Provenance* indicando o usuário como agente e a atividade aberta como entidade, unidos por uma aresta do tipo *AttachedMessage* (ver seção 2.5.3).

Depois que a mensagem for enviada, ela pode ser acessada pela mesma aba e usuários podem responder e iniciar uma conversa com base na mensagem original, como mostrado na figura 3.9.

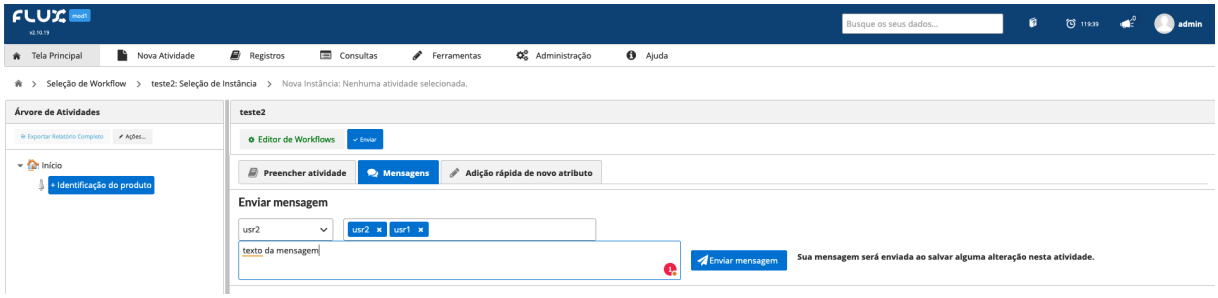
Quando um usuário é adicionado aos destinatários de uma mensagem, um aviso vermelho é exibido no seu menu de "Administração", indicando que ele deve ir para a

Figura 3.5: Visão geral da tela de preenchimento de atividade de *workflow*.



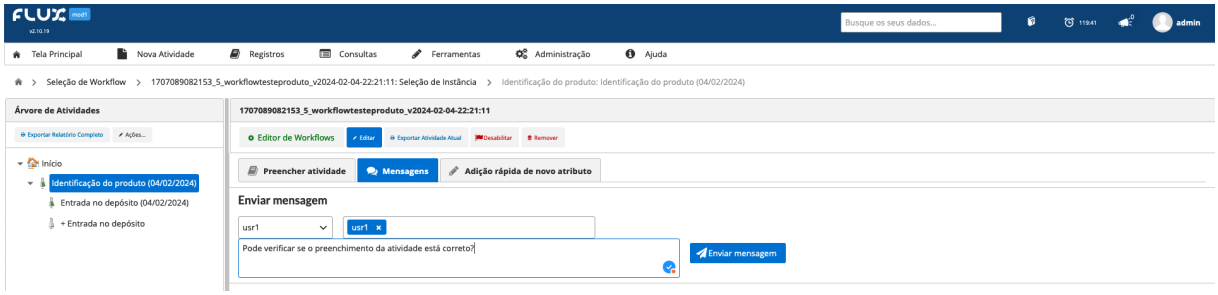
Fonte: Elaborado pelo autor.

Figura 3.6: Aba de mensagens da tela de preenchimento de atividade de *workflow*.



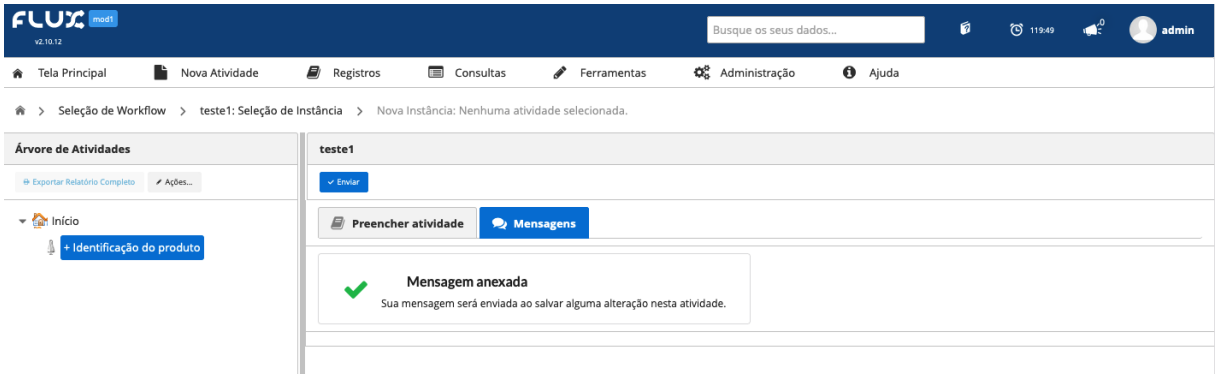
Fonte: Elaborado pelo autor.

Figura 3.7: Aba de mensagens da tela de preenchimento de atividade de *workflow* com o texto e destinatários preenchidos.



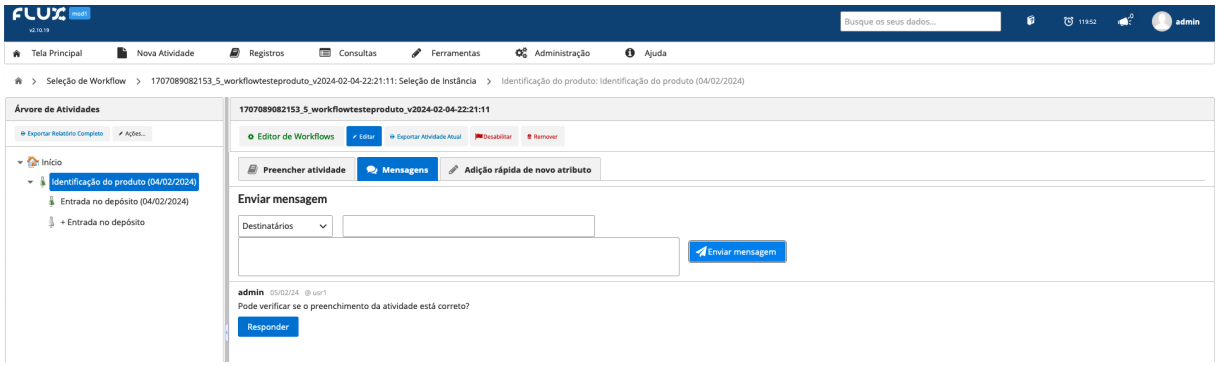
Fonte: Elaborado pelo autor.

Figura 3.8: Aba de mensagens da tela de preenchimento de atividade de *workflow* com uma mensagem anexada.



Fonte: Elaborado pelo autor.

Figura 3.9: Aba de mensagens da tela de preenchimento de atividade de *workflow* com uma mensagem enviada.



Fonte: Elaborado pelo autor.

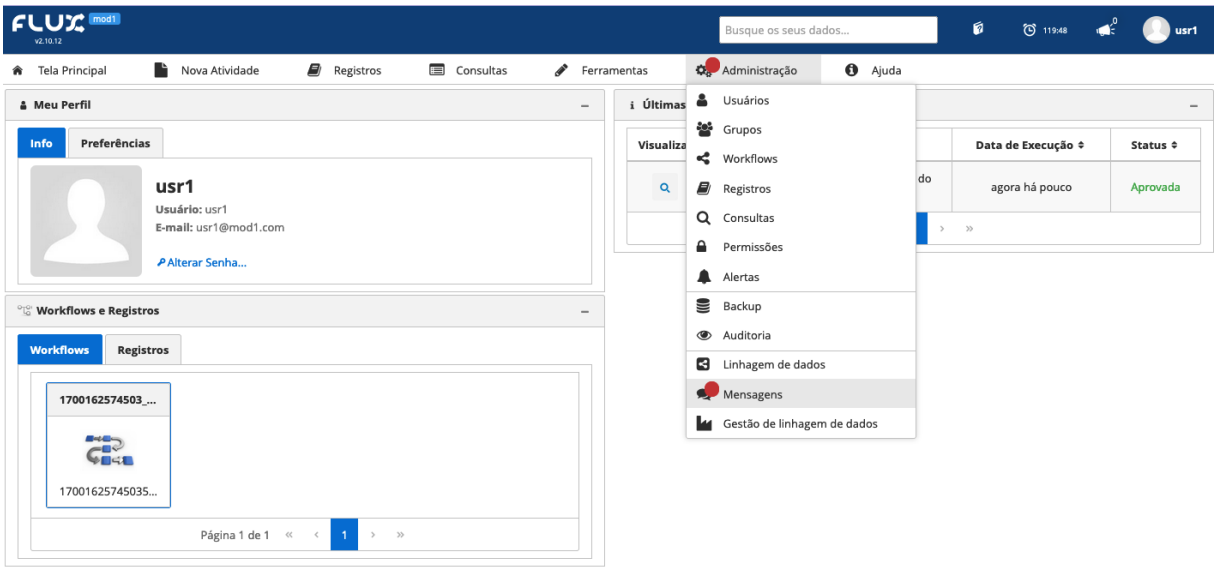
tela de mensagens, como mostrado na figura 3.10. Esta tela (ver figura 3.11) contém duas abas, uma caixa de entrada, contendo todas as mensagens que o usuário foi marcado e indicando quais ainda não foram lidas, e uma aba de saída contendo todas as mensagens que usuário enviou. Ao clicar em "Ir para a atividade", o usuário é direcionado para a atividade que contém a mensagem e ela é marcada como lida.

3.1.3 Visualizações de dados

Para os usuários poderem extrair informações foram desenvolvidas visualizações das trilhas de *Data Provenance* que podem ser acessadas pelo botão Linhagem de dados do menu de administração (ver seção 3.1.1).

Para usuários não-técnicos do sistema, foi desenvolvida uma visualização simplificada dos registros contendo apenas uma linha do tempo dos acontecimentos, como mos-

Figura 3.10: Aviso de mensagem recebida no menu de "Administração"do Flux.



Fonte: Elaborado pelo autor.

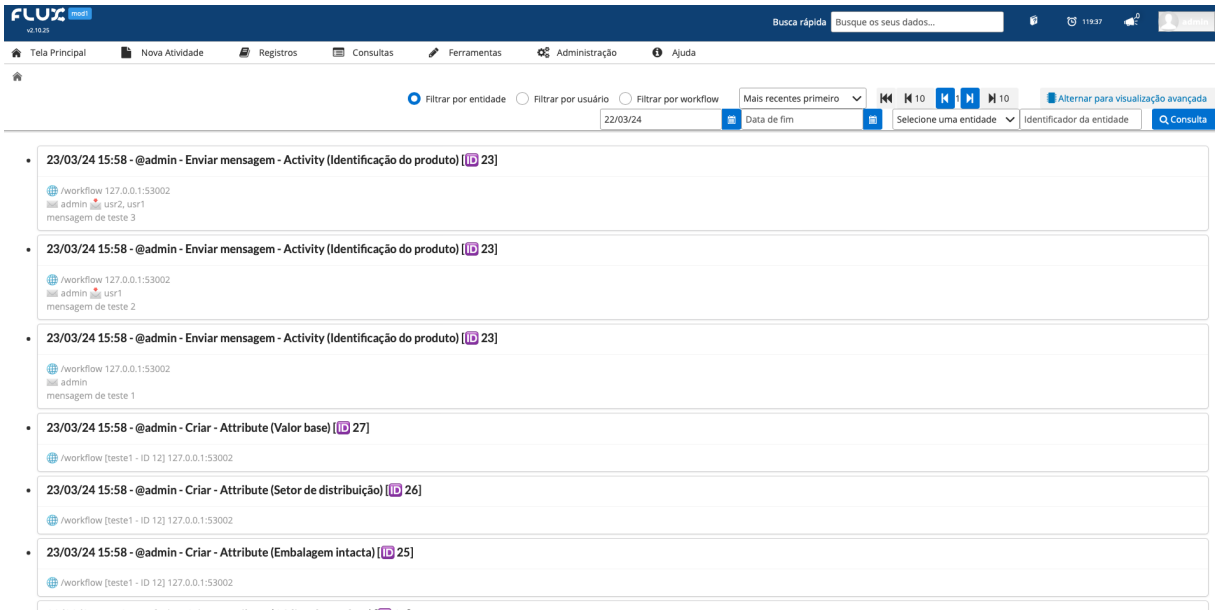
Figura 3.11: Tela de mensagens do Flux mostrando que o usuário tem uma mensagem não lida.

Fonte: Elaborado pelo autor.

trado na figura 3.12. Nesta figura também podemos ver os filtros disponíveis no canto superior esquerdo, onde o usuário tem as seguintes opções:

- Filtrar por datas: o usuário pode escolher uma data de início e uma de fim para a sua pesquisa;
- Filtrar por tipo de entidade: filtrar por registros de um tipo de entidade específica ou com um identificador específico;
- Filtrar por usuário: buscar todos os registros de alterações realizadas por um usuário específico;
- Filtrar por *workflow*: pesquisar registros de um *workflow*, ou de uma de suas instâncias, ou de uma atividade da instância selecionada, ou de um atributo da atividade selecionada.

Figura 3.12: Visualização simplificada em formato de linha do tempo dos registros de *Data Provenance*.



Fonte: Elaborado pelo autor.

Usuários que possuem um conhecimento maior sobre *Data Provenance* ou que querem analisar mais profundamente o comportamento do sistema podem usar o botão "Alternar para visualização avançada" no canto superior esquerdo (ver figura 3.12), e serão direcionados para uma tela com mais opções de visualizações, sendo elas:

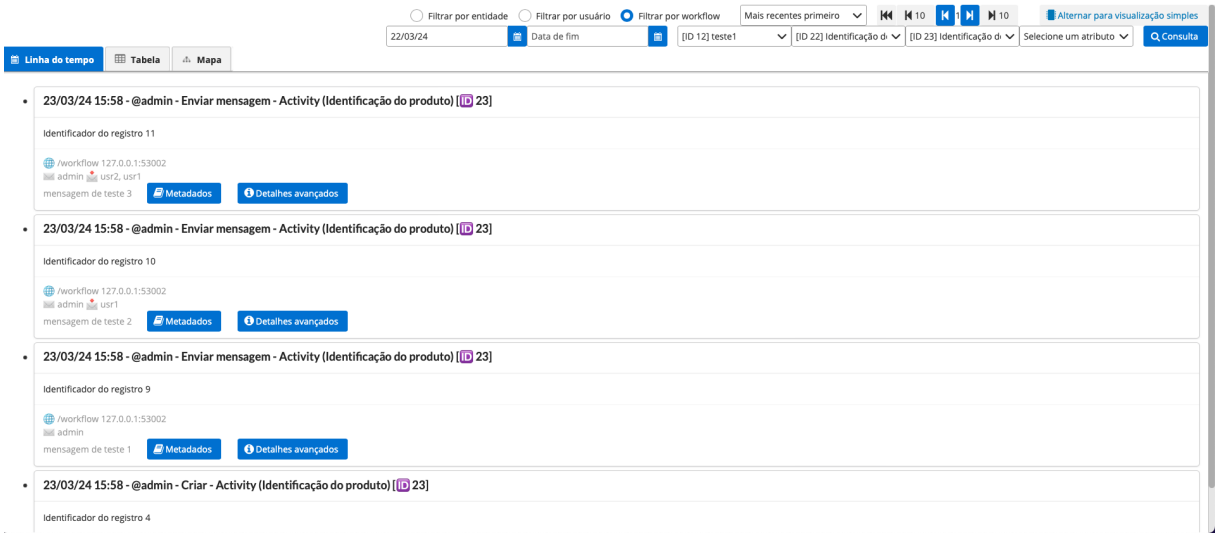
- Linha do tempo: visualização similar à simplificada, porém, com mais detalhes, exemplificada na figura 3.13;
- Tabela: os registros são listados em formato tabular com todas as informações obtidas nas suas colunas, incluindo os metadados, exemplificada na figura 3.15;
- Mapa: o grafo dos registros é renderizado para o usuário compreender visualmente as relações entre as entidades, agentes, e atividades, exemplificada na figura 3.16.

O objetivo da visualização em grafo é trazer uma experiência similar à proposta inicialmente na seção 1.3.3 e exemplificada na figura 1.7. Ao clicar em um dos vértices do grafo, o usuário recebe um *pop-up* informando todos os registros relacionados com aquela determinada entidade, atividade, ou agente, como mostrado na figura 3.17.

Para retornar à visualização simplificada, basta que o usuário clique no botão "Alternar para visualização simplificada" no canto superior esquerdo da tela.

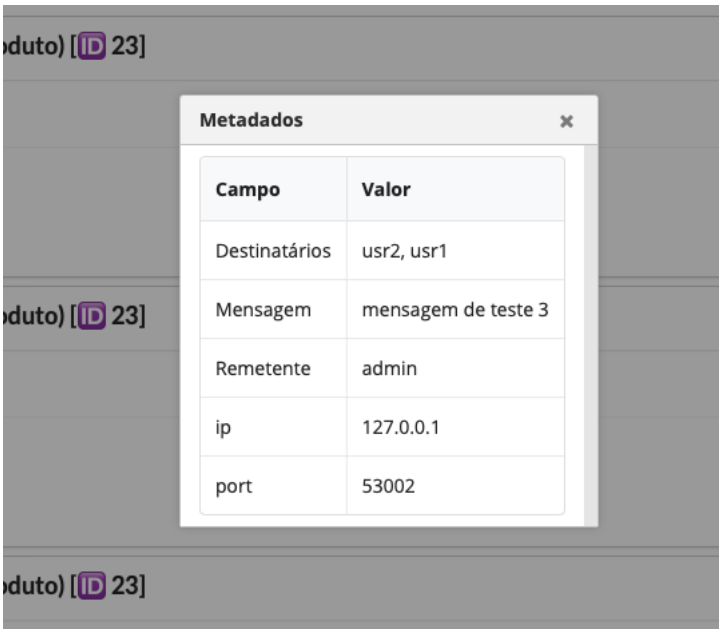
Na visualização avançada em formato de linha do tempo e de tabela (figuras 3.13 e 3.15, respectivamente), é exibido um botão de detalhes em cada um dos registros. Este botão leva o usuário para a tela de informações adicionais, composta por quatro abas para seleção de escopo, cada uma contendo as seguintes possibilidades de visualização de dados:

Figura 3.13: Visualização avançada em formato de linha do tempo detalhada.



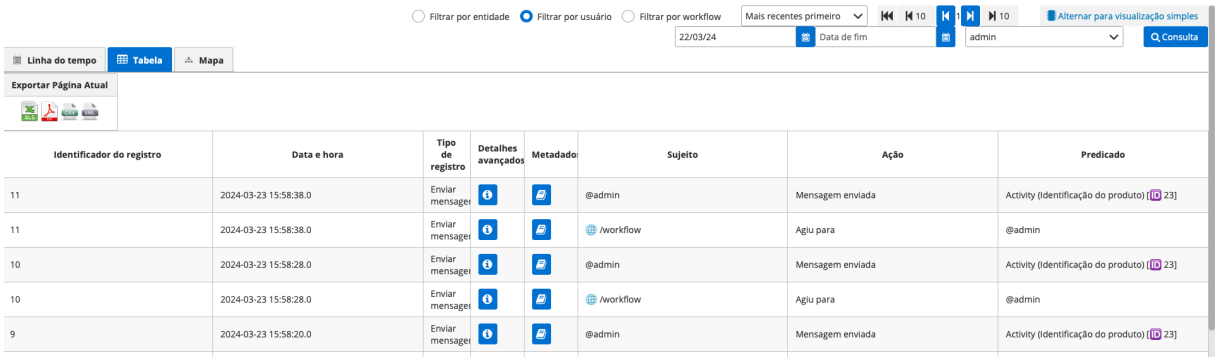
Fonte: Elaborado pelo autor.

Figura 3.14: Exibição dos metadados de um registro.



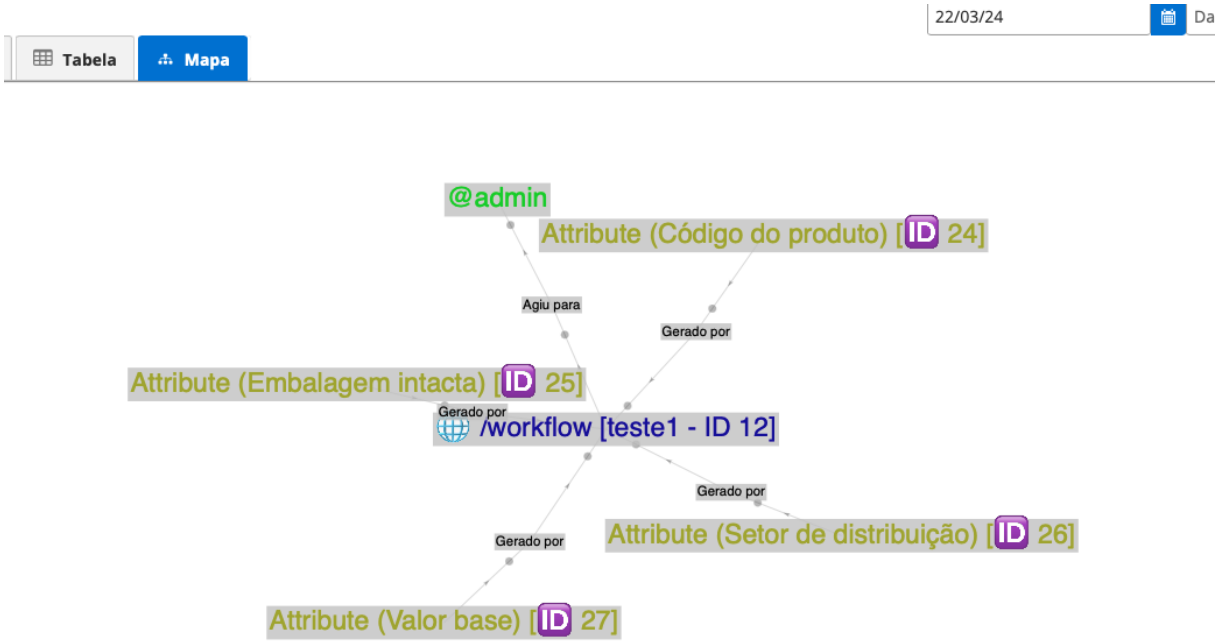
Fonte: Elaborado pelo autor.

Figura 3.15: Visualização avançada em formato tabular.



Fonte: Elaborado pelo autor.

Figura 3.16: Visualização avançada em formato de grafo.



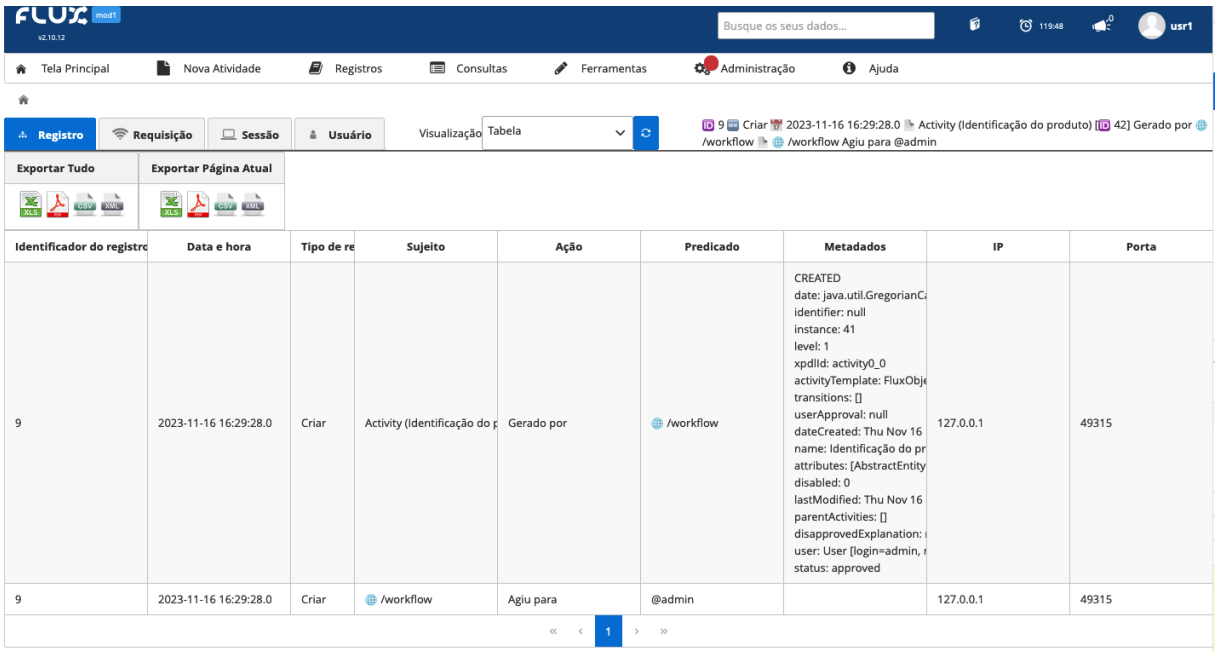
Fonte: Elaborado pelo autor.

Figura 3.17: Dados exibidos ao clicar em um vértice do grafo.

Fonte: Elaborado pelo autor.

- Tabela, a mesma visualização tabular vista na tela de "Linhagem de dados" (figura 3.15).
- Linha do tempo detalhada, visualização avançada da linha do tempo, mostrada na figura 3.12.
- Mapa, a mesma visualização em grafo vista na tela de "Linhagem de dados" (figura 3.16).

Figura 3.18: Detalhes do registro visualizados em formato de tabela com escopo de registro selecionado.

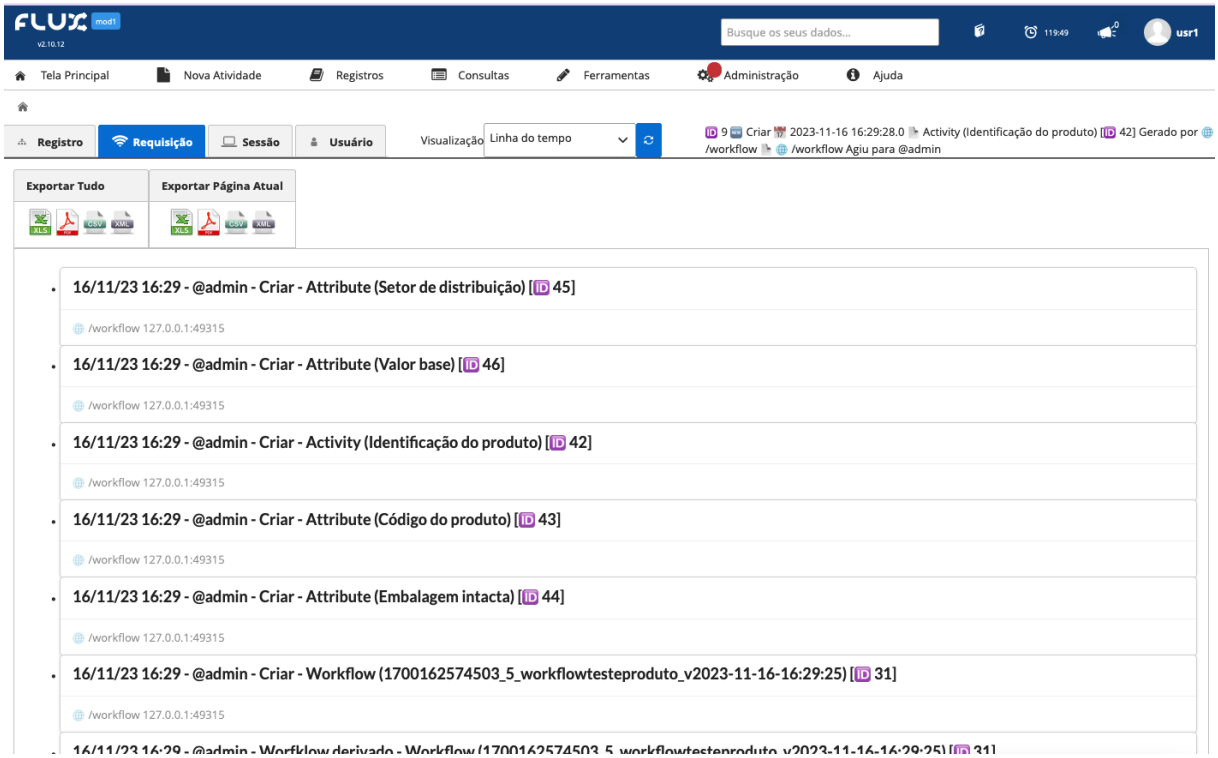


Fonte: Elaborado pelo autor.

A figura 3.18 nos mostra a tela de detalhes de registro. No seu canto superior direito podemos ver um resumo do registro original que está sendo estudado, no lado esquerdo deste resumo, temos um menu *dropdown* para alterar a visualização de dados que está sendo aplicada (ver figura 3.20), e no canto superior esquerdo temos as abas de seleção de escopo, onde podemos ver registros relacionados ao original com base nos seguintes escopos:

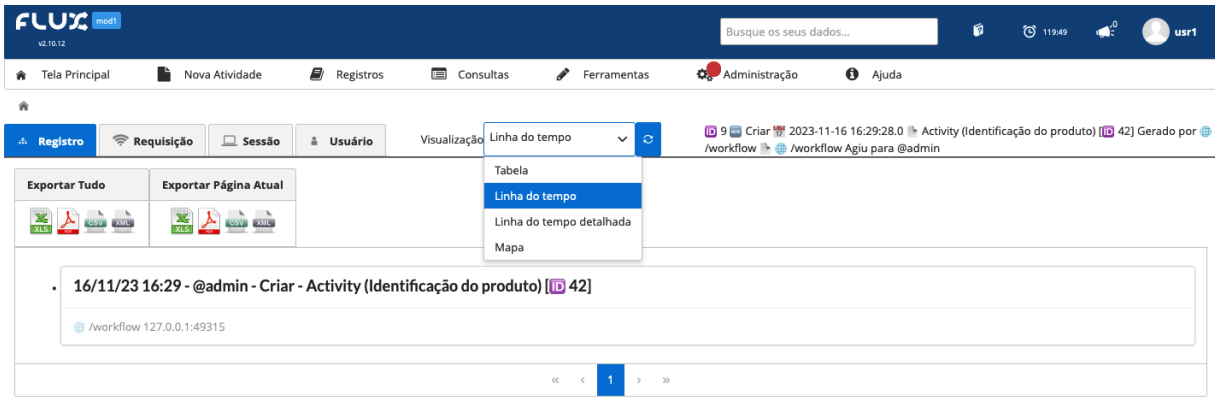
- Registro, apenas o registro original.
- Requisição, registros gravados durante a mesma requisição que o original.
- Sessão, registros gravados durante a mesma sessão de usuário que o original.
- Usuário, registros de operações realizadas pelo mesmo usuário que o original.

Figura 3.19: Detalhes do registro visualizados em formato de linha do tempo com escopo de requisição selecionado.



Fonte: Elaborado pelo autor.

Figura 3.20: Seleção de visualização para os detalhes do registro.

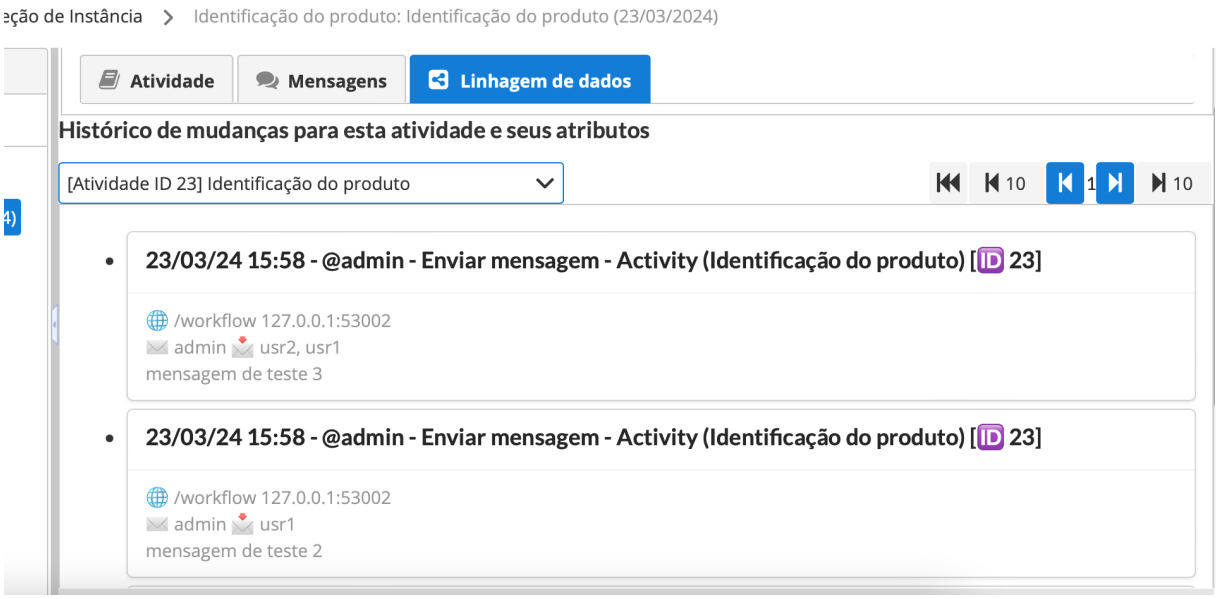


Fonte: Elaborado pelo autor.

O objetivo da seleção de escopos é dar mais formas para compreendermos como os dados estão sendo alterados e avaliarmos a operação do sistema.

Para facilitar a análise de históricos em tempo real, foi adicionada uma aba na tela de preenchimento de atividades com um resumo dos dados da atividade selecionada ou de seus atributos, conforme o usuário selecionar, como mostrado na figura 3.21.

Figura 3.21: Visualização de dados históricos na página de preenchimento de uma atividade.



Fonte: Elaborado pelo autor.

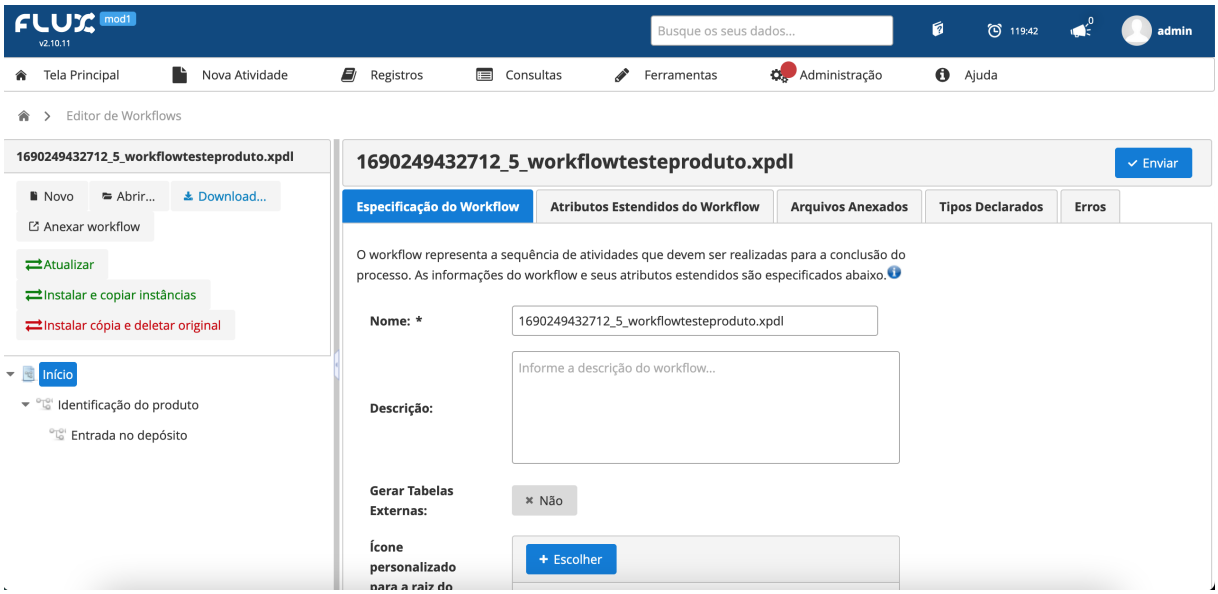
3.1.4 Edição de workflows em tempo real

Foi adicionado na tela de preenchimento de atividades, para usuários que possuem alguma habilitação para a funcionalidade de edição de *workflows* em tempo real (ver seções 2.6 e 3.1.1), um botão (ver figura 3.23) chamado "Editor de *Workflows*".

Ao clicar neste botão, o usuário é direcionado para o editor de *workflows* (figura 3.22) com opções a mais relativas às permissões que lhe foram dadas. Com isso é possível fazer grandes alterações na estrutura de um *workflow* em tempo real e ter essas alterações aplicadas a todo o conjunto de instâncias deste ou a cópias das mesmas.

Foi também desenvolvido um fluxo simplificado para adições rápidas de novos campos em *workflows*, que só pode ser utilizado e visualizado por usuários com a permissão "Instalar cópia e deletar original" (ver seção 3.1.1), que consiste em uma aba adicional na tela de preenchimento de atividades, ao lado direito da aba de mensagens (ver seção

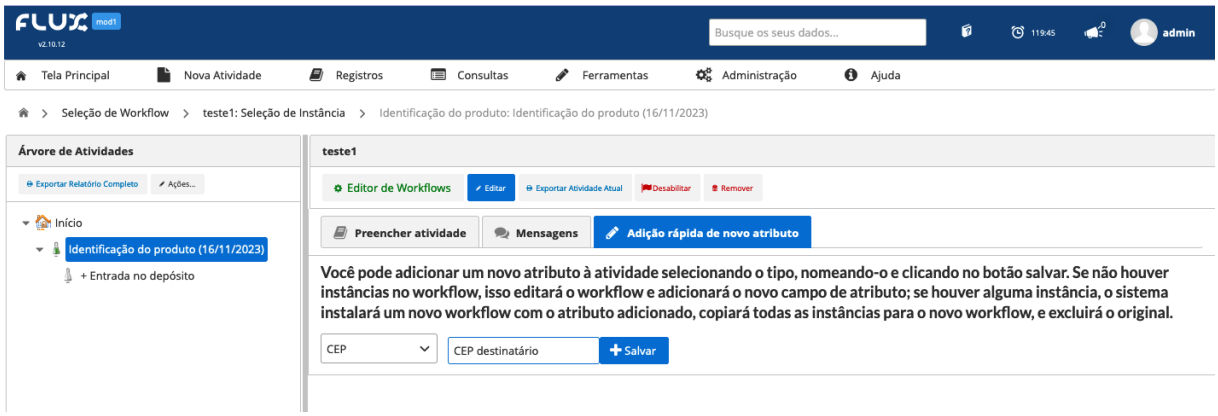
Figura 3.22: Editor de *workflows* com todas as novas permissões.



Fonte: Elaborado pelo autor.

3.1.2) que permite adição rápida de um atributo novo. Esta nova aba conta com um aviso que explica essa funcionalidade para o usuário, que consiste em criar um novo *workflow* contendo o campo especificado, copiar as instâncias do *workflow* original e, em seguida, apagar o original e todas as suas instâncias. As figuras 3.23 e 3.24 exemplificam, em ordem, esse fluxo.

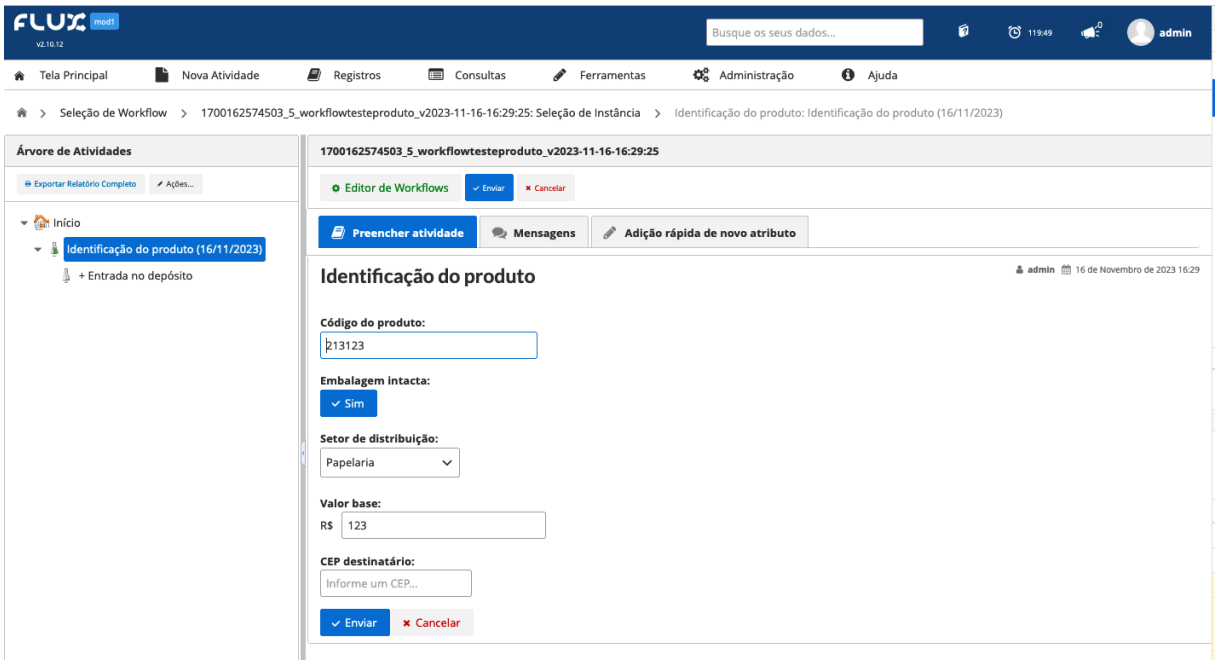
Figura 3.23: Seleção de tipo de atributo para adição rápida na tela de preenchimento de atividades.



Fonte: Elaborado pelo autor.

Reforçamos que a funcionalidade de deleção de um *workflow* só deve ser executada em casos específicos e que só pode ser executada por usuários habilitados para tanto. As trilhas de *Data Provenance* nos permitem conectar o histórico do *workflow* derivado ao do original, então não perdemos dados históricos ao deletar o original, porém os responsáveis pelos laboratórios podem preferir manter os dois *workflows* e comparar os campos presentes em ambos.

Figura 3.24: Novo atributo adicionado diretamente pela tela de atividades.



Fonte: Elaborado pelo autor.

3.2 Aplicações para as trilhas de Data Provenance

Nesta seção discutiremos algumas aplicações para as trilhas de *Data Provenance* que puderam ser observadas ao longo deste trabalho.

3.2.1 Casos de uso

Um caso de uso importante para esta aplicação do projeto foi com a equipe do CTVacinas [4] onde o uso do Flux foi otimizado pelas novas funcionalidades desenvolvidas, já que os usuários puderam ter acesso simplificado à troca de mensagens, histórico de preenchimento dos atributos, e alterações estruturais no *workflow*, sem contar com sistemas externos de trocas de mensagem ou de gestão de histórico de alteração.

A título de exemplo, descreveremos um caso de uso com três personagens, o técnico de laboratório Alberto, sua supervisora Bruna, e o gerente do laboratório Cláudio.

Inicialmente o técnico Alberto realiza os seguintes passos:

- Entra no sistema
- Abre o *workflow* W1 no qual está trabalhando

- Preenche uma atividade nova e anexa uma mensagem com uma dúvida para a sua supervisora

  Em seguida a supervisora Bruna:

- Entra no sistema e percebe que tem uma nova mensagem
- É direcionada para a atividade preenchida e verifica o seu histórico
- Percebe que a dúvida é pertinente, e que o *workflow* precisa de um novo atributo
- Adiciona o novo campo em tempo real
- Envia uma mensagem avisando Alberto sobre a decisão

  E então o gerente Cláudio:

- Entra no sistema e verifica como está o *workflow* W1
- Percebe que tem um novo campo
- Navega para a tela de linhagem de dados para verificar quem adicionou o novo campo

Como podemos ver por este exemplo, as mudanças realizadas neste trabalho além de adicionar confiabilidade e rastreabilidade, também melhoram a experiência dos diversos tipos de usuários finais.

Nas próximas subseções discutiremos como todo o fluxo citado acima pode ser acionado diretamente pela tela de preenchimento de atividades, trazendo muita agilidade e facilidade de uso para os usuários.

3.2.2 Auditoria

Assim como trilhas de auditoria convencionais (seção 1.2.2), os grafos de *Data Provenance* podem ser aplicados na auditoria de *software* para avaliar como uma entidade evoluiu ao longo do tempo, já que respondem às perguntas “como?”, “por quem?”, e “quando?” um dado transitou entre os estados, além de incluir contextos preciosos sobre estas alterações, portanto, neste trabalho criamos um sistema de auditoria mais completo do que o anterior do Flux.

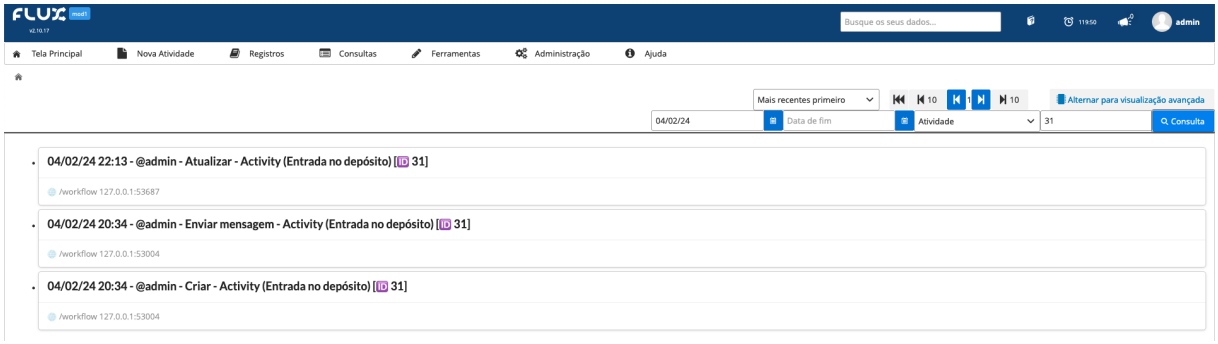
A título de exemplo, para termos o histórico de uma entidade basta selecionarmos o tipo e identificador desta na tela de linhagem de dados mostrada anteriormente, podemos

fazer o mesmo para ver como um usuário utilizou o sistema ao longo do tempo utilizando a filtragem por usuários.

Na figura 3.25 temos exemplificada a auditoria de todas as operações que envolveram a atividade com identificador 31, incluindo as mensagens enviadas a partir da mesma.

Seguindo este exemplo, também podemos usar a visualização avançada para abrir os metadados de cada registro e ter acesso às mensagens trocadas em cada alteração e obter ainda mais contexto sobre as mudanças realizadas do que uma trilha de auditoria convencional nos traria.

Figura 3.25: Auditoria de uma entidade usando as visualizações de dados disponíveis.



Fonte: Elaborado pelo autor.

3.2.3 Troca de mensagens

Como mencionado na seção 2.5.2, os *templates* de *Data Provenance* se mostraram elementos muito práticos para a construção de sistemas de troca de mensagens por carregarem consigo todo o contexto de uma alteração realizada.

Usuários tem a possibilidade de discutir os trabalhos realizados sem a necessidade de descrever manualmente o que foi feito, ou podem pedir aprovação a um usuário administrador de um preenchimento de atividades utilizando a ferramenta de anexar mensagens diretamente da tela de preenchimento.

Essas alterações trouxeram uma grande melhoria na experiência dos usuários com o sistema Flux, além de explorarem um caso de uso pouco trabalhado na literatura de *Data Provenance* e auditoria de *software* em geral ao acoplarmos a troca de mensagens com os registros em si.

3.2.4 Rastreabilidade entre entidades distintas

As trilhas de *Data Provenance* nos habilitam a identificar interações entre entidades distintas, pois, diferentemente das trilhas de auditoria convencionais que apenas registram acontecimentos de uma determinada entidade, os grafos gerados indicam como uma entidade evoluiu e se relacionou com outras entidades, agentes, e atividades durante o seu ciclo de vida.

Essa propriedade foi crucial para a funcionalidade de edição de *workflows* em tempo real porque permite que consigamos rastrear precisamente o momento em que houve a bifurcação no histórico dessa entidade e como os dois *workflows* evoluíram concorrentemente ao longo do tempo após aplicada a operação. Apenas com a auditoria convencional existente no Flux sem esta propriedade não seria possível que essa *feature* fosse desenvolvida, já que o sistema não contaria com os recursos de rastreabilidade necessários para apoiá-la.

Na figura 3.26 temos um exemplo desta aplicação, onde buscamos pelos dados relacionados à entidade *workflow* com identificador 37 e observamos que esta entidade foi derivada de outro *workflow*.

Figura 3.26: Grafo gerado a partir dos registros envolvendo o *workflow* com identificador 37.

Fonte: Elaborado pelo autor.

Além das informações que podemos extrair visualmente, os grafos de *Data Provenance* são estruturas muito interessantes para aplicação de técnicas de aprendizado de máquinas diretamente do banco de dados para que padrões possam ser encontrados automatizadamente. Ao aplicar técnicas de *data mining* nos grafos de *Data Provenance*

podemos, por exemplo, podemos identificar que um atributo foi editado muito mais do que os outros o que pode ser um indicador de que este passo do experimento não está sendo executado corretamente, ou que um conjunto de atividades sempre estão sendo preenchidas simultaneamente o que pode indicar que elas poderiam ser unificadas em uma nova versão do *workflow*.

Capítulo 4

Conclusão

Neste último capítulo concluiremos o trabalho com uma reflexão de tudo que foi feito e como atingimos os resultados que buscamos inicialmente e abriremos portas para trabalhos futuros que podem se basear neste.

4.1 Considerações finais

Este trabalho tratou da contextualização e implementação de trilhas de *Data Provenance* para LIMS 1.2.4, baseando-se em experiências em sistemas similares [5] [7].

Alteramos o sistema Flux [9] [8] [22] para coletar estas informações e com isso conseguimos construir duas *features* interessantes para os usuários, a troca de mensagens e a edição de *workflows* em tempo real.

Através das visualizações de dados desenvolvidos, trouxemos melhorias na forma de ler, interagir e compreender as alterações feitas no Flux pelos seus diversos perfis de usuários.

Como a obtenção dos dados foi feita por meio de *listeners* de eventos desacoplados do restante do sistema, e os administradores possuem controle sobre a quantidade de dados que querem que sejam armazenados 2.4.1, o objetivo de que a extração de *Data Provenance* fosse feita de forma não-intrusiva foi atingido.

Os cinco itens inicialmente elencados como essenciais para sistemas laboratoriais foram endereçados neste projeto:

- Confiabilidade, já que as trilhas são coletados diretamente da integração com o banco de dados, o nível mais baixo que conseguiríamos ir sem uma reestruturação arquitetural do sistema.
- Reprodutibilidade, os grafos obtidos nos contam a história de como o experimento foi realizado.

- Rastreabilidade, devido às trilhas adicionais e a possibilidade de visualizar interconexões entre as entidades.
- Acessibilidade, os dados são de fácil acesso e formatados para otimizar a leitura humana.
- Confidencialidade, apenas usuários administradores tem acesso aos dados que foram obtidos pelo módulo desenvolvido.

4.2 Trabalhos futuros

Nesta seção trataremos de alguns trabalhos que podem ser feitos futuramente com base neste, podendo servir como base para outras dissertações de mestrado, projetos profissionais, ou para teses de doutorado.

4.2.1 Exportação do módulo de Data Provenance como um serviço genérico

Um dos objetivos deste trabalho era de que o módulo de *Data Provenance* fosse possível de ser exportado como um serviço genérico capaz de ser consumido por qualquer sistema.

Com as técnicas de arquitetura aplicadas discutidas na seção 2.3.2, esse trabalho será relativamente fácil de ser realizado, já que seria necessário apenas implementar a interface *IProvenanceRepository* e uma forma de acesso ao sistema, seja via um servidor *web* ou por alguma aplicação de linha de comando, por exemplo.

Caso este caminho seja adotado no futuro, é a recomendação do autor que uma base de dados orientada a grafos, como o *Neo4j* [19], seja utilizada para persistir as trilhas recolhidas, já que estas são otimizadas para os tipos de dados trabalhados pelo sistema.

4.2.2 Detecção automática de falhas através da auditoria contínua

Como mencionado na seção 1.2.2, o estado da arte da auditoria de *software* está atualmente relacionado com a obtenção de dados e avaliação destes por meio de testes automatizados em tempo real [11].

Esta ideia também pode ser aplicada nas trilhas de *Data Provenance* obtidas, desde que os algoritmos para os testes sejam adequados para trabalharem com os dados em formato de grafo ao invés do formato tabular estruturado.

As estratégias descritas no estudo referenciado em [11] podem ser aplicadas diretamente no serviço de *Data Provenance* (ver seção 4.2.1) e seria plenamente possível que o este também contasse com uma interface para executar testes automatizados, que deveria ser implementada pelo consumidor, seguindo o princípio da inversão de dependências (mais detalhes na referência [21]).

4.2.3 Avaliação automática de ciclo de vida de entidades

Como mencionado na seção 3.2.4, é possível utilizar os dados do sistema para extrair padrões e *insights* com os dados obtidos no formato de grafos.

Nesta seção propomos um passo além, sendo utilizar técnicas de aprendizado profundo em grafos [23] para processar as informações de *Data Provenance* e construir *dashboards* inteligentes que tragam conteúdos relevantes para os usuários administradores, através do serviço de *Data Provenance* (ver seção 4.2.1).

Propomos que, além dos requisitos já citados, o serviço exportado a partir do módulo desenvolvido conte com a possibilidade de executar modelos de aprendizado de máquina que podem ser selecionados e ajustados pelos administradores para que toda essa integração e processamento da informação ocorra de forma imperceptível para os sistemas consumidores externos.

Desta forma, o sistema não só irá executar testes pré-configurados (ver seção 4.2.2), como também conseguirá avaliar o ciclo de vida das entidades envolvidas, e detectar quando alguma entidade passar a exibir comportamento fora deste padrão, gerando assim uma espécie de detecção automatizada de falhas, sejam elas sistêmicas ou humanas.

Referências

[1] DCMI Usage Board. Dcmi metadata terms. Standard, DCMI Usage Board, Dublin, IE, January 2020. URL https://www.dublincore.org/specifications/dublin-core/dcmi-terms/.

[2] Workflow Management Coalition. Workflow management coalition, March 2024. URL https://wfmc.org/.

[3] Community. Hibernate, April 2023. URL https://hibernate.org/.

[4] CTVacinas. Ctvacinas, March 2024. URL http://www.ctvacinas.ufmg.br/.

[5] V. Curcin, S. Miles, R. Danger, Y. Chen, R. Bache, and A. Taweel. Implementing interoperable provenance in biomedical research. *Future Generation Computer Systems*, 34:1–16, 2014. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2013.12.001. URL https://www.sciencedirect.com/science/article/pii/S0167739X13002653. Special Section: Distributed Solutions for Ubiquitous Computing and Ambient Intelligence.

[6] Vasa Curcin, Elliot Fairweather, Roxana Danger, and Derek Corrigan. Templates as a method for implementing data provenance in decision support systems. *Journal of Biomedical Informatics*, 65:1–21, 2017. ISSN 1532-0464. doi: https://doi.org/10.1016/j.jbi.2016.10.022. URL https://www.sciencedirect.com/science/article/pii/S1532046416301599.

[7] Brendan C. Delaney, Vasa Curcin, Anna Andreasson, Theodoros N. Arvanitis, Hilde Bastiaens, Derek Corrigan, Jean-Francois Ethier, Olga Kostopoulou, Wolfgang Kuchinke, Mark McGilchrist, Paul van Royen, and Peter Wagner. Translational medicine and patient safety in europe: Transform—architecture for the learning health system in europe. *BioMed Research International*, 2015:961526, Oct 2015. ISSN 2314-6133. doi: 10.1155/2015/961526. URL https://doi.org/10.1155/2015/961526.

[8] Alessandra C. Faria-Campos, Luciene B. Balottin, Gianlucca Zuin, Vinicius Garcia, Paulo HS Batista, José M. Granjeiro, and Sérgio VA Campos. Fluxcttx: A lims-based tool for management and analysis of cytotoxicity assays data. *BMC Bioinformatics*, 16(19):S8, Dec 2015. ISSN 1471-2105. doi: 10.1186/1471-2105-16-S19-S8. URL https://doi.org/10.1186/1471-2105-16-S19-S8.

[9] Alessandra C. Faria-Campos, Lucas A. Hanke, Paulo HS Batista, Vinicius Garcia, and Sérgio VA Campos. An innovative electronic health record system for rare and complex diseases. *BMC Bioinformatics*, 16(19):S4, Dec 2015. ISSN 1471-2105. doi: 10.1186/1471-2105-16-S19-S4. URL https://doi.org/10.1186/1471-2105-16-S19-S4.

[10] Damian Fessey. Software auditing - why, and how...? *Computer Audit Update*, 1994:12–20, 1994. ISSN 0960-2593. doi: https://doi.org/10.1016/0960-2593(94)90045-0. URL https://www.sciencedirect.com/science/article/pii/0960259394900450.

[11] S. Flowerday, A.W. Blundell, and R. Von Solms. Continuous auditing technologies and models: A discussion. *Computers & Security*, 25(5):325–331, 2006. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2006.06.004. URL https://www.sciencedirect.com/science/article/pii/S0167404806000964.

[12] International Organization for Standardization. Geographic information — Metadata — Part 2: Extensions for acquisition and processing. Standard, International Organization for Standardization, Geneva, CH, January 2019. URL https://www.iso.org/standard/67039.html.

[13] The R Foundation. The r project for statistical computing, May 2024. URL https://www.r-project.org/.

[14] Martin Fowler. *Patterns of Enterprise Application Architecture*. The Addison-Wesley Signature Series. Addison Wesley, May 2002. ISBN 0-321-12742-0. URL https://www.martinfowler.com/books/eaa.html.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addision-Wesley Professional Computing Series. Addison-Wesley Professional, October 1994. ISBN 0201633612.

[16] Yolanda Gil and Simon Miles. PROV Model Primer, 2013. URL https://www.w3.org/TR/2013/NOTE-prov-primer-20130430.

[17] Paul Groth and Luc Moreau. An Overview of the PROV Family of Documents, April 2013. URL https://www.w3.org/TR/prov-overview/.

[18] Mary D. Hinton. *Laboratory information management systems: development and implementation for a quality assurance laboratory*. Marcel Dekker Inc., 1995. ISBN 0824794583.

[19] Neo4j Inc. Neo4j, 2023. URL https://neo4j.com/.

[20] Timothy Lebo, Satya Sahoo, and Deborah McGuinness. PROV-O: The PROV Ontology, 2013. URL https://www.w3.org/TR/2013/REC-prov-o-20130430.

[21] Robert C. Martin. *Clean Architecture, A Craftsman's Guide To Software Structure And Design*. Robert C. Martin Series. Prentice Hall, October 2017. ISBN 0-13-449416-4.

[22] Alexandre Melo, Alessandra Faria-Campos, Daiane Mariele DeLaat, Rodrigo Keller, Vinícius Abreu, and Sérgio Campos. Sigla: an adaptable lims for multiple laboratories. *BMC Genomics*, 11(5):S8, Dec 2010. ISSN 1471-2164. doi: 10.1186/1471-2164-11-S5-S8. URL https://doi.org/10.1186/1471-2164-11-S5-S8.

[23] Luca Oneto, Nicoló Navarin, Battista Biggio, Federico Errica, Alessio Micheli, Franco Scarselli, Monica Bianchini, Luca Demetrio, Pietro Bongini, Armando Tacchella, and Alessandro Sperduti. Towards learning trustworthily, automatically, and with guarantees on graphs: An overview. *Neurocomputing*, 493:217–243, 2022. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2022.04.072. URL https://www.sciencedirect.com/science/article/pii/S0925231222004520.

[24] Wil M. P. van der Aalst. Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language, 2003. URL http://www.padsweb.rwth-aachen.de/wvdaalst/publications/p201.pdf.

[25] Wil M. P. van der Aalst. *Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management*, pages 1–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-27755-2. doi: 10.1007/978-3-540-27755-2_1. URL https://doi.org/10.1007/978-3-540-27755-2_1.

Apêndice A

Código java da interface IProvenance

```
package br.ufmg.dcc.luar.flux.provenance.interfaces;

import java.util.Date;
import java.util.List;
import java.util.UUID;

import br.ufmg.dcc.luar.flux.provenance.models.NodeType;
import br.ufmg.dcc.luar.flux.provenance.models.ProvenanceGraph;
import br.ufmg.dcc.luar.flux.provenance.models.ProvenanceMessage
    ;
import br.ufmg.dcc.luar.flux.provenance.models.
    ProvenanceMessageBasic;
import br.ufmg.dcc.luar.flux.provenance.models.
    ProvenanceMessageUser;
import br.ufmg.dcc.luar.flux.provenance.models.ProvenanceTrace;

public interface IProvenance {
    public Long logTrace(ProvenanceTrace trace);

    public Long logTrace(ProvenanceTrace trace, UUID spanId);

    public List<ProvenanceTrace> getTracesForReference(String
        reference, int offset, int limit, boolean
        descendingTimestamp);

    public List<ProvenanceTrace> getTracesForReference(String
        reference, Date minimumDate, Date maximumDate, int offset
        , int limit, boolean descendingTimestamp);

    public List<ProvenanceTrace> getTracesForReference(NodeType
```

```
    nodeType, String reference, Date minimumDate, Date
    maximumDate, int offset, int limit, boolean
    descendingTimestamp);

public ProvenanceGraph getProvenanceGraphForReference(String
     reference, int offset, int limit, boolean
    descendingTimestamp);

public ProvenanceTrace getTrace(Long traceId);

public List<ProvenanceTrace> getUserMessagesSent(Long userId
    );

public List<ProvenanceTrace> getUserMessagesReceived(Long
    userId);

public List<ProvenanceMessage> getThreadMessages(String
    threadId);

public void sendMessageToReference(String message, String
    referenceId, ProvenanceMessageUser sender, List<
    ProvenanceMessageUser> receivers);

public void saveMessageToThread(String message, String
    threadId, ProvenanceMessageUser sender, List<
    ProvenanceMessageUser> receivers);

public String saveMessageToTrace(String message, Long
    traceId, ProvenanceMessageUser sender, List<
    ProvenanceMessageUser> receivers);

public List<ProvenanceTrace> getTracesByRequestId(String
    requestId, int offset, int limit, boolean
    descendingTimestamp);

public List<ProvenanceTrace> getTracesBySessionId(String
    session, int offset, int limit, boolean
    descendingTimestamp);
```

```
    void markThreadAsRead(String threadId, Long userId);

    public void removeMessageFromTrace(Long traceId);

    public void addMessageToTrace(Long traceId, Long messageId);
}
```

Apêndice B

Código java da interface IProvenanceRepository

```
package br.ufmg.dcc.luar.flux.provenance.interfaces;

import java.util.Date;
import java.util.List;
import java.util.Set;
import java.util.UUID;

import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryEdge;
import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryMail;
import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryMailMessage;
import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryMessage;
import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryNode;
import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryTrace;
import br.ufmg.dcc.luar.flux.provenance.interfaces.models.
    IProvenanceRepositoryUser;
import br.ufmg.dcc.luar.flux.provenance.models.NodeType;

public interface IProvenanceRepository {
    public Object beginTransaction();

    public void commitTransaction(Object transaction);
```

```
public void rollbackTransaction(Object transaction);

public Long insertTrace(IProvenanceRepositoryTrace trace);

public List<Long> insertNodes(List<IProvenanceRepositoryNode
    > nodes);

public List<Long> insertEdges(List<IProvenanceRepositoryEdge
    > edges);

public Long insertMessage(IProvenanceRepositoryMessage
    message);

public List<Long> insertMail(List<IProvenanceRepositoryMail>
     mail);

public List<Long> upsertUsers(List<IProvenanceRepositoryUser
    > users);

public IProvenanceRepositoryTrace getTraceById(Long traceId)
    ;

public List<IProvenanceRepositoryTrace> getTracesById(Set<
    Long> traceIds);

public List<IProvenanceRepositoryEdge> getEdgesByTraceId(
    Long traceId);

public List<IProvenanceRepositoryNode> getNodesByTraceId(
    Long traceId);

public void updateTraceMessageId(Long traceId, Long
    messageId);

public IProvenanceRepositoryMessage getMessage(Long
    messageId);

public List<IProvenanceRepositoryMail> getMessageMail(Long
    messageId);
```

```
    public List<IProvenanceRepositoryUser> getMessageUsers(Long
        messageId);

    public void setMessageReceiverRead(Long messageId, Boolean
        receiverRead);

    public List<IProvenanceRepositoryMessage> getThreadMessages(
        String threadId);

    public Set<Long> getTraceIdsByReference(NodeType nodeType,
        String reference, Date minimumDate, Date maximumDate, int
         offset, int limit, boolean descendingTimestamp);

    public List<IProvenanceRepositoryTrace> getTracesByRequestId
        (String requestId, int offset, int limit, boolean
        descendingTimestamp);

    public List<IProvenanceRepositoryTrace> getTracesBySessionId
        (String sessionId, int offset, int limit, boolean
        descendingTimestamp);

    public IProvenanceRepositoryUser getUser(Long userId);

    public List<IProvenanceRepositoryTrace> getUserMessagesSent(
        Long userId);

    public List<IProvenanceRepositoryTrace>
        getUserMessagesReceived(Long userId);

    void markThreadAsRead(String threadId, Long userId);
}
```

Apêndice C

Código java dos modelos de dados utilizados

```
public interface IProvenanceRepositoryEdge {
    public Long getTraceId();
    public Long getEdgeId();
    public Long getOriginNodeId();
    public Long getTargetNodeId();
    public EdgeType getEdgeType();
}

public interface IProvenanceRepositoryMail {
    public Long getMessageId();
    public Long getSenderId();
    public Long getReceiverId();
    public Boolean getRead();
}

public interface IProvenanceRepositoryMailMessage extends
   IProvenanceRepositoryMail {
    public String getMessageText();
    public Date getTimestamp();
    public String getThreadId();
}

public interface IProvenanceRepositoryMessage {
    public Long getMessageId();
    public String getThreadId();
    public String getMessageText();
    public Date getTimestamp();
```

```
}

public interface IProvenanceRepositoryNode {
    public Long getNodeId();
    public Long getTraceId();
    public NodeType getNodeType();
    public String getReference();
    public List<IProvenanceRepositoryNodeAnnotation>
       getAnnotations();
}

public interface IProvenanceRepositoryNodeAnnotation {
    public String getKey();
    public String getValue();
}

public interface IProvenanceRepositoryTrace {
    public Long getTraceId();
    public Long getMessageId();
    public Date getTimestamp();
    public UUID getSpanId();
    public String getSessionId();
    public String getRequestId();
}

public interface IProvenanceRepositoryUser {
    public Long getUserId();
    public String getReference();
}
```

Apêndice D

Código java do ouvinte de eventos do banco de dados

```
package br.ufmg.dcc.luar.flux.listener;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.UUID;

import javax.persistence.EntityManager;
import javax.servlet.http.HttpServletRequest;

import org.hibernate.event.spi.PostCommitInsertEventListener;
import org.hibernate.event.spi.PostDeleteEvent;
import org.hibernate.event.spi.PostDeleteEventListener;
import org.hibernate.event.spi.PostInsertEvent;
import org.hibernate.event.spi.PostLoadEvent;
import org.hibernate.event.spi.PostLoadEventListener;
import org.hibernate.event.spi.PostUpdateEvent;
import org.hibernate.event.spi.PostUpdateEventListener;
import org.hibernate.persister.entity.EntityPersister;
import org.springframework.web.context.request.
    RequestContextHolder;
import org.springframework.web.context.request.
    ServletRequestAttributes;

import com.google.gson.Gson;
```

```
import br.ufmg.dcc.luar.flux.beans.LoginBean;
import br.ufmg.dcc.luar.flux.engine.provenance.ProvenanceIgnore;
import br.ufmg.dcc.luar.flux.engine.provenance.
    ProvenanceLogEngine;
import br.ufmg.dcc.luar.flux.facade.provenance.
    ProvenanceManagementFacade;
import br.ufmg.dcc.luar.flux.model.AbstractEntity;
import br.ufmg.dcc.luar.flux.model.FluxRevision;
import br.ufmg.dcc.luar.flux.model.User;
import br.ufmg.dcc.luar.flux.model.provenance.
    ProvenanceManagementEntity;
import br.ufmg.dcc.luar.flux.provenance.models.
    ProvenanceConstants;
import br.ufmg.dcc.luar.flux.provenance.models.
    ProvenanceMessageFull;
import br.ufmg.dcc.luar.flux.service.JpaService;
import br.ufmg.dcc.luar.flux.util.JSFUtil;

public class ProvenanceHibernateListener implements
    PostCommitInsertEventListener, PostUpdateEventListener,
    PostLoadEventListener, PostDeleteEventListener {
     @Override
     public boolean requiresPostCommitHanding(EntityPersister
        persister) {
         return true;
     }

     @Override
     public void onPostInsert(PostInsertEvent event) {
         ProvenanceLogEngine provenanceEngine =
            ProvenanceLogEngine.buildProvenanceEngine(event.
            getEntity().getClass().getSimpleName().toLowerCase(),
             false);
         if (!shouldLogProvenance(provenanceEngine, event.
            getEntity(), true, false, false, false))
             return;

         HashMap<String, String> fields = new HashMap<String,
            String>();
```

```
        for (int i = 0; i < event.getState().length; i++) {
            fields.put(event.getPersister().getPropertyNames()[i
                ], fieldToString(event.getState()[i]));
        }
        provenanceEngine.addInsertTrace((AbstractEntity) event.
            getEntity(), fields);
        provenanceEngine.commitTrace();
    }

    @Override
    public void onPostUpdate(PostUpdateEvent event) {
        ProvenanceLogEngine provenanceEngine =
            ProvenanceLogEngine.buildProvenanceEngine(event.
            getEntity().getClass().getSimpleName().toLowerCase(),
             false);
        if (!shouldLogProvenance(provenanceEngine, event.
            getEntity(), false, true, false, false))
             return;

        HashMap<String, String> oldFields = new HashMap<String,
            String>();
        HashMap<String, String> updatedFields = new HashMap<
            String, String>();
        for (int index : event.getDirtyProperties()) {
            oldFields.put(event.getPersister().getPropertyNames
                ()[index], fieldToString(event.getOldState()[
                index]));
            updatedFields.put(event.getPersister().
                getPropertyNames()[index], fieldToString(event.
                getState()[index]));
        }

        provenanceEngine.addUpdateTrace((AbstractEntity) event.
            getEntity(), oldFields, updatedFields);
        provenanceEngine.commitTrace();
    }

    @Override
    public void onPostLoad(PostLoadEvent event) {
```

```
        ProvenanceLogEngine provenanceEngine =
            ProvenanceLogEngine.buildProvenanceEngine(event.
            getEntity().getClass().getSimpleName().toLowerCase(),
             true);
        if (!shouldLogProvenance(provenanceEngine, event.
            getEntity(), false, false, true, false))
             return;
        provenanceEngine.addLookupTrace((AbstractEntity) event.
            getEntity());
        provenanceEngine.commitTrace();
    }

    @Override
    public void onPostDelete(PostDeleteEvent event) {
        ProvenanceLogEngine provenanceEngine =
            ProvenanceLogEngine.buildProvenanceEngine(event.
            getEntity().getClass().getSimpleName().toLowerCase(),
             false);
        if (!shouldLogProvenance(provenanceEngine, event.
            getEntity(), false, false, false, true))
             return;
        provenanceEngine.addDeleteTrace((AbstractEntity) event.
            getEntity());
        provenanceEngine.commitTrace();
    }

    private Boolean shouldLogProvenance(ProvenanceLogEngine
        engine, Object entity, boolean insert, boolean update,
        boolean lookup, boolean delete) {
        if (engine.hasMessage())
             return true;

        if (entity.getClass().isAnnotationPresent(
            ProvenanceIgnore.class)) {
             ProvenanceIgnore ignore = entity.getClass().
                 getAnnotation(ProvenanceIgnore.class);
             if (ignore.ignoreAll()) {
                  return false;
             }
```

```
            if (insert && ignore.ignoreInsert()) {
                return false;
            }
            if (update && ignore.ignoreUpdate()) {
                return false;
            }
            if (lookup && ignore.ignoreLookup()) {
                return false;
            }
            if (delete && ignore.ignoreDelete()) {
                return false;
            }
        }

        ProvenanceManagementFacade managementFacade = new
            ProvenanceManagementFacade();
        ProvenanceManagementEntity managementEntity =
            managementFacade.findByEntityName(entity.getClass().
            getName());
        if (managementEntity == null) {
            return false;
        }
        if (insert && !managementEntity.getCreateEnabled()) {
            return false;
        }
        if (lookup && !managementEntity.getReadEnabled()) {
            return false;
        }
        if (update && !managementEntity.getUpdateEnabled()) {
            return false;
        }
        if (delete && !managementEntity.getDeleteEnabled()) {
            return false;
        }

        return (entity instanceof AbstractEntity) && (!(entity
            instanceof FluxRevision));
    }
```

```
    private String fieldToString(Object field) {
        if (field == null)
            return "null";
        return field.toString();
    }

    @Override
    public void onPostInsertCommitFailed(PostInsertEvent event)
       {
    }
}
```