ColaBD - Uma Plataforma Web colaborativa para mapeamento e modelagem relacional de banco de dados

Ruan Sanchez[1], **Maria Vitoria de Freitas**[1], **Fábio Aiub Sperotto**[1]

[1] Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina
R. Heitor Villa Lobos, 225 - São Francisco, Lages - SC, 88506-400 – Brasil

{ruan.hss10, maria.v07}@aluno.ifsc.edu.br

fabio.sperotto@ifsc.edu.br

Abstract. *Collaborative database modeling can be a challenging process, especially when it involves remote collaboration and integration with different Database Management Systems (DBMSs). To facilitate this process, this paper presents ColaBD, a web platform that enables real-time collaborative database mapping and modeling, offering support for multiple DBMSs. The platform allows several users to work together in an integrated way, making the development workflow more dynamic and efficient. The ColaBD platform obtained positive evaluations among users who tested the system, making it an option in relational database modeling activities in a collaborative way.*

Keywords: Real time, WebSocket, Database, Modeling, Collaborative, Angular, FastAPI

Resumo. *A modelagem de um bancos de dados em equipe pode ser um processo desafiador, especialmente quando envolve colaboração remota e integração com diferentes Sistemas Gerenciadores de Banco de Dados (SGBDs). Para facilitar esse processo, este artigo apresenta o ColaBD, uma plataforma web que permite o mapeamento e a modelagem colaborativa de bancos de dados em tempo real, oferecendo suporte a múltiplos SGBDs. A plataforma possibilita que diversos usuários trabalhem juntos de forma integrada, tornando o fluxo de desenvolvimento mais dinâmico e eficiente. A implementação envolve tecnologias como Angular e FastAPI para comunicação e manipulação de dados. A plataforma ColaBD obteve positivas avaliações entre os usuários que testaram o sistema, tornando-a como uma opção nas atividades de modelagem de banco de dados relacional de forma colaborativa.*

Palavras-chaves: Tempo real, WebSocket, Banco de dados, Modelagem, Colaboração, Angular, FastAPI

1. Introdução

Apesar da importância da modelagem para a integridade e eficiência dos sistemas, de acordo com Booch et al. (2005), as ferramentas tradicionais utilizadas nesse processo muitas vezes carecem de recursos que atendam às demandas de equipes modernas de desenvolvimento, sendo pouco eficazes em ambientes que exigem mudanças rápidas e colaboração constante.

Na experiência dos autores, ferramentas de modelagem de dados com suporte ao compartilhamento e à colaboração são fundamentais para a produtividade das equipes, sobretudo em

ambientes de trabalho remoto. A ausência desses recursos tende a dificultar a comunicação, gerar desalinhamentos e aumentar o retrabalho, enquanto soluções colaborativas favorecem a padronização, a agilidade e a integração entre os profissionais.

Segundo Ambler (2003), ferramentas clássicas de modelagem de dados tendem a ser orientadas a documentação estática e não oferecem suporte adequado para práticas ágeis, como integração contínua ou versionamento de modelos. Além disso, essas soluções frequentemente operam de forma isolada, dificultando a integração com ambientes DevOps[1] e com outras ferramentas utilizadas ao longo do ciclo de desenvolvimento de software, como por exemplo o BR Modelo Web (BRMW)[2]. Essa limitação afeta diretamente a produtividade e a consistência entre diferentes etapas do projeto.

De acordo com Silberchatz et al. (2020), a criação de uma aplicação de banco de dados é uma tarefa complexa, envolvendo os projetos de esquema de banco de dados, dos programas que acessam e atualizam os dados e o de um esquema de segurança para controlar o acesso a esses dados. Sendo assim Elmasri e Navathe (2018) ressaltou que ao desenvolver um projeto é necessário levantamento de requisitos e a construção do projeto conceitual, fases importantes para o inicio de um projeto.

Contudo Date (2004) afirma que o projeto de banco de dados deve iniciar com a compreensão profunda dos requisitos informacionais dos usuários, buscando identificar quais dados são relevantes, como se relacionam entre si e quais restrições devem ser respeitadas para manter a integridade das informações. Após essa análise inicial, segue-se o projeto conceitual, no qual se constrói uma representação lógica dos dados de forma independente de qualquer tecnologia ou Sistema Gerenciador de Banco de Dados (SGBD) específico. Consequentemente, enfatiza-se a ideia de que um projeto conceitual bem estruturado é fundamental para que o banco de dados atenda aos objetivos do sistema e ofereça suporte confiável às necessidades dos usuários.

Posteriormente, na elaboração de modelos para os SGBDs, existem desafios que os profissionais como desenvolvedores e analistas de dados enfrentam diariamente. Entre as dificuldades existentes está a padronização de modelos e a colaboração entre membros de uma equipe de desenvolvimento, além da transformação de estruturas de dados complexas em modelos relacionais coerentes e compatíveis com os SGBDs utilizados (Elmasri e Navathe, 2018).

Para o presente trabalho, será desenvolvida uma plataforma que visa ser uma ferramenta de apoio ao processo de desenvolvimento de sistemas, promovendo a adoção de boas práticas de modelagem de dados e contribuindo para a melhoria da comunicação e colaboração entre profissionais da área de tecnologia da informação. Para a resolução da plataforma, os objetivos específicos são:

- Permitir a criação, edição e visualização de modelos relacionais de banco de dados de forma interativa e intuitiva;
- Oferecer suporte à colaboração em tempo real, permitindo que múltiplos usuários trabalhem simultaneamente no mesmo projeto;
- Viabilizar a exportação personalizada de scripts SQL, adaptados conforme o SGBD selecionado, inicialmente para PostgreSQL e MySQL;

[1] DevOps é uma cultura e conjunto de práticas que visa melhorar a interação entre as partes de Desenvolvimeto (parte responsável pelo desenvolvimento do produto) e Operações (parte que irá manter o produto estável e pronto para os clientes consumirem).

[2] Disponível em `https://www.brmodeloweb.com/lang/pt-br/index.html`

- Promover a reutilização e compartilhamento de modelos, facilitando a integração entre projetos e equipes.
- Permitir a visualização do histórico de alteração dos *schemas*.

A pesquisa caracteriza-se como aplicada, com abordagem qualitativa e de natureza exploratória. Como procedimento técnico será por pesquisa-ação, tendo como finalidade o desenvolvimento de um software voltado à modelagem colaborativa de banco de dados. Para alcançar os objetivos propostos, o trabalho será estruturado em quatro etapas distintas, conduzidas de forma incremental e iterativa, respeitando os princípios metodológicos definidos.

Na primeira etapa, será realizada uma revisão bibliográfica, acompanhada de uma análise comparativa de ferramentas de modelagem de dados já consolidadas no mercado. O objetivo é compreender as principais limitações das abordagens atuais e levantar boas práticas e funcionalidades que possam contribuir como referência para o desenvolvimento da proposta deste trabalho.

Na segunda etapa, será elaborado o escopo funcional da plataforma, com a definição dos principais requisitos e funcionalidades, seguido do desenvolvimento de protótipos de interface utilizando ferramentas como Figma. Na terceira etapa ocorrerá o desenvolvimento da aplicação. A aplicação será dividida entre *front-end* e *back-end*, empregando tecnologias modernas como Angular[3] para a interface web e Python[4] com FastAPI[5] para os serviços de *back-end*.

Na quarta e última etapa, a aplicação será testada com diferentes cenários de uso, validando a usabilidade, a integridade das exportações e a eficácia da colaboração entre usuários. Espera-se, com isso, entregar uma ferramenta funcional, acessível e extensível, que contribua significativamente para o processo de modelagem de bancos de dados em ambientes educacionais e profissionais.

2. Referencial Teórico

Esta seção está dividida em quatro subseções. A primeira aborda a modelagem de banco de dados relacional, destacando sua importância no desenvolvimento de sistemas. A segunda discute plataformas colaborativas em tempo real e seu papel na edição simultânea de dados. A terceira explora práticas colaborativas em engenharia de software, focando na integração de equipes para acelerar o desenvolvimento. Por fim, a última subseção apresenta projetos relacionados e suas contribuições para a modelagem de dados colaborativa.

2.1. Modelagem de Banco de Dados Relacional

A modelagem relacional é uma abordagem necessária para organizar dados em estruturas chamadas relações, popularmente conhecidas como tabelas. Esse modelo foi proposto por Codd (1970), como uma alternativa mais flexível e matematicamente fundamentada do que os modelos hierárquico e em rede utilizados na época. Segundo Codd (1970), o modelo relacional permite representar dados e seus relacionamentos de forma simples e lógica, usando conceitos da álgebra relacional e da lógica de predicados.

A álgebra relacional é um conjunto de operações matemáticas definidas sobre relações (tabelas) que permite consultar e transformar dados de forma sistemática. Essas operações incluem seleção, projeção, união, diferença, produto cartesiano e junções, e produzem como resultado outras relações. Isso garante previsibilidade e consistência nas manipulações dos dados, além de

[3]Disponível em `https://angular.dev/`
[4]Disponível em `https://www.python.org/`
[5]Disponível em `https://fastapi.tiangolo.com/`

fornecer uma base teórica sólida para o desenvolvimento de linguagens de consulta. A lógica de predicados complementa essa abordagem ao permitir a formulação de expressões lógicas que descrevem propriedades e condições sobre os dados, viabilizando consultas declarativas precisas e compreensíveis.

O principal objetivo da modelagem relacional é representar os dados de forma consistente e compreensível, mantendo a integridade e a eficiência no armazenamento e na recuperação das informações. Ela visa estruturar os dados de forma a minimizar a redundância, facilitar a manipulação por meio de linguagens declarativas (como SQL) e garantir que as dependências lógicas do sistema estejam bem definidas. Portanto esse tipo de modelagem resolve diversos desafios encontrados em abordagens anteriores, tais como: dificuldade de acesso a dados complexos; redundância e inconsistência de informações; rígida dependência entre a estrutura física e lógica dos dados; falta de independência dos dados em relação aos aplicativos.

Segundo Elmasri e Navathe (2018), o desenvolvimento de um banco de dados envolve seis fases principais, cada uma com um papel fundamental na criação de um sistema robusto e funcional:

- Levantamento de requisitos: identificação das necessidades dos usuários e do contexto de uso dos dados.
- Projeto conceitual: modelagem abstrata dos dados por meio de ferramentas como o modelo Entidade-Relacionamento (ER).
- Projeto lógico: transformação do modelo conceitual em um modelo compatível com o modelo relacional.
- Projeto físico: definição de aspectos de implementação como índices, armazenamento e particionamento de dados.
- Implementação: criação das estruturas no Sistema Gerenciador de Banco de Dados (SGBD) e inserção dos dados.
- Manutenção e evolução: monitoramento, ajustes e atualizações para garantir o desempenho e a integridade do banco ao longo do tempo.

Durante a etapa de projeto lógico, é feita a transposição do modelo conceitual para uma estrutura compatível com um SGBD, definindo tabelas, atributos, chaves primárias e estrangeiras, além da normalização. A plataforma proposta neste trabalho auxilia significativamente nesta etapa ao permitir a criação, edição e visualização interativa de modelos relacionais, promovendo clareza na estrutura dos dados e facilitando a validação colaborativa por diferentes membros da equipe, inclusive em tempo real.

Na etapa de projeto físico, o modelo lógico é convertido em estruturas reais no SGBD. Essa etapa envolvendo decisões sobre tipos de dados otimizados, criação de índices, definição de *constraints*, particionamento de tabelas, entre outros aspectos técnicos. A plataforma proposta visa ser um diferencial neste processo ao exportar scripts SQL personalizados para diferentes SGBDs e fornecer um serviço de edição simultânea para modelagem física do banco de dados, assim permitindo que uma ou mais pessoas trabalhem em conjunto de diferentes locais.

2.2. Plataformas Colaborativas em Tempo Real

A colaboração em tempo real permite que equipes interajam simultaneamente em um mesmo ambiente digital, mesmo com os colaboradores estando em diferentes localizações, sincronizando

informações e alterações de forma instantânea. Em contextos de engenharia de software, essa capacidade melhora significativamente a produtividade e reduz erros, acelerando o ciclo de desenvolvimento (Gruber et al., 2017).

Para viabilizar a colaboração em tempo real o conceito de *Computer Supported Cooperative Work* (CSCW) foi introduzido por Greif e Cashman (1988) na década de 1980, o CSCW investiga não apenas os aspectos técnicos envolvidos no desenvolvimento de ferramentas colaborativas, mas também fatores sociais e organizacionais que influenciam a eficácia da interação em grupo (Grudin, 1994). Essa abordagem torna-se importante para plataformas colaborativas em tempo real, pois auxilia na compreensão dos desafios relacionados à comunicação, coordenação e manutenção da consistência de dados durante atividades simultâneas realizadas por equipes de desenvolvimento (Ellis et al., 1991).

Ademais, os editores colaborativos em tempo real, ou *Real-Time Collaborative Editors* (RCEs) são sistemas que permitem a múltiplos usuários trabalharem simultaneamente sobre um mesmo documento ou modelo, mantendo réplicas distribuídas que garantem consistência entre todas as alterações feitas pelos participantes. Esses editores colaborativos são essenciais em contextos distribuídos, onde equipes precisam interagir de maneira fluida, mesmo com integrantes dispersos geograficamente, proporcionando maior produtividade, eficiência e redução de conflitos durante o trabalho colaborativo (Wang e Sun, 2016).

De acordo com Cherif e Imine (2015), os *Real-Time Collaborative Editors* (RCEs) devem atender a um conjunto de requisitos fundamentais que asseguram a integridade, a previsibilidade e a qualidade da experiência colaborativa. Esses requisitos são especialmente relevantes em sistemas distribuídos, onde múltiplos usuários interagem simultaneamente sobre réplicas locais de um mesmo documento, exigindo mecanismos robustos para manter a consistência entre as edições. Entre os principais requisitos identificados pelos autores, destacam-se:

- Convergência: As réplicas do documento devem convergir para um estado comum, mesmo diante de operações concorrentes.
- Preservação de intenção: As ações dos usuários devem ser refletidas de maneira que suas intenções originais sejam mantidas, evitando efeitos colaterais indesejados.
- Suporte a operações de desfazer *undo*: É fundamental que o sistema permita reverter ações, mesmo em ambientes distribuídos, garantindo que as operações de desfazer não comprometam a consistência do documento.

Esses três pilares, convergência, preservação de intenção e desfazer, constituem a base teórica e prática para o desenvolvimento de editores colaborativos confiáveis. Eles orientam o design de algoritmos e estruturas internas que sustentam a experiência de edição em tempo real, garantindo que múltiplos usuários possam interagir simultaneamente sem comprometer a integridade do documento compartilhado. A aderência a esses princípios é essencial para assegurar não apenas a consistência técnica das réplicas, mas também uma experiência de usuário fluida e previsível (Cherif e Imine, 2015). Além disso, esses requisitos influenciam diretamente a escolha das técnicas de controle de concorrência e sincronização adotadas em sistemas colaborativos, como as abordagens baseadas em *Operational Transformation* (OT) e *Conflict-free Replicated Data Types* (CRDTs).

Entretanto, um dos principais desafios relacionados à colaboração simultânea é garantir a consistência e integridade dos dados frente a alterações concorrentes realizadas por múltiplos

usuários. Para lidar com esse desafio, técnicas como *Operational Transformation* (OT) têm sido amplamente empregadas. Sun e Ellis (1998) destacam que a OT possibilita que edições simultâneas sejam integradas sem conflitos significativos, ajustando dinamicamente as operações em execução e garantindo que todos os participantes vejam uma versão coerente do documento ou modelo compartilhado.

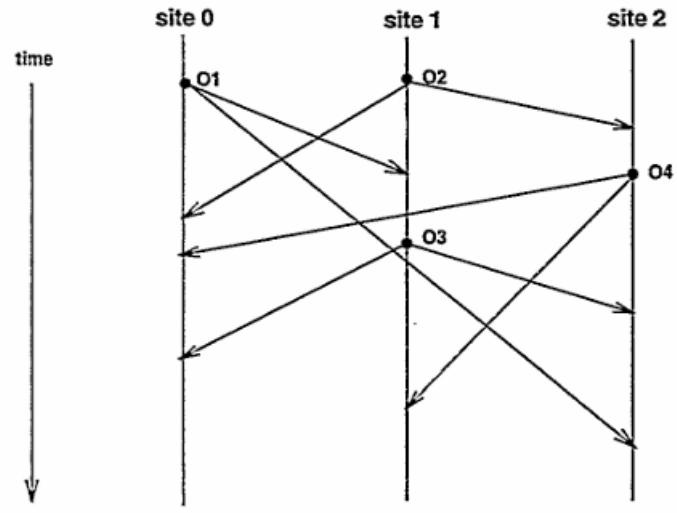

Figura 1. Um cenário de uma sessão de edição em grupo em tempo real. Fonte: Sun e Ellis (1998)

A Figura 1 representa um cenário típico de colaboração em grupo em tempo real, onde múltiplos usuários, distribuídos em diferentes locais, realizam operações de edição que precisam ser propagadas e aplicadas em todos os pontos da rede. Cada linha vertical indica a linha do tempo de um usuário, e as setas mostram como as ações se propagam entre os sites. Mesmo com poucas operações, já é possível observar a complexidade envolvida na ordem de execução, pois as ações podem ocorrer de forma concorrente ou em ordens distintas dependendo do momento em que chegam a cada site. Relações de dependência e independência entre as operações tornam o controle de concorrência um aspecto delicado, exigindo mecanismos sofisticados para garantir que todos os usuários tenham uma visão coerente do documento, independentemente das variações na ordem de recebimento e aplicação das operações Sun e Ellis (1998).

Mais recentemente, a técnica dos *Conflict-free Replicated Data Types* (CRDTs) emergiu como uma abordagem complementar eficiente para problemas de consistência em sistemas distribuídos. Segundo Shapiro et al. (2011), os CRDTs permitem operações concorrentes sem necessidade de sincronização estrita, o que melhora significativamente o desempenho e escalabilidade das plataformas colaborativas, tornando-os especialmente adequados para aplicações distribuídas em larga escala.

2.3. Projetos Relacionados

Esta subseção tem como objetivo apresentar e analisar plataformas existentes que abordam a modelagem de bancos de dados de maneira visual ou colaborativa. Por meio da revisão dessas ferramentas, busca-se compreender as soluções já disponíveis no mercado e identificar suas limitações e potencialidades, a fim de estabelecer um panorama que justifique o desenvolvimento da proposta deste trabalho.

Para uma análise mais organizada, os trabalhos relacionados foram divididos conforme a ferramenta analisada. Em cada item, são descritas as funcionalidades principais, os diferenciais e eventuais limitações observadas em relação à proposta deste trabalho, com ênfase nos aspectos de colaboração, exportação de modelos e suporte a diferentes SGBDs.

A DrawSQL[6] é uma plataforma online voltada para a criação e compartilhamento de diagramas de banco de dados. Seu foco está na usabilidade e na visualização gráfica de entidades e relacionamentos, permitindo a exportação dos modelos para formatos SQL compatíveis com diversos SGBDs. A Figura 2 ilustra a interface da ferramenta. Embora ofereça funcionalidades colaborativas, como comentários e compartilhamento controlado, a edição simultânea em tempo real é restrita aos planos pagos.

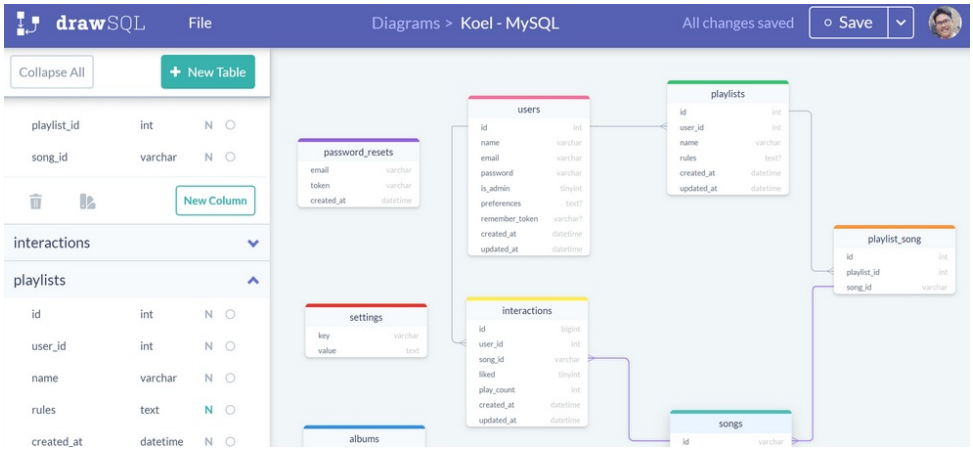

Figura 2. Página do serviço do Drawsql, mostrando modelagem lógica e relações entre tabelas.

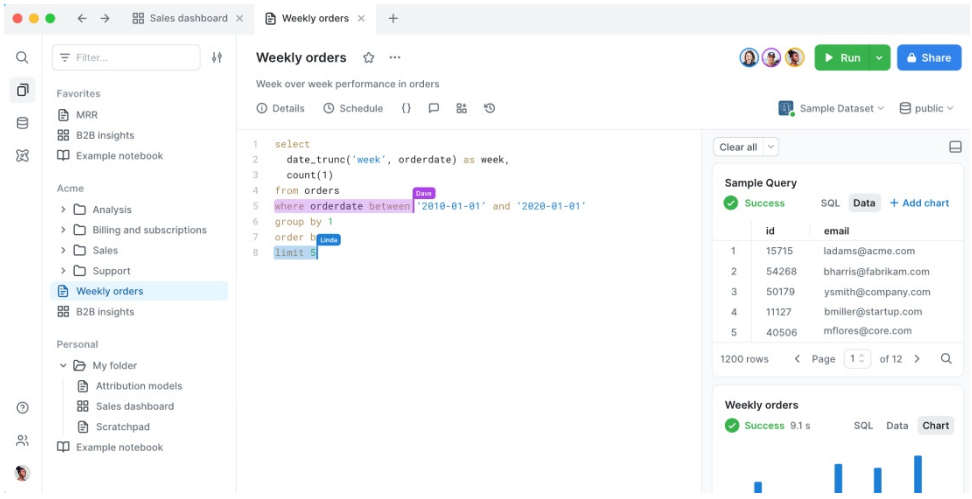

Figura 3. Página do serviço PopSQL, onde mostra uma colaboração simultânea na escrita de um script em SQL.

O PopSQL[7] é uma ferramenta colaborativa para criação, execução e compartilhamento de consultas SQL, com integração a diversos bancos de dados reais. Possui um plano gratuito com

[6]Disponível em `https://drawsql.app/`
[7]Disponível em `https://popsql.com/`

recursos básicos e planos pagos com funcionalidades avançadas e colaboração em equipe. Um dos seus diferenciais é a possibilidade de visualizar a estrutura do banco por meio de um navegador de esquemas, que reflete dinamicamente a estrutura atual do banco de dados conectado, como mostrado na Figura 3. Embora ofereça uma visão da estrutura do banco, o PopSQL não possui uma interface dedicada para modelagem visual completa, como diagramas ER.

O DBDiagram.io [8]é uma ferramenta de modelagem de dados leve, baseada em scripts em uma linguagem de marcação própria para definição de entidades e relacionamentos chamada *Database Markup Language* (DBML). Sua proposta minimalista é voltada para desenvolvedores que buscam rapidez na criação de diagramas e exportação para SQL, na Figura 4 é possível ver a interface da plataforma.

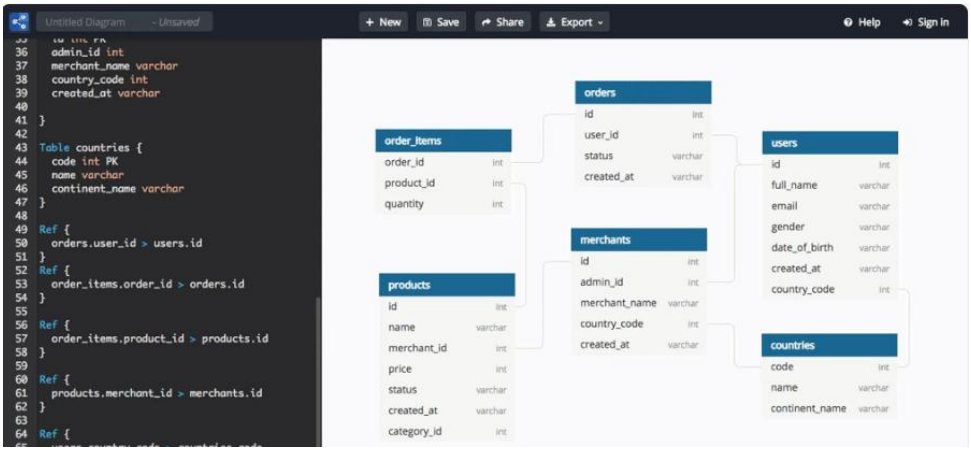

Figura 4. Página do site DBDiagram.io, no qual mostra a modelagem física e o script respectivo.

No entanto, a ausência de funcionalidades colaborativas em tempo real, controle de permissões e integração com bancos de dados reais no plano gratuito da ferramenta, limita seu uso em ambientes de desenvolvimento colaborativo. Ainda assim, é uma ferramenta muito útil em estágios iniciais de projeto ou para documentação rápida de modelos.

3. Desenvolvimento

Esta seção apresenta o desenvolvimento do projeto, organizado em quatro subseções principais. Na subseção 3.1 são detalhadas as tecnologias e ferramentas aplicadas no sistema. A subseção 3.2 expõe os requisitos funcionais e não funcionais do sistema, apresentando tanto a fundamentação teórica quanto as tabelas que resumem as funcionalidades e restrições do projeto, além de diagramas UML para ilustrar os principais casos de uso e fluxos do sistema. Em seguida, a subseção 3.3 aborda a modelagem do banco de dados, apresentando o diagrama entidade-relacionamento (ER) e explicando as principais relações de persistência de dados, sempre contextualizando essas decisões a requisitos funcionais e não funcionais relevantes. Por fim, a subseção 3.4 é dedicada ao projeto de interface, trazendo imagens das telas planejadas e explicações que evidenciam a relação entre o design das interfaces e os requisitos definidos.

3.1. Tecnologias e Ferramentas Aplicadas

A plataforma colaborativa em tempo real que será desenvolvida neste trabalho faz uso de um conjunto integrado de tecnologias modernas para atender aos requisitos de modelagem relacional

[8]Disponível em `https://dbdiagram.io/home`

de bancos de dados de forma eficiente e escalável. No *front-end*, serão utilizadas as ferramentas Angular, TypeScript[9] e JointJS[10]. No *back-end*, serão aplicados o FastAPI, a linguagem Python e um banco de dados composto por MongoDB[11] e Postgres[12]. A comunicação em tempo real entre os usuários será viabilizada pela biblioteca Socket.IO[13], garantindo sincronização imediata durante a colaboração.

O *framework*[14] Angular permite desenvolver aplicações web dinâmicas baseadas em componentes reutilizáveis e interfaces reativas, promovendo uma experiência fluida e organizada para o usuário.

A linguagem TypeScript adiciona tipagem estática ao JavaScript, trazendo recursos avançados que tornam o desenvolvimento mais seguro, escalável e de fácil manutenção, especialmente em projetos de médio e grande porte.

Com a biblioteca JointJS, é possível criar diagramas gráficos interativos, como fluxogramas e modelos relacionais, essenciais para representar visualmente estruturas de banco de dados de forma intuitiva e manipulável.

A comunicação em tempo real é viabilizada pela biblioteca Socket.IO, que utiliza WebSockets[15] para manter conexões persistentes, garantindo a sincronização instantânea das alterações realizadas de forma colaborativa.

No back-end, o FastAPI se destaca como um *framework* moderno, que facilita a criação de Application Programming Interface (API)[16] REST[17] de alto desempenho, com foco em velocidade, segurança e validação automática dos dados recebidos.

Para poder usar o *framework* FastAPI é necessário a utilização da linguagem Python, amplamente reconhecida por sua simplicidade e legibilidade, serve de base para o desenvolvimento do servidor, contribuindo para a produtividade e clareza do código.

Não obstante, para o armazenamento de dados relacionado aos *schemas* criados será usado o MongoDB que oferece uma abordagem NoSQL orientada a documentos, permitindo maior flexibilidade na estruturação das informações. Por outro lado, o armazenamento das demais relações do usuário, bem como autenticação e vincúlo entre *schemas* será persistido no banco relacional Postgres.

3.2. Requisitos Funcionais e Não Funcionais

Os requisitos de um sistema são divididos em funcionais e não funcionais. Os requisitos funcionais descrevem o que o sistema deve fazer, ou seja, as funcionalidades e comportamentos esperados a partir das interações dos usuários. Já os requisitos não funcionais definem restrições e qualidades

[9]Disponível em `https://www.typescriptlang.org/`

[10]Disponível em `https://www.jointjs.com/`

[11]Disponível em `https://www.mongodb.com/`

[12]Disponível em `https://www.postgresql.org`

[13]Disponível em `https://socket.io/pt-br/`

[14]Framework é uma estrutura de código pronta que fornece ferramentas, bibliotecas e padrões para facilitar o desenvolvimento de software.

[15]WebSockets são uma tecnologia que permite comunicação contínua e em tempo real entre o navegador e o servidor, sem a necessidade de recarregar a página ou fazer múltiplas requisições.

[16]API é o meio por onde o *front-end* consome e faz requisição para *back-end* para o consumo de dados.

[17]REST é um padrão que define um conjunto de regras e princípios para a criação de APIs web, pode se encontrar mais informações em `https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

que o sistema deve possuir, como desempenho, segurança, usabilidade, entre outros aspectos que garantem sua eficiência e robustez.

A Figura 5 representa possibilidades que um usuário tem ao usar a plataforma ColaBD, ele inicia o fluxo verificando se já possui cadastro no sistema. Caso não possua, ele é direcionado para realizar o cadastro; caso contrário, pode efetuar o login e acessar o sistema. Após o login, o usuário é levado ao dashboard, onde pode visualizar seus *schemas*. Se já possuir *schemas*, ele tem acesso a diversas ações, como editar o titulo e suas respectivas células[18], duplicar, excluir e convidar colaboradores para trabalharem com ele.

Caso não possua nenhum *schema*, o sistema oferece a opção de criar um novo. Além disso, o usuário pode gerenciar seus times, visualizando os times que já possui. Se houver times cadastrados, ele pode editar suas informações, excluir times existentes e adicionar colaboradores. Se ainda não possuir nenhum time, o sistema permite criar um novo. Por fim, o usuário pode encerrar sua sessão a qualquer momento realizando o logout.

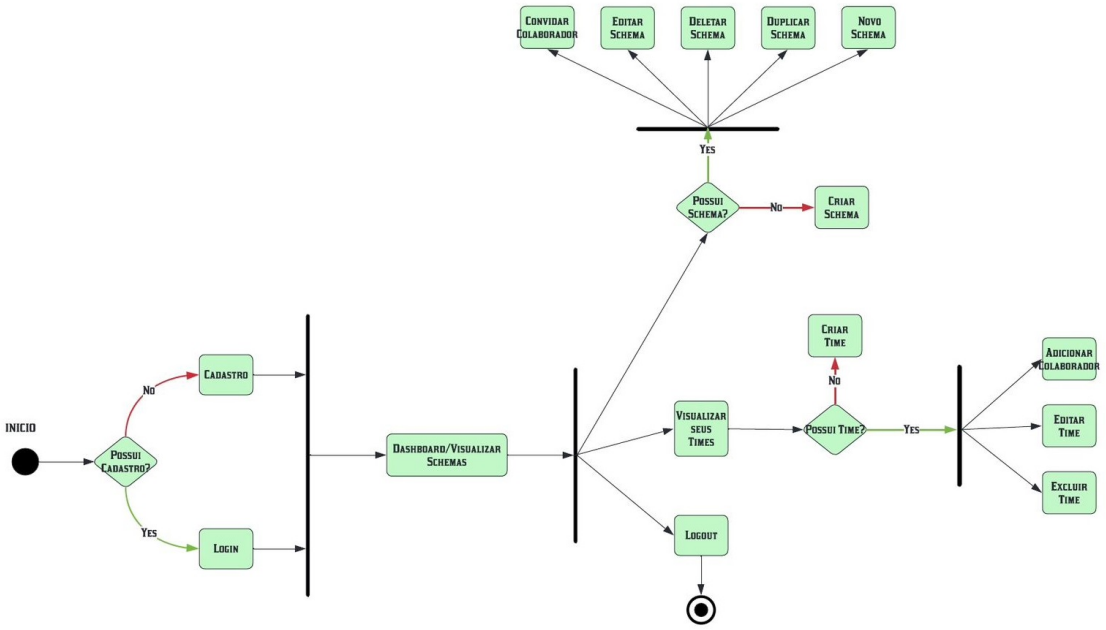

Figura 5. Diagrama UML de caso de uso da plataforma ColaBD

[18] A célula é um outro nome que pode ser usado para referenciar a tabela de um *schema*, as quais são usadas para fazer a modelagem do banco de dados.

Tabela 1. Requisitos Funcionais

ID	Descrição	Classificação
RF01	Cadastro de usuários com validação de nome, e-mail e senha.	Essencial
RF02	Autenticação de usuários via e-mail e senha utilizando *Json Web Token* (JWT)[19].	Essencial
RF03	Criação, edição e exclusão de modelos de banco com nome, SGBD e configuração inicial.	Essencial
RF04	Colaboração em tempo real entre usuários convidados com atualização instantânea via WebSocket.	Essencial
RF05	Manipulação de tabelas e relacionamentos no diagrama (atributos, chaves, restrições).	Essencial
RF06	Geração de script SQL conforme SGBD selecionado, com base em dicionário de mapeamento interno.	Desejável
RF07	Visualização de histórico de alterações com data, autor e tipo de modificação.	Importante
RF08	Salvar e carregar projetos na nuvem, com controle de acesso por proprietário e colaboradores.	Importante
RF09	Geração de relatórios e população de dados em bancos criados na plataforma, utilizando inteligência artificial.	Desejável
RF10	Gerenciamento de permissões e cargos entre usuários.	Importante

Tabela 2. Requisitos Não Funcionais

ID	Descrição	Categoria
RNF01	A plataforma deve ser compatível com navegadores modernos (HTML5).	Compatibilidade
RNF02	O sistema deve utilizar autenticação segura baseada em tokens JWT, com validação em todas as rotas protegidas.	Segurança
RNF03	O sistema deve implementar controle de acesso a projetos, permitindo apenas ao proprietário e colaboradores autorizados visualizar ou editar os modelos.	Segurança
RNF04	O sistema deve integrar uma IA para transformar a modelagem relacional em scripts SQL adequados ao SGBD escolhido.	Integração
RNF05	O sistema deve garantir colaboração em tempo real sem sobrescrita de dados, implementando filas e mecanismos de lock para evitar conflitos em operações simultâneas.	Colaboração

A interação entre o usuário e os casos de uso reflete os principais comportamentos esperados da plataforma, como autenticação, colaboração e manipulação de esquemas. Esses comportamentos foram detalhados e organizados no Quadro 1, que apresenta os requisitos funcionais.

3.3. Modelagem Banco de Dados

A modelagem do banco de dados proposta para a plataforma visa atender aos diferentes requisitos de funcionalidade, escalabilidade e flexibilidade do sistema. Para isso, optou-se pela utilização combinada de um banco de dados relacional e um banco de dados não relacional. O banco relacional foi escolhido para gerenciar autenticação, controle de acesso e o esquema de permissões dos usuários, proporcionando robustez, segurança e integridade referencial nesses processos críticos do sistema. Já o banco de dados não relacional é utilizado para armazenar entidades dinâmicas, tabelas e *schemas* criados colaborativamente pelos próprios usuários da plataforma.

Essa abordagem encontra respaldo conforme Bjeladinovic et al. (2020), que afirmam que a combinação de bancos relacionais e não relacionais é particularmente vantajosa em sistemas modernos que lidam com diferentes tipos de dados, estruturados e semi-estruturados. Onde a integração de componentes híbridos *SQL/NoSQL* permite a representação eficiente de múltiplos formatos de dados em uma mesma arquitetura, adaptando-se às necessidades específicas de cada parte do sistema.

Assim, a combinação dos dois modelos de banco de dados garante tanto a segurança e confiabilidade nas operações de autenticação e permissões quanto a escalabilidade e maleabilidade necessárias para o armazenamento e manipulação das criações dos usuários.

O modelo relacional definido no Supabase [20] organiza as principais entidades do sistema em tabelas interligadas, utilizando chaves primárias e estrangeiras para assegurar a integridade referencial. A tabela `user` armazena as informações básicas de cada usuário, como nome, e-mail, senha e a data de inserção, sendo identificada pela chave primária *id*.

A escolha pelo uso do Supabase se deu principalmente pela sua facilidade de configuração e integração com a plataforma desenvolvida. Por ser uma plataforma de *back-end* como serviço (BaaS), com o Supabase, apesar de possuir outros diversos recursos, foi possível ter uma interface intuitiva para o gerenciamento de bancos de dados PostgreSQL e armazenamento de arquivos, com isso foi possível reduzir significativamente o tempo de desenvolvimento e a complexidade da infraestrutura.

A tabela `team` representa os times criados pelos usuários, contendo atributos como nome, `owner_user_id` (referenciando o usuário proprietário do time) e data de inserção. A relação entre usuários e times é do tipo muitos-para-muitos e é gerenciada pela tabela intermediária `user_team`, que conecta `user_id` e `team_id`.

De forma semelhante, a associação entre usuários e esquemas também segue uma relação muitos-para-muitos, registrada na tabela `user_schema`, que vincula `user_id` e `schema_id`, desse modo fazendo com que a modelagem permita múltiplos usuários e que eles tenham acesso ao mesmo esquema, promovendo colaboração.

A tabela `schema` concentra os dados dos modelos criados na plataforma, armazenando informações como `title`, `display_picture`, `database_model`, além de timestamps de inserção e atualização. O campo `owner_user_id`, presente em `team`, e a ligação com `user_schema`, asseguram que cada esquema possa ser associado tanto ao usuário que o criou quanto aos membros que possuem acesso a ele, esse conjunto de tabelas garante flexibilidade para representar a colaboração entre usuários e equipes, ao mesmo tempo em que preserva a consistência e a rastreabilidade das informações dentro da plataforma.

Com a proposta de oferecer maior flexibilidade e organização no armazenamento dos modelos de banco de dados, cada tabela criada no editor visual é representada como um objeto

[20] Disponível em `https://supabase.com/`

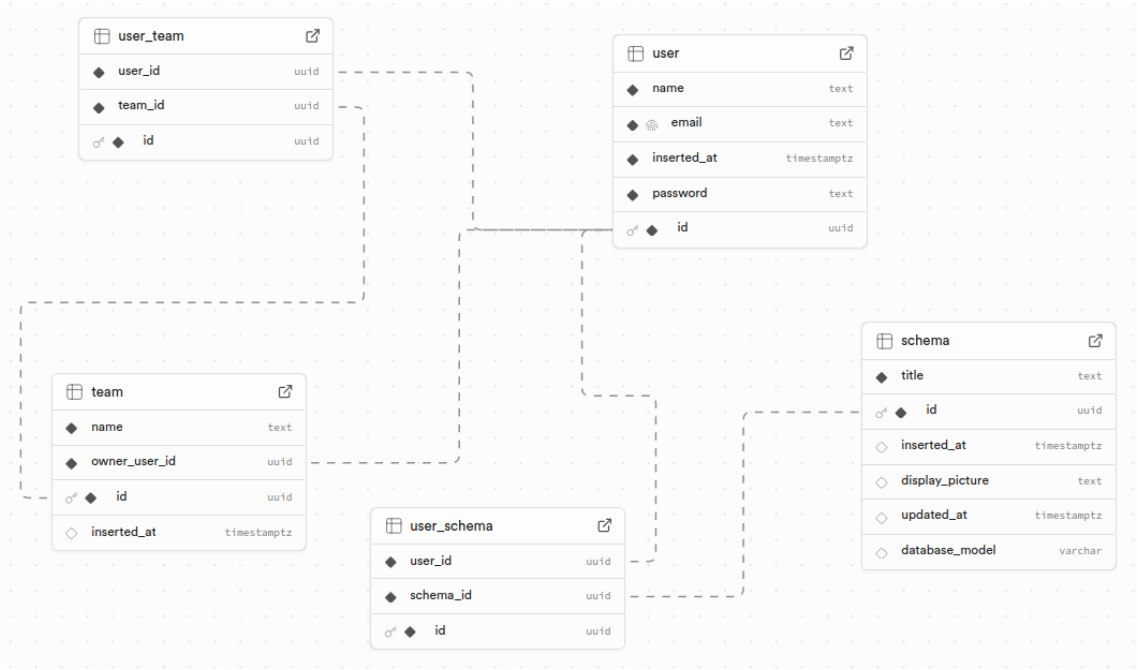

Figura 6. Mapeamento do Banco Relacional


```
_id: ObjectId('6856ff396bdc1a86959f04aa')
cells : Array (3)
  0: Object
     type : "standard.Rectangle"
     id : "id-da-tabela-usuario"
    attrs : Object
      label : Object
      row0-name : Object
         text : "id"
      row0-type : Object
         text : "INT"
      row0-meta : Object
         pk : true
         fk : false
     sql : "CREATE TABLE Usuario (id INT PRIMARY KEY);"
  1: Object
     type : "standard.Rectangle"
     id : "id-da-tabela-perfil"
    attrs : Object
      label : Object
     sql : "CREATE TABLE Perfil (...);"
  2: Object
     type : "standard.Link"
    source : Object
       id : "id-da-tabela-usuario"
    target : Object
       id : "id-da-tabela-perfil"
    attrs : Object
      label : Object
         text : "FK_usuario_perfil"
```


Figura 7. Mapeamento Coleção Models

individual dentro do *array cells*, contendo tanto suas propriedades estruturais quanto um campo exclusivo sql que armazena a *query*[21] de criação daquela tabela de forma independente.

As informações de chave primária e chave estrangeira são salvas em propriedades customizadas associadas às colunas de cada tabela, permitindo identificar facilmente as restrições do modelo relacional.

Já os relacionamentos entre tabelas são registrados como elementos do tipo Link, que conectam os identificadores das tabelas envolvidas, também presentes no mesmo *array*. Essa estrutura centraliza, em um único documento no MongoDB, todos os dados necessários para restaurar o diagrama visual, exportar o SQL de cada tabela separadamente e manter o controle detalhado das regras e vínculos do banco de dados modelado na plataforma.

O JointJS representa cada tabela do banco como um objeto JSON contendo informações sobre posição, tamanho, identidade única e um conjunto de atributos visuais (como nome da tabela e colunas). Isso permite salvar e restaurar todo o diagrama facilmente, além de facilitar a exportação para formatos visuais como SVG[22].

3.4. Projeto de Interface

Nesta subseção será exibido o protótipo de interfaces desenvolvido por meio da plataforma (Figma, 2025). Deve-se levar em consideração que as telas desenvolvidas nessa plataforma são apenas protótipos e ao final do trabalho possa ser que esteja com alguns aspectos diferentes.

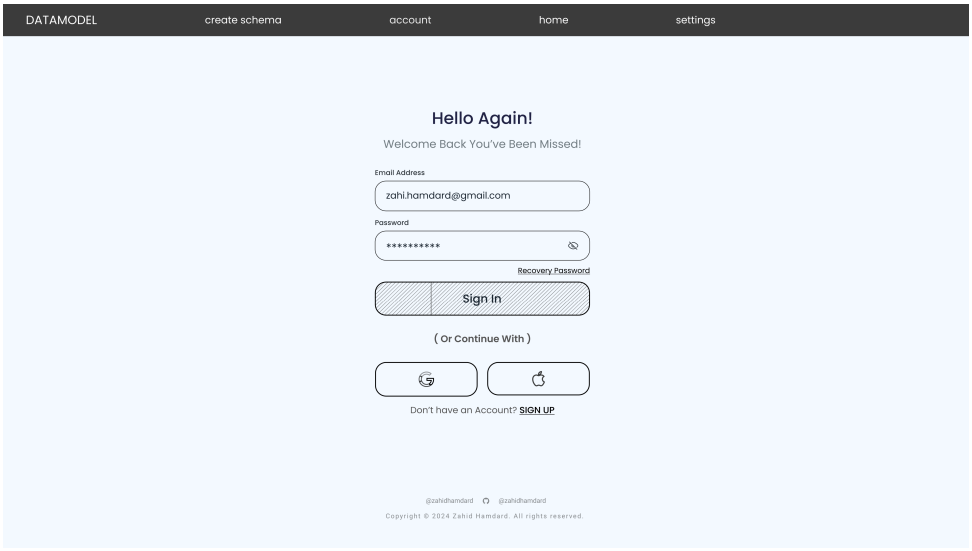

Figura 8. Página de *login* da plataforma ColaBD.

A tela representada na Figura 8 mostra a página de login da plataforma ColaBD, a qual possui dois campos obrigatórios para a autenticação no sistema que é o campo de e-mail e senha. Caso não possua *login*, pode clicar no *link* do texto *SING UP* e assim será redirecionado para a pagina de se cadastrar na pltaforma. Os botões com o logo da Google LLC (2024) e Apple Inc. (2024) são para se autenticar no sistema com as respectivas contas.

A tela de cadastro foi projetada para viabilizar o requisito RF01, permitindo que novos usuários efetuem o registro a partir da validação de nome, e-mail e senha, garantindo um fluxo

[21] Query refere-se a uma instrução formal utilizada para manipular dados em banco de dados.

[22] SVG (Scalable Vector Graphics) é um formato de imagem vetorial baseado em XML, que permite redimensionamento sem perda de qualidade e é amplamente usado em aplicações web.

seguro e controlado de novos acessos à plataforma. De forma complementar, a tela de login atende ao requisito RF02, proporcionando a autenticação de usuários já cadastrados mediante a inserção de e-mail e senha, fluxo fundamental para a proteção das informações e o controle de sessões. Ambas as interfaces também oferecem opções de autenticação via provedores externos, ampliando a flexibilidade e facilitando o acesso do usuário ao sistema.

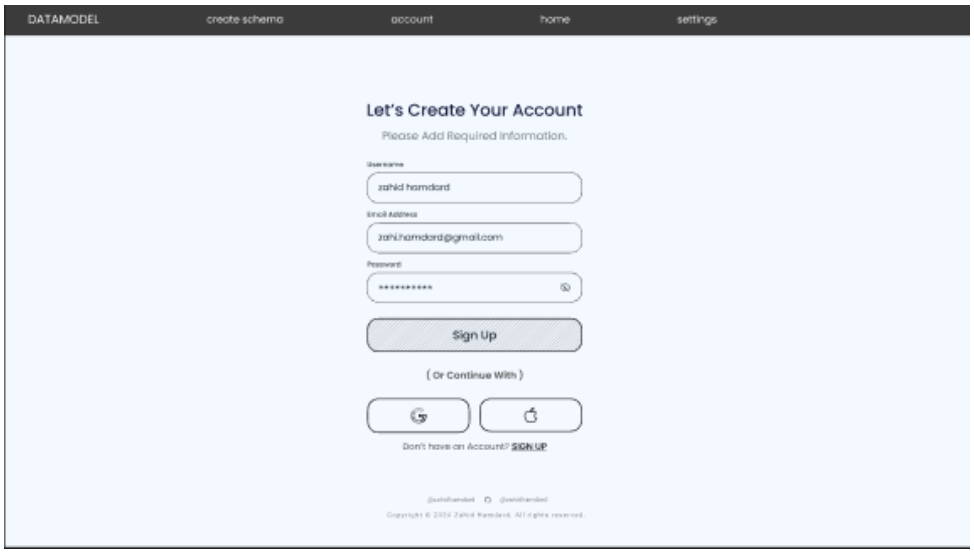

Figura 9. Página de cadastro da plataforma ColaBD.

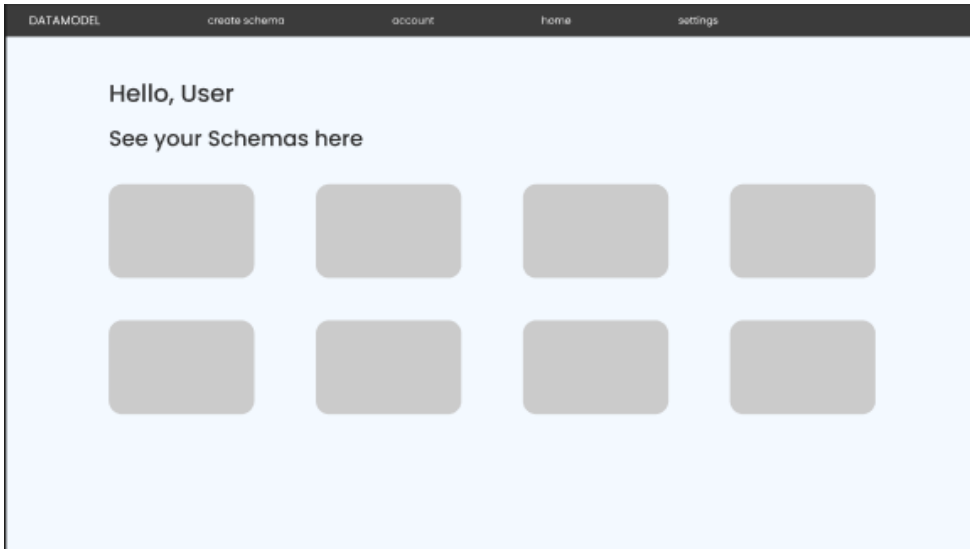

Figura 10. Página principal da plataforma ColaBD

A imagem da Figura 10 mostra a página principal da plataforma contendo uma saudação *Hello, User*, onde *User* seria o nome do usuário que estivesse logado seguida da frase *See your Schemas here*. Abaixo dessas mensagens, há caixas retangulares que representam visualmente os esquemas ou modelos com os quais o usuário trabalhou.

Além disso, esta tela proporciona ao usuário uma visão organizada de seus projetos e esquemas já criados, permitindo acesso facilitado e visualização rápida dos modelos armazenados,

em consonância com o requisito RF03, que prevê a criação, edição e exclusão de modelos de banco, bem como sua configuração inicial.

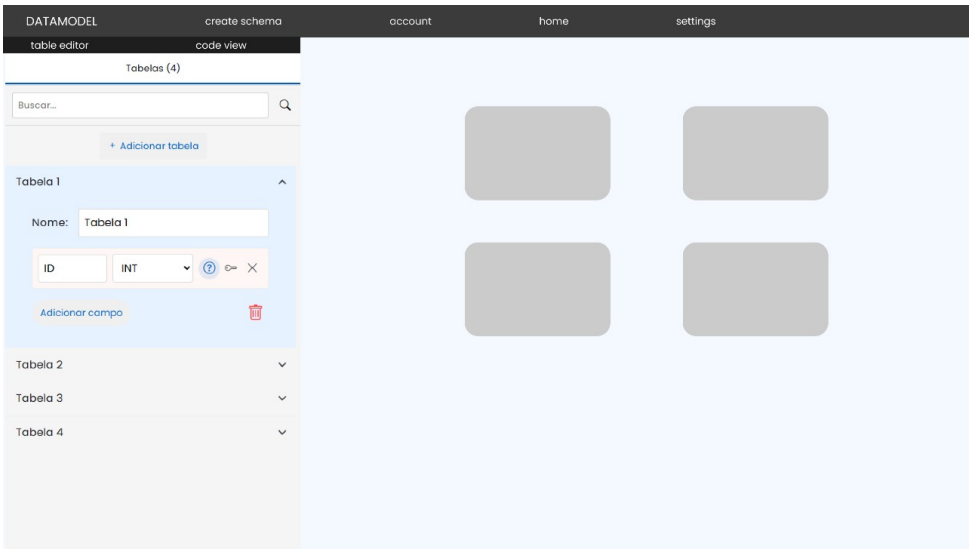

Figura 11. Página de modelagem do banco de dados relacional

A Figura 11 exibe uma interface de modelagem de banco de dados, dividida em duas seções principais. À esquerda, em um painel vertical, encontra-se o editor de tabela, onde será possível cria-las e definir seus atributos e tipos, contudo também será possível ver cada tabela em detalhe, mostrando seu nome e suas respectivas colunas, podendo remover, editar ou definir como *Primary Key*[23] ou *nullables*[24], conforme especificado no requisito RF05. A direita, um grande painel apresenta uma área visual para a modelagem, onde são exibidas as tabelas e suas respectivas ligações.

Não obstante, a funcionalidade de geração automática de scripts SQL a partir do modelo criado atende ao requisito RF06, proporcionando ao usuário a exportação do esquema para diferentes SGBDs conforme o dicionário de mapeamento interno da plataforma. A interface também pode contemplar o acompanhamento do histórico de alterações nos *schemas*, associando cada modificação ao usuário responsável, conforme previsto no requisito RF07, o que contribui para a rastreabilidade e o controle colaborativo dos projetos desenvolvidos.

3.5. Implementações

No *back-end*, desenvolvido com FastAPI, concentram-se as regras de negócio e o gerenciamento das interações com o banco de dados fornecido pelo Supabase. Nessa camada, foram implementadas rotas para atender operações como *GET*, *POST*, *PUT* e *DELETE* para a parte de manipulação de *schemas* e apenas rotas *POST* para a parte de autenticação, assim garantindo que os dados dos modelos sejam criados, consultados, modificados ou excluídos de forma consistente.

[23] *Primary Keys* ou chaves primárias são usadas para serem identificadores de uma tabela, normalmente sendo o id de um objeto

[24] *Nullables* referem-se a colunas que aceitam valores nulos, indicando que determinado dado pode estar ausente ou indefinido em um registro.

Figura 12. Página do Swagger mostrando os *endpoints* (rotas) do Colabd.

Na Figura 12 é apresentada a documentação da API desenvolvida, gerada com o auxílio da ferramenta Swagger[25]. Essa ferramenta permite visualizar de forma estruturada as rotas disponíveis no sistema, bem como analisar os retornos de cada requisição, incluindo mensagens de erro e de sucesso, o que facilita a integração com o *front-end*.

O *front-end*, implementado em Angular, é responsável pela interface e experiência do usuário. Para possibilitar a edição visual e interativa dos diagramas, integrou-se a biblioteca JointJS, que viabiliza a manipulação gráfica de tabelas, atributos e relacionamentos. Essa camada mantém comunicação constante com o *back-end*, assegurando a atualização em tempo real das informações entre os usuários conectados.

No gerenciamento do processo de desenvolvimento, utilizou-se o GitHub como repositório de código e ferramenta de controle de versão, adotando uma estratégia de ramificações para separar o código estável daquele em desenvolvimento. O acompanhamento das tarefas foi realizado por meio do Trello[26], enquanto o Figma[27] foi empregado para a elaboração de protótipos e fluxos de interface, permitindo validar previamente as soluções visuais antes da implementação.

As etapas de codificação foram conduzidas em ambientes integrados de desenvolvimento (IDEs), como Visual Studio Code[28] e PyCharm[29], que proporcionaram agilidade e integração com as tecnologias utilizadas. Por fim, o sistema foi implantado na plataforma Render[30], responsável pela hospedagem tanto do *front-end* quanto do *back-end*. Para garantir maior disponibilidade e desempenho dos serviços, optou-se pela utilização do plano Starter oferecido pela plataforma.

Além disso, para organizar o desenvolvimento e evidenciar a evolução das funcionalidades implementadas, a plataforma foi estruturada em versões incrementais, permitindo uma evolução controlada e mensurável de cada módulo do sistema. Essa estratégia de versionamento foi essencial para validar gradualmente as decisões técnicas e arquiteturais, reduzindo riscos de

[25]Disponível em `https://swagger.io/`
[26]Disponível em `https://trello.com/`
[27]Disponível em `https://figma.com/`
[28]Disponível em `https://code.visualstudio.com/`
[29]Disponível em `https://www.jetbrains.com/pycharm/`
[30]Disponível em `https://render.com/`

implementação e facilitando a integração contínua entre as diferentes camadas da aplicação.

A primeira versão constituiu-se como uma aplicação voltada à criação de *schemas* de forma totalmente *offline*, focada na validação da base técnica da modelagem visual por meio da biblioteca JointJS. Nessa etapa inicial, buscou-se comprovar a viabilidade da representação gráfica de tabelas, atributos e relacionamentos.

A segunda versão introduziu o conceito de colaboração, ainda que de forma offline, permitindo que múltiplos usuários compartilhassem e editassem *schemas* em um mesmo ambiente de trabalho, mas sem atualização em tempo real. Essa iteração teve como principal objetivo validar o fluxo colaborativo e a integridade dos dados manipulados de forma simultânea, preparando a arquitetura para suportar comunicação assíncrona entre clientes e o servidor.

Por fim, a terceira versão implementou a colaboração em tempo real. Nessa fase, a comunicação entre os usuários passou a ser mediada por canais de WebSocket, utilizando a biblioteca *Socket.IO* para garantir sincronização instantânea das alterações realizadas no editor visual. Com isso, tornou-se possível observar em tempo real as ações dos participantes conectados, como movimentação de tabelas, criação de relacionamentos e edição de atributos.

3.5.1. Arquitetura

A arquitetura da plataforma foi concebida com o objetivo de oferecer uma solução escalável e organizada para o mapeamento relacional de bancos de dados. O sistema adota uma arquitetura em camadas, baseada no modelo cliente-servidor, composta por três componentes principais: *front-end*, *back-end* e persistência de dados, além da integração com serviços externos de inteligência artificial.

A Figura 13 apresenta o Diagrama de Contexto da plataforma, elaborado conforme o modelo *C4*[31]. Esse diagrama ilustra, de forma macro, como o sistema ColaBD interage com seus usuários e com os serviços externos utilizados.

A partir desse modelo, é possível compreender os principais elementos que compõem a solução:

- Usuário: acessa a plataforma por meio de um navegador web, podendo criar, editar e visualizar modelos relacionais de banco de dados de forma colaborativa.
- ColaBD: sistema central que fornece a interface interativa, as APIs e os serviços de colaboração em tempo real. É responsável por gerenciar a comunicação entre os usuários e os bancos de dados.
- Supabase: banco de dados relacional que armazena informações de autenticação, equipes, schemas e permissões de acesso.
- MongoDB: banco de dados não relacional utilizado para armazenar os dados estruturais e dinâmicos das tabelas e relacionamentos criados pelos usuários no editor visual.
- Groq: serviço externo de inteligência artificial utilizado para auxiliar na geração e tradução de scripts SQL conforme o Sistema Gerenciador de Banco de Dados (SGBD) selecionado.

[31] O modelo C4, proposto por Simon Brown, é uma abordagem de visualização arquitetural que organiza a descrição de sistemas em quatro níveis: Contexto, Contêiner, Componente e Código. Ele permite representar, de forma hierárquica, desde a visão geral das interações do sistema até os detalhes de implementação.

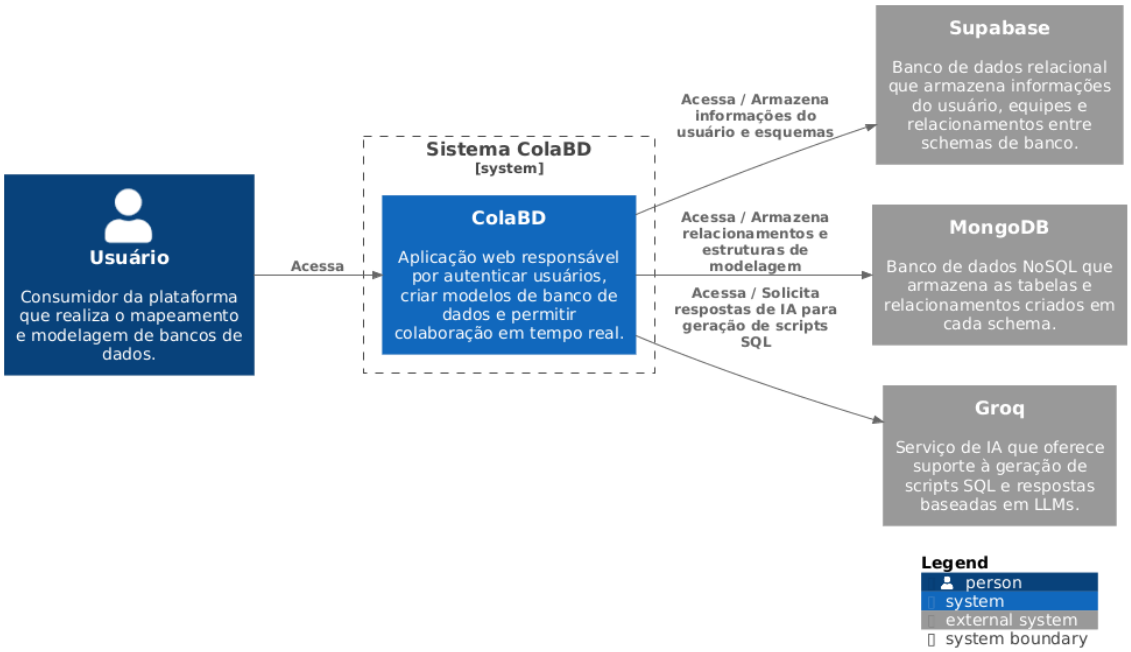

Figura 13. Diagrama de Contexto da plataforma *ColaBD*.

A Figura 14 apresenta o Diagrama de Contêiner da plataforma *ColaBD*, desenvolvido também segundo o modelo *C4*. Esse diagrama detalha a decomposição do sistema em seus principais contêineres de software e bases de dados, destacando suas responsabilidades e interações. A arquitetura adota o padrão cliente-servidor, composta por uma aplicação de interface (*front-end*), um serviço de regras de negócio (*back-end*) e uma camada de persistência de dados, além da integração com um serviço externo de IA.

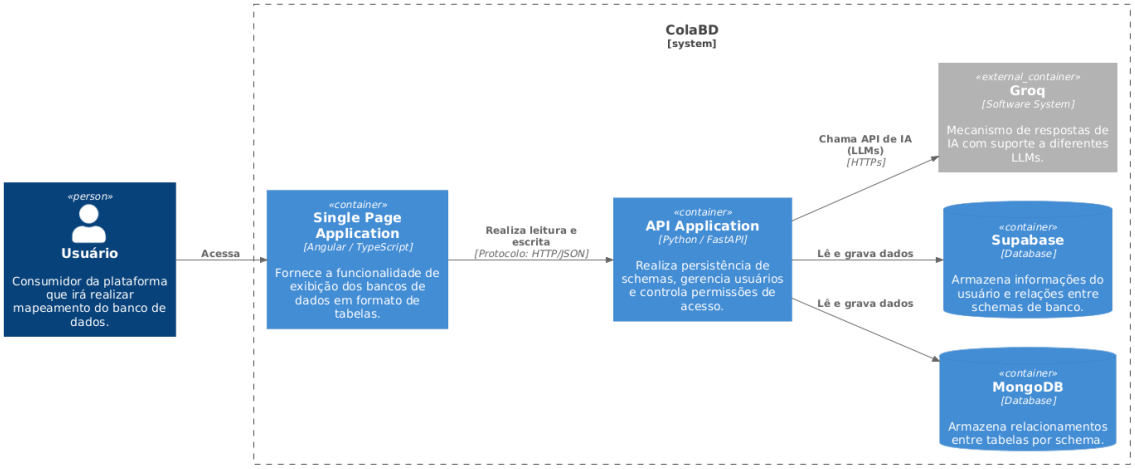

Figura 14. Diagrama de Contêiner da Plataforma ColaBD.

O *front-end* é representado pela *Single Page Application (SPA)*, desenvolvida em Angular e TypeScript, responsável por oferecer uma interface interativa e responsiva para exibição e manipulação dos modelos relacionais. Essa aplicação se comunica com a *API Application*, imple-

mentada em *Python* com o framework *FastAPI*, que centraliza as regras de negócio, autenticação, controle de usuários e coordenação das funcionalidades colaborativas em tempo real.

Na camada de dados, o sistema integra dois bancos com propósitos distintos: o *Supabase*, utilizado para armazenar informações relacionais como usuários, equipes e esquemas de banco; e o MongoDB, responsável pelo armazenamento não relacional das tabelas e relacionamentos criados no editor visual.

Por fim, o sistema realiza comunicação com o serviço externo Groq, que fornece suporte a modelos de linguagem (*LLMs*) para auxiliar na geração, tradução e padronização de scripts SQL conforme o Sistema Gerenciador de Banco de Dados (SGBD) selecionado pelo usuário.

3.5.2. Implementações de Segurança

A segurança da plataforma ColaBD foi concebida de forma integrada entre *front-end* e *back-end*, em conformidade com os requisitos funcionais RF02 e RF10, além dos requisitos não funcionais RNF02 e RNF03. No *front-end*, adotou-se uma arquitetura de autenticação baseada em tokens, atendendo diretamente ao RF02 e ao RNF02.

No aspecto de autorização (RF10 e RNF03), foram configurados *guards* nas rotas do Angular, que verificam autenticação e escopos antes de permitir o carregamento de módulos sensíveis. O *Cross-Origin Resource Sharing* (CORS)[32] foi limitado apenas a domínios estritamente necessários, em atendimento ao RNF03.

No *back-end*, a autenticação com JWT (RF02, RNF02) foi implementada conforme ilustrado na implementação 1. O token é gerado no momento do login, assinado com algoritmo HS256 a partir de chave secreta definida em variáveis de ambiente, contendo apenas informações essenciais como *id* e *email*, e possui expiração configurada. Esse mesmo token é utilizado em requisições HTTP protegidas.

```
def create_access_token(data: dict):
    expire = datetime.now(timezone.utc) +
        timedelta(minutes=EXPIRATION_MINUTES)
    to_encode = {"id": data["id"], "email": data["email"]}
    jwt_token = jwt.encode(to_encode, os.getenv("SECRET_KEY"),
        algorithm=ALGORITHM)
    return Response(data=InfoAuth(access_token=jwt_token, exp=expire,
        token_type="Bearer"), success=True)
```

Implementação 1. Função de criação de JWT no back-end

As senhas dos usuários são armazenadas de forma segura por meio de técnicas de *hashing* com o algoritmo `bcrypt`, via biblioteca `passlib`, evitando o armazenamento em texto puro e dificultando ataques de força bruta conforme Implementação 2. A verificação ocorre durante o login, comparando o hash armazenado com a senha fornecida pelo usuário (Implementação 3).

```
def gerar_hash_senha(password: str) -> str:
    return pwd_context.hash(password)
```

Implementação 2. Função de geração de hash de senha

[32]CORS é um mecanismo de segurança que faz com que os navegadores permitam que uma aplicação web acesse recursos ou tenha seus recursos acessados de uma origem ou destino diferente do seu próprio

```python
def verificar_hash_senha(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)
```

Implementação 3. Função de verificação de senha durante o login

Além da autenticação, foram adotadas práticas adicionais no servidor para atender ao RNF03, tais como:

- Uso de variáveis de ambiente para gerenciar segredos (como chaves JWT e credenciais de banco), evitando exposição no código-fonte;
- Armazenamento seguro de arquivos no Supabase, acessíveis somente por meio de URLs assinadas com tempo de expiração;
- Padronização de mensagens de erro, assegurando clareza ao usuário sem revelar detalhes internos da aplicação;

3.5.3. Esquemas e Aspectos Colaborativos

A implementação do recurso de *schemas* representa um dos núcleos funcionais da plataforma ColaBD, atendendo diretamente aos requisitos funcionais RF03 (criação, edição e exclusão de modelos de banco), RF04 (colaboração entre usuários), RF05 (manipulação de tabelas e relacionamentos no diagrama) e RF08 (armazenamento e carregamento de projetos na nuvem com controle de acesso). Do ponto de vista não funcional, este módulo também está alinhado ao RNF05, que exige mecanismos de colaboração sem sobrescrita de dados e com controle de concorrência adequado.

O *back-end* expõe um conjunto de endpoints RESTful no controlador `controller_schema.py`, os quais permitem criar, consultar, atualizar e excluir *schemas*. Todas as operações são autenticadas e dependem do identificador do usuário obtido a partir do token JWT, garantindo que somente proprietários ou colaboradores autorizados possam acessar ou modificar os modelos (requisito RF08).

A criação de um *schema* associa automaticamente o criador como proprietário, atendendo ao RF03. O código na Implementação 4 ilustra o endpoint de criação, no qual a linha 2 recebe o id do usuário na variável *current_user_id*, a partir da captação do JWT que veio nos dados da requisição e na linha 3 é persistido no banco o novo *schema*, onde é passado por parâmetro as variáveis *current_user_id* e *schema_data* na função *create_schema*:

```python
@router.post("/", response_model=Response)
async def create_schema(schema_data: CreateSchema, current_user_id:
    str = Depends(get_current_user_id)):
    result = await service_schema.create_schema(schema_data,
        current_user_id)
    if not result.success:
        http_exception(result, 400)
    return Response(data=result.data, success=True)
```

Implementação 4. Criação de *schema* no back-end

A consulta detalhada na Implementação 5, retorna tanto os metadados quanto as tabelas modeladas do respectivo schema persistidas no MongoDB, que representam a estrutura do diagrama modelado pelo usuário:

```python
@router.get("/{schema_id}", response_model=Response)
async def get_schema_by_id(schema_id: str, current_user_id: str =
    Depends(get_current_user_id)):
    result = await service_schema.get_schema_with_cells(schema_id,
        current_user_id)
    if not result.success:
        http_exception(result, 404)
    return Response(data=result.data, success=True)
```

Implementação 5. Consulta detalhada de *schema*

```python
    async def get_schema_with_cells(self, schema_id: str,
        current_user_id: str) -> Response:
        schema_result = await
            self.repo_schema.get_schema_by_id(schema_id)
        if not schema_result.success:
            return Response(data="Schema nao encontrado",
                success=False)

        schema_data = schema_result.data

        user_schemas_result = await
            self.repo_schema.get_by_user_id(current_user_id)

        user_schema_ids = [us["schema_id"] for us in
            user_schemas_result.data]
        if schema_id not in user_schema_ids:
            return Response(data="Acesso negado: voce nao tem
                permissao para acessar este schema", success=False)

        cells_data = None
        if schema_data.get("database_model"):
            cells_result = await self.repo_cells.get_cells_by_id
            (schema_data["database_model"])
            if cells_result.success:
                cells_data = cells_result.data

        response_data = {
            "schema": schema_data,
            "cells": cells_data.get("cells", []) if cells_data else [],
            "has_cells": cells_data is not None,
            "database_model_id": schema_data.get("database_model")
        }

        return Response(data=response_data, success=True)
```

Implementação 6. Recuperando do banco de dados o *schema* e suas respectivas tabelas

A função *get_schema_with_cells* na implementação 6 tem como objetivo buscar um schema específico no banco de dados juntamente com suas tabelas associadas, garantindo que o usuário tenha permissão de acesso. Inicialmente, ela consulta o repositório para obter o schema pelo seu identificador `schema_id`. Em seguida, verifica se o usuário atual `current_user_id` possui acesso a esse schema, comparando-o com a lista de schemas vinculados ao usuário.

Caso o schema não pertença ao usuário, a função retorna uma resposta de acesso negado.

Se o schema possuir um modelo de banco de dados associado `database_model`, a função busca também as tabelas relacionadas a esse modelo. Por fim, ela monta uma resposta contendo os dados do schema, as tabelas (quando existentes), um indicador de presença de tabelas e o identificador do modelo de banco de dados. Em caso de qualquer erro durante o processo, a função captura a exceção e retorna uma resposta informando a falha.

Conforme a Implementação 7 a atualização permite modificar tanto as células (linha 4), quanto os metadados (linha 7), assim recebendo um `multipart form` que pode conter opcionalmente uma miniatura (`display_picture`) para enriquecer a visualização nas listagens.

```
1 @router.put("/", response_model=Response)
2 async def update_schema(
3     schema_id: str = Form(...),
4     cells: str = Form(...),
5     display_picture: Optional[UploadFile] = File(None),
6     current_user_id: str = Depends(get_current_user_id)):
7     update_data = UpdateSchemaData(schema_id, json.loads(cells))
8     result = await service_schema.update_schema(update_data,
          current_user_id, display_picture)
```

Implementação 7. Atualização de schema com células e miniatura

Outras operações, conforme implementação 8, complementam o CRUD, como alteração de título (linhas 1 a 6) e exclusão (linhas 8 a 13).

```
1  @router.put("/title/{schema_id}", response_model=Response)
2  async def update_schema_title(schema_id: str, schema_title:
       UpdateSchemaTitle, current_user_id: str =
       Depends(get_current_user_id)):
3      result = await service_schema.update_schema_title(schema_id,
           schema_title.new_title, current_user_id)
4      if not result.success:
5          http_exception(result, 500)
6      return Response(data=result.data, success=True)
7
8  @router.delete("/{schema_id}", response_model=Response)
9  async def delete_schema(schema_id: str, current_user_id: str =
       Depends(get_current_user_id)):
10     result = await service_schema.delete_schema(schema_id,
           current_user_id)
11     if not result.success:
12         http_exception(result, 404)
13     return Response(data=result.data, success=True)
```

Implementação 8. Atualização de título e exclusão de schema

Na camada de serviço (`service_schema.py`), cada operação passa por verificações de vínculo de usuário ao schema, garantindo conformidade com o RF10 e com o RNF03 (controle de acesso). Um usuário somente pode modificar um schema se estiver vinculado a ele, como proprietário ou colaborador.

Na implementação 9 um exemplo simplificado do fluxo da atualização. Sendo que nas linhas 4 a 7 é validada a existência do *schema* e seu vínculo com o usuário. Apenas então nas linhas 8 a 11 ocorre a persistência das células no MongoDB, cujo identificador é armazenado como ponteiro no schema.

```
async def update_schema(self, update_schema_data, current_user_id:
    str, display_picture=None) -> Response:
    schema_id = update_schema_data.schema_id
    schema_result = await self.repo_schema.get_schema_by_id(schema_id)
    user_schemas_result = await
        self.repo_schema.get_by_user_id(current_user_id)
    user_schema_ids = [us["schema_id"] for us in
        user_schemas_result.data]
    if schema_id not in user_schema_ids:
        return Response(data="Acesso negado", success=False)
    cells_result = await self.repo_cells.create_cells({"cells":
        update_schema_data.cells})
    database_model_id = cells_result.data
    update_result = await
        self.repo_schema.update_schema_database_model(schema_id,
        database_model_id)
    return Response(data={"schema_id": schema_id, "database_model":
        database_model_id}, success=True)
```

Implementação 9. Regras de autorização na atualização de schema

A colaboração em sua primeira versão foi implementada de forma assíncrona, por meio de convites via e-mail. Essa abordagem atende ao RF04, permitindo que múltiplos usuários trabalhem sobre um mesmo projeto, ainda que sem edição simultânea em tempo real.

O *back-end* oferece um endpoint de `PATCH` para vincular colaboradores a um schema existente, na Implementação 10 o vinculo ocorre com a identificação do schema com o e-mail e identificado do usuário (linhas 3 a 7).

```
@router.patch("/", response_model=Response)
async def vinculate_schema(schema_data: VinculateSchema,
    current_user_id: str = Depends(get_current_user_id)):
    result = await service_schema.vinculate_user_to_schema(
        schema_data.schema_id,
        schema_data.user_email,
        current_user_id,
    )
    if not result.success:
        http_exception(result, 400)
    return Response(data=result.data, success=True)
```

Implementação 10. Vinculação de colaborador a um schema

A lógica correspondente no serviço verifica se o usuário que realiza a vinculação possui acesso ao schema, busca o novo colaborador pelo e-mail informado e cria o vínculo, conforme a implementação 11.

```
async def vinculate_user_to_schema(self, schema_id: str, user_email:
    str, current_user_id: str) -> Response:
    user_schemas_result = await
        self.repo_schema.get_by_user_id(current_user_id)
    if schema_id not in [us["schema_id"] for us in
        user_schemas_result.data]:
        return Response(data="Acesso negado", success=False)
    user_result = await self.repo_user.selectOne(type("UserEmailData",
        (), {"email": user_email}))
    user_id = user_result.data["id"]
```

```
    create_dict = {"id": str(uuid.uuid4()), "user_id": user_id,
        "schema_id": schema_id}
    create_result = await
        self.repo_schema.create_user_schema(create_dict)
    return Response(data={"message": "Usuario vinculado com sucesso",
        "user_id": user_id}, success=True)
```

Implementação 11. Serviço de vinculação de usuário a schema

Enquanto na implementação 12, a orquestração da interface inclui um modal de compartilhamento, que valida o e-mail e dispara o evento de vinculação.

```
export class ShareModalComponent {
  @Input() isVisible = false;
  @Input() schemaTitle = '';
  @Output() close = new EventEmitter<void>();
  @Output() shareWithUser = new EventEmitter<string>();

  emailToShare = '';
  isSharing = false;

  onShare() {
    if (this.emailToShare.trim() &&
        this.isValidEmail(this.emailToShare)) {
      this.isSharing = true;
      this.shareWithUser.emit(this.emailToShare.trim());
    }
  }
}
```

Implementação 12. Modal de compartilhamento no Angular

Além do CRUD e do compartilhamento assíncrono, conforme na implementação 13, o *back-end* implementa a exportação de modelos para SQL. Esses arquivos são armazenados em buckets[33] privados no Supabase e distribuídos por meio de URLs assinadas com tempo de expiração. Esse recurso garante que colaboradores possam acessar scripts exportados sem comprometer a segurança, alinhando-se tanto ao RF06 (exportação de SQL) quanto ao RNF03 (acesso controlado).

```
signed = storage_bucket.create_signed_url(path=file_path,
    expires_in=3600)
signed_url = signed.get("signedURL") or signed.get("signed_url") or
    signed.get("url")
if not signed_url:
    raise HTTPException(status_code=500, detail="Nao foi possivel
        obter signed URL")
```

Implementação 13. Exportação com URL assinada no Supabase

3.5.4. Exportações de Dados

A exportação de dados no ColaBD foi concebida para atender à portabilidade técnica do modelo relacional e à documentação do projeto, alinhando-se aos requisitos funcionais RF06 (exportação

[33] O bucket refere-se a um contêiner virtual usado para armazenar dados ou objetos de forma organizada

do projeto para SQL por SGBD), bem como aos requisitos não funcionais RNF03 (distribuição segura de artefatos) e RNF06 (interoperabilidade com múltiplos SGBDs). A implementação combina dois fluxos complementares: (i) geração de *scripts* SQL no *back-end* e distribuição por URL assinada com expiração e (ii) exportação de imagens do diagrama diretamente no *front-end* para fins de relatórios e anexos.

No fluxo de exportação em SQL, o *front-end* serializa o estado do diagrama (JointJS) e o envia ao *endpoint* de geração, informando o SGBD-alvo. O *back-end* valida parâmetros, gera o *script* (via provedor de IA), armazena o resultado em *bucket* privado na Supabase e retorna uma URL assinada com expiração, garantindo distribuição segura (RNF03). No fluxo de **exportação de imagem**, o *front-end* captura a área do diagrama e produz um arquivo de imagem PNG com resolução elevada (RF07).

Para fins de documentação (RF07), na implementação 14, nas linhas 8 a 29, a tela captura o diagrama como imagem utilizando `html2canvas`. Antes da captura, elementos de controle são ocultados, após a captura, o estado visual é restaurado. A resolução é aumentada (*scale*=2) para garantir legibilidade em relatórios.

```
1  async takeScreenshot(filename: string = 'schema-diagram.png'):
       Promise<void> {
2    try {
3      const canvasElement = this.diagramElement.nativeElement as
           HTMLElement;
4      const controlsElement =
           canvasElement.parentElement?.querySelector('.diagram-controls')
           as HTMLElement;
5      if (controlsElement) {
6        controlsElement.style.display = 'none';
7      }
8
9      const canvas = await html2canvas(canvasElement, {
10       backgroundColor: '#ffffff',
11       scale: 2,
12       useCORS: true,
13     });
14
15     if (controlsElement) {
16       controlsElement.style.display = '';
17     }
18
19     canvas.toBlob((blob: Blob | null) => {
20       if (blob) {
21         this.downloadBlob(blob, filename);
22       }
23     }, 'image/png', 1.0);
24   } catch (error) {
25   }
26 }
```

Implementação 14. Captura do diagrama como PNG para documentação

No *back-end*, o *endpoint* `POST /generate-sql` recebe o *schema* (estado serializado do JointJS) e o SGBD alvo (RF06). A partir disso, constrói um *prompt* e consulta o provedor de IA Groq, configurado por variáveis de ambiente (`GROQ_API_KEY`, `GROQ_MODEL`), que retorna um JSON com o campo `sql_code`, conforme demonstrado na implementação 16.

Em seguida, o servidor grava o conteúdo em um arquivo temporário com `content-type: application/sql`, realiza o *upload* para um *bucket* privado na Supabase e gera uma URL assinada com tempo de expiração (RNF03).

A adoção do provedor de IA por meio de um *role system* parametrizável, em vez de um gerador SQL próprio, visa simplificar o código e ampliar a compatibilidade entre diferentes SGBDs. Essa estratégia elimina a necessidade de manter regras sintáticas específicas para cada dialeto SQL, reduzindo a complexidade de manutenção e aumentando a escalabilidade da solução. Assim, o servidor concentra-se apenas na validação do resultado e na disponibilização segura do arquivo.

```
http_client = httpx.Client()
client = Groq(api_key=groq_api_key, http_client=http_client)
completion = client.chat.completions.create(
    model=os.getenv("GROQ_MODEL", "openai/gpt-oss-20b"),
    messages=[
        {"role": "system", "content": "Voce e um assistente que
           responde apenas em JSON valido."},
        {"role": "user", "content": prompt},
    ],
    response_format={"type": "json_object"},
    temperature=0.2,
)
content = completion.choices[0].message.content
parsed = json.loads(content)
sql_code = parsed.get("sql_code")
```

Implementação 15. Geração de SQL via Groq e parsing de resposta JSON

Na implementação 16, após gerar o arquivo, o serviço realiza o *upload* para a Supabase e cria a URL assinada com expiração curta, mitigando riscos de vazamento e acessos indevidos (RNF03).

```
supabase = create_client(connection_url, bucket_key)
storage_bucket = supabase.storage.from_("exports")
upload_resp = storage_bucket.upload(
    path=file_path,
    file=file_bytes,
    file_options={"content-type": "application/sql", "upsert": "true"},
)

signed = storage_bucket.create_signed_url(path=file_path,
   expires_in=3600)
signed_url = signed.get("signedURL") or signed.get("signed_url") or
   signed.get("url")
```

Implementação 16. Upload seguro para Supabase e criação de URL assinada

O padrão adotado permite incluir novos SGBDs com baixo acoplamento (RNF06), basta parametrizar o *prompt* e, quando necessário, aplicar pós-processamento ao `sql_code`. Como limitação, a qualidade do SQL depende do *prompt* e da robustez do dicionário de mapeamento.

3.5.5. Funcionalidades de Tempo Real

A colaboração em tempo real constitui a evolução de maior complexidade da plataforma ColaBD, elevando o suporte colaborativo da etapa assíncrona para a sincronização instantânea entre múltiplos clientes conectados a um mesmo *schema*. Essa funcionalidade materializa o requisito funcional RF04 (colaboração entre usuários) com comportamento simultâneo e, do ponto de vista não funcional, endereça RNF05 (controle de concorrência e prevenção de sobrescrita indevida), por meio de uma arquitetura orientada a eventos com autenticação.

A solução adota WebSocket (WS) com auxilio da biblioteca Socket.IO, com a aplicação cliente (*front-end*) emitindo e recebendo eventos granulares sobre a edição do diagrama (criação, atualização, movimentação e remoção de tabelas), ou seja, só é enviado ou recebido as pequenas alterações ao invés do diagrama inteiro com todas as relações das tabelas a cada modificação.

Contudo, a conexão via WS é autenticada por JWT e cada canal de comunicação é separado com base no identificador do schema, garantindo isolamento de envio e recebimento de dados por diagrama. No servidor, um canal do WebSocket recebe as atualizações, aciona uma camada de serviço responsável pela persistência e manipulação dos dados assíncrona e após isso transmite aos demais clientes conectados àquele schema.

Na implementação 17, é apresentado uma função, no lado do *front-end*, que faz a conexão com o servidor, via WS. Na linha 6 é instanciado o objeto `socket` que será usado para configurar a comunicação e receber e emitir os dados pelos canais. Esse objeto possui dois atributos o `transports` que define o tipo de comunicação que será usada (WebSocket) e o `autoConnect` que vai definir se deve fazer uma requisição automaticamente ao instanciar o objeto `socket` que está com o valor *false* predefinido para deixar essa opção desabilitada, necessário para garantir que a sessão só seja aberta quando o token e o identificador do *schema* estiverem disponíveis.

```
1
2  @Injectable({ providedIn: 'root' })
3  export class SchemaApiWebsocketService {
4    public schemaAtualizadoSubject = new Subject<any>();
5    private baseUrl: string = 'https://develop-colabd.onrender.com';
6    private socket: Socket = io(this.baseUrl, {
7      transports: ["websocket"],
8      autoConnect: false
9    });
10
11   constructor(private localStorageService: LocalStorageService) { }
12
13   connectWS(schemaId: string | null){
14     const token =
         this.localStorageService.obterDadosLocaisSalvos()?.access_token;
15     this.socket.auth = { token, schema_id: schemaId };
16     this.socket.connect();
17
18     this.onCreatedSchema();
19     this.onUpdatedSchema();
20     this.onDeletedSchema();
21     this.onMovedSchema();
22   }
23 }
```

Implementação 17. Recebimento e Envio de eventos no *Front-end*

O método connectWS na linha 13 inicializa a conexão WebSocket, obtendo o token de autenticação na linha 14 e em seguida e o identificador do *schema*, no qual ambos são enviados ao servidor durante a conexão na linha 16. Em seguida, são registrados os *listeners* da linhas 18 a 21), responsáveis por receber eventos de criação, atualização, exclusão e movimentação de tabelas.

Na Implementação 18, o método `atualizacaoSchema` nas linhas 1 a 3 é usado para enviar atualizações ao servidor, emitindo os dados modificados no canal correspondente. Já o método `toClass` iniciado na linha 5 converte os dados recebidos do servidor em instâncias de classes específicas para maior clareza e melhor manuseio deles ao desenvolver o código, em seguida notifica a aplicação sobre a atualização. Por fim, as funções `onCreatedSchema`, `onUpdatedSchema`, `onDeletedSchema` e `onMovedSchema` da linha 10 a 13 ficam aguardando receber eventos do servidor e processam as alterações recebidas, mantendo o modelo de dados sincronizado em tempo real.

```
1   atualizacaoSchema(schema_update: BaseTable, channel_emit: string) {
2     this.socket.emit(channel_emit, schema_update);
3   }
4
5   private toClass<T extends object>(cls: new () => T, data: any): void
        {
6     const received_data = Object.assign(new cls(), data);
7     this.schemaAtualizadoSubject.next(received_data);
8   }
9
10  onCreatedSchema(){ this.socket.on("receive_new_table", d =>
        this.toClass(CreateTable, d)); }
11  onUpdatedSchema(){ this.socket.on("receive_updated_table", d =>
        this.toClass(UpdateTable, d)); }
12  onDeletedSchema(){ this.socket.on("receive_deleted_table", d =>
        this.toClass(DeleteTable, d)); }
13  onMovedSchema(){ this.socket.on("receive_moved_table", d =>
        this.toClass(MoveTable, d)); }
14 }
```

Implementação 18. Recebimento e Envio de eventos no *Front-end*

Com base na Implementação 19, é possível ver as classes em que os dados adquiridos por meio das funções da Implementação 18 que recebem os eventos, são convertidos nessas classes específicas `CreateTable`, `UpdateTable`, `MoveTable` e `DeleteTable`. A classe `BaseTable` (linha 1) contém apenas o identificador da tabela, servindo como estrutura base para os demais modelos. As interfaces *Position* e *Size*, linhas 3 e 4 respectivamente, referenciam a posição e o tamanho da tabela no diagrama.

Contudo são criadas classes específicas para cada tipo de evento: `CreateTable`, que representa a criação de uma nova tabela e inclui informações como tipo, posição, tamanho e atributos visuais; `UpdateTable`, que define atualizações nas propriedades de uma tabela existente; `MoveTable`, que indica o deslocamento de uma tabela na interface; e `DeleteTable`, que representa a exclusão de uma tabela. Esses modelos garantem que as mensagens trocadas por meio do WebSocket sejam tipadas, padronizadas e consistentes, facilitando a comunicação entre cliente e servidor.

```
1 export class BaseTable { id: string = ""; }
2
3 export interface Position { x: number; y: number; }
4 export interface Size { width: number; height: number; }
```

```
5
6  export class CreateTable extends BaseTable {
7    type: string = "standard.Rectangle";
8    position!: Position;
9    size!: Size;
10   attrs!: object;
11 }
12
13 export class UpdateTable extends BaseTable { attrs!: Partial<object> |
       {}; }
14 export class MoveTable extends BaseTable { position!: Position; }
15 export class DeleteTable extends BaseTable { }
```

Implementação 19. Modelo de mensagens tipadas para eventos de colaboração

No apêndice A, que ilustra tela do diagrama, a conexão WebSocket é estabelecida com o `schema_id` obtido da rota. No construtor (linhas 1 a 5), são injetados os serviços responsáveis pela comunicação e pela obtenção de parâmetros da rota. No método `ngOnInit` (linha 7), a conexão é iniciada por meio de `connectWS` (linha 8), e o componente `schemaAtualizadoSubject` é usado para monitorar e receber qualquer alteração de tabelas vindas do servidor (linhas 10 a 13).

Quando uma atualização é recebida, a variável `dadosRecebidos` é ativada (linha 11) para evitar o reenvio da mesma mensagem, e o método `loadJointJSFromWS` (linha 12) aplica as alterações no diagrama, mantendo a sincronização em tempo real.

O apendice B representa o código da função `setupGraphChangeListener` que é responsável por monitorar alterações locais no diagrama e enviar essas modificações ao servidor via WebSocket. Ele registra diferentes manipuladores para eventos do JointJS, garantindo que apenas ações iniciadas pelo usuário (e não as recebidas do servidor) sejam transmitidas.

Nas linhas 2 a 5, o evento `add` detecta a criação de novas tabelas e, caso a alteração não tenha origem no servidor (`!this.dadosRecebidos`), chama o método `addCellAndSend` para enviar a atualização. Logo em seguida da linha 8, o contador `indexTablesLoaded` é incrementado de forma controlada, garantindo a sincronização entre o número de tabelas carregadas e as efetivamente processadas.

O evento `remove` (linhas 10 a 15) trata a exclusão de elementos utilizando o método `removeCellAndSend`. Já o evento `change:attrs` (linhas 16 a 20) monitora alterações visuais ou estruturais nos atributos das tabelas, enviando-as por meio de `updateCellAndSend`. Por fim, o evento `change:position` (linhas 22 a 28) detecta movimentações de elementos no diagrama e chama `moveCellAndSend` quando aplicável, também atualizando o contador `indexTablesLoaded` (linha 27).

Essa estrutura garante que as mudanças locais sejam refletidas no servidor sem gerar *loop*, ou seja, sem reemitir atualizações recebidas e mantém o estado do diagrama sempre sincronizado e consistente entre os usuários.

No servidor, o canal de comunicação em tempo real é implementado com a biblioteca `Socketio` operando o código que controla a parte responsável por gerenciar conexões, autenticação e difusão de eventos entre os clientes, como mostra na Implementação 20.

As linhas 4 a 7 definem a lista de origens permitidas (`origins`) para conexões WebSocket, configurando as políticas de CORS e garantindo a comunicação segura apenas com domínios autorizados. Em seguida, nas linhas 9 a 12, a instância `sio` é criada com o modo assíncrono

ativado e as permissões de CORS aplicadas.

A chamada ao método `sioenter_room` (linhas 14 a 19) na implementação 20, registra dinamicamente os nomes dos eventos WebSocket com base no `schema_id`, vinculando cada tipo de ação, como criação, exclusão, atualização e movimentação de tabelas a um schema específico.

O evento `connect` na linha 7 é executado sempre que um novo cliente se conecta. Ele extrai o `token` e o `schema_id` do objeto `auth` (linhas 8 e 9) e utiliza a função `get_current_user_WS` para validar o token e identificar o usuário (linha 11). Em seguida, associa o usuário e o esquema ao serviço WebSocket (linhas 13 a 14) e armazena o relacionamento entre o identificador de sessão e o schema (linha 16).

Essa arquitetura garante que cada schema tenha seus próprios canais de evento, evita o *loop* de mensagens por meio do controle de `sid`, e mantém a autenticação unificada com o canal REST, reforçando os requisitos RF04 (sincronização colaborativa) e RNF05 (segurança de autenticação).

```
1  sio = socketio.AsyncServer(
2      async_mode="asgi",
3      cors_allowed_origins=origins
4  )
5
6  @sio.event
7  async def connect(sid, environ, auth):
8      token = auth.get("token")
9      schema_id = auth.get("schema_id")
10
11     user_id: str = get_current_user_WS(token)["id"]
12
13     user_sid_schemaId[sid] = schema_id
14     user_sid_userId[sid] = user_id
15
16     await sio.enter_room(sid, schema_id)
17
18     await service_websocket.initialie_cells(schema_id, user_id)
19
20     logger.info(f"User {user_id} connected to schema {schema_id} (sid:
           {sid}). Socket joined room '{schema_id}'")
```

Implementação 20. Esqueleto do controlador do canal e eventos

Na Implementação 21, é ilustrado o trecho do código onde há a validação do token do usuário, assim impedindo conexões não autenticadas:

```
1  def get_current_user_WS(user_token: str) -> dict:
2      return get_token_decoded(user_token)
3
4  def get_token_decoded(token: str):
5      result = decode_access_token(token)
6
7      if not result.success:
8          raise HTTPException(
9              status_code=status.HTTP\_401\_UNAUTHORIZED,
10             detail="Token invalido ou expirado",
11             headers={"WWW-Authenticate": "Bearer"},
12         )
13
```

```
14      return result.data
```

Implementação 21. Validação do token no `connect`

O trecho de código apresentado na Implementação 22, mostra um mecanismo assíncrono de persistência de dados com o objetivo de evitar conflitos e reduzir o número de escritas consecutivas no banco de dados. O método `salvamento_agendado` é responsável por gerenciar o agendamento do salvamento das alterações recebidas.

Em seguida, o código verifica se já existe uma tarefa de salvamento pendente associada ao mesmo `schema_id`. Caso exista, essa tarefa é cancelada, indicando que o esquema foi alterado novamente antes do término do salvamento anterior. Essa abordagem evita que múltiplas operações concorram para gravar dados obsoletos ou intermediários no banco. Após o cancelamento, uma nova tarefa é criada utilizando `asyncio.create_task`, que executa o método `salvamento_com_atraso`. Esse método realiza a persistência real dos dados após um pequeno intervalo de tempo, permitindo agrupar várias modificações sucessivas em uma única operação.

Dessa forma, o servidor adota uma estratégia de `debounce`[34] assíncrono, onde as alterações são temporariamente acumuladas e somente a versão final é persistida. Isso melhora a eficiência do sistema, reduz a carga sobre o banco de dados e assegura que apenas o estado mais atualizado do esquema seja armazenado.

```
1  def __preprocess_schema_received_data(self, received_data: BaseTable):
2
3  async def salvamento_agendado(self, received_data: BaseTable):
4      if (self.schema_id not in self.pending_updates):
5
6          except asyncio.CancelledError:
7              logger.info(f"Operacao cancelada, pois o schema foi
                   alterado")
8              return
```

Implementação 22. Serviço de persistência assíncrona

A persistência final reusa a `ServiceSchema` para: validar existência do *schema*; checar o vínculo/permiso do usuário; salvar o conteúdo (células) no MongoDB e atualizar o ponteiro `database_model` no registro do *schema*. Essa separação permite evoluir o formato do documento salvo sem alterar o contrato do *schema*.

A figura15 ilustra a comunicação em tempo real por meio do WebSocket, onde o usuário acessa a plataforma, responsável por exibir e editar os bancos de dados em formato de tabelas. As alterações realizadas na interface são enviadas pelo WebSocket, que mantém um canal separado para cada schema, permitindo sincronização imediata entre diferentes usuários.

O *back-end* recebe essas atualizações, executa as operações de criação, exclusão, movimentação e atualização, e aplica um controle para evitar gravações excessivas. Além disso, os dados são armazenados no Supabase, que guarda informações de usuários e schemas, e no MongoDB, que armazena os dados das tabelas.

[34] `Debounce` é uma técnica usada para controlar por tempo a frequência com que uma função é executada

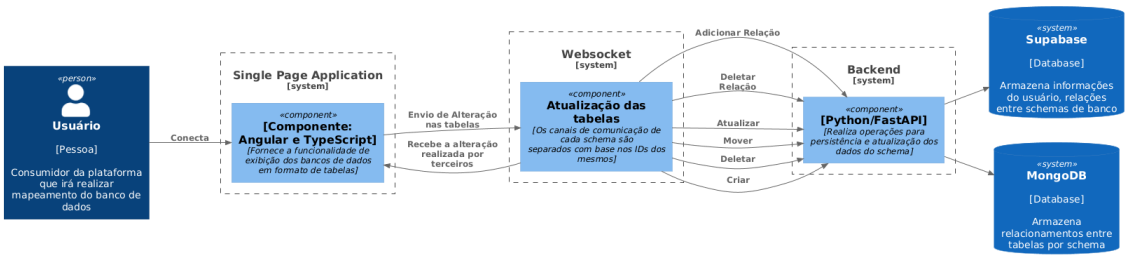

Figura 15. Diagrama de fluxo dos dados por meio do WebSocket.

3.5.6. Versionamento de Esquemas

No controle de versões permite gerenciar e acompanhar a evolução de um diagrama por meio do registro de versões salvas. Nele, são listadas as versões existentes com informações como autor responsável, nome da versão e data e horário do salvamento. A versão mais recente é destacada como versão atual, facilitando a identificação do estado atual do diagrama, enquanto versões anteriores permanecem disponíveis para consulta ou restauração, como é possível ver na Figura 16.

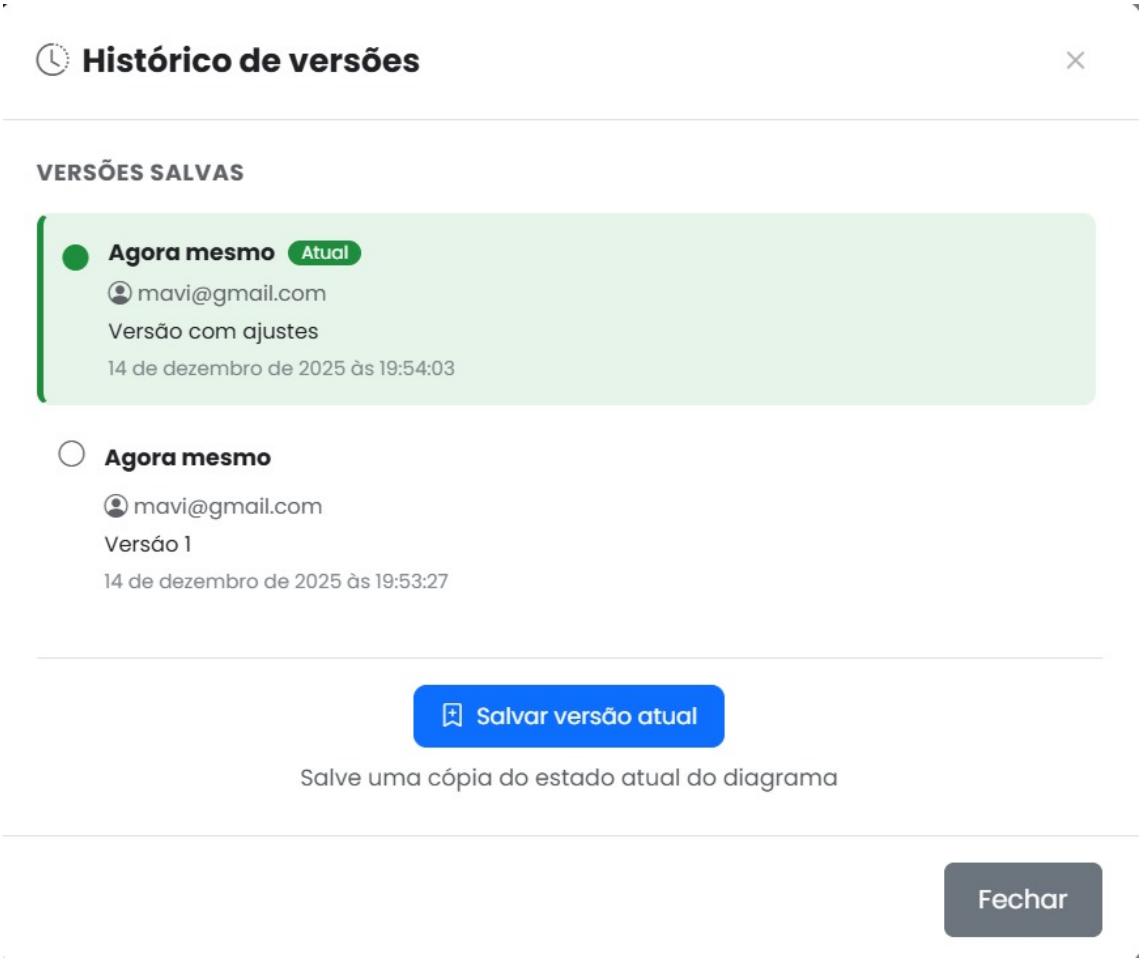

Figura 16. Modal com histórico de alterações.

A implementação 23 é responsável por buscar e retornar todas as versões associadas a um esquema específico. Ele recebe o `schema_id` para identificar o esquema desejado e o `current_user_id` (que, neste trecho, não é utilizado). Inicialmente, o método registra logs informando a operação e consulta o repositório (repo_version) para obter as versões armazenadas no banco de dados.

Caso a consulta não seja bem-sucedida, o método retorna diretamente o resultado de erro. Se for bem-sucedida, ele organiza os dados retornados e devolve essas informações encapsuladas em um objeto indicando sucesso. Em situações de erro inesperado, a exceção é capturada, registrada nos logs e retornada como uma resposta de falha.

```python
async def get_versions_by_schema(self, schema_id: str,
    current_user_id: str) -> Response:
        try:
            logger.info(f"Service: Getting versions for schema
                {schema_id}")

            result = await
                self.repo_version.get_versions_by_schema_id(schema_id)

            if not result.success:
                return result

            version_details_list = result.data or []
            logger.info(f"Service: Returning
                {len(version_details_list)} version details")

            return Response(data=version_details_list, success=True)

        except Exception as e:
            logger.error(f"Service: Error in get_versions_by_schema:
                {str(e)}")
            return Response(data=str(e), success=False)
```

Implementação 23. Recuperando versões antigas do *schema*

4. Avaliações e Discussões

Com o objetivo de avaliar a usabilidade e a percepção dos usuários quanto à eficiência da plataforma, foi disponibilizado um formulário de feedback para doze participantes que testaram o sistema. O questionário abordou aspectos como facilidade de uso, ocorrência de erros, tempo de resposta, confiança na utilização e percepção geral sobre as funcionalidades implementadas.

O formulário de avaliação foi disponibilizado a estudantes e profissionais da área de Computação, com experiência em desenvolvimento de software e modelagem de banco de dados. O objetivo do questionário foi coletar percepções sobre a experiência de uso, desempenho e usabilidade do sistema, a fim de identificar seus pontos fortes e possíveis melhorias. No momento da avaliação realizada com os participantes, as implementações descritas na subseção 3.5.6 ainda não haviam sido finalizadas. Os dados brutos da pesquisa encontram-se disponíveis no Apêndice C.

As perguntas aplicadas foram:

1. A utilização do sistema é fácil e intuitiva?
2. O sistema apresentou erros ou travamentos durante o uso?
3. O tempo de resposta do sistema foi satisfatório?

4. Você se sentiu confiante ao utilizar o sistema?
5. Quais funcionalidades do sistema você achou mais interessantes?
6. Como você melhoraria este sistema? Escreva sugestões, críticas e elogios.
7. Em uma escala de 0 a 10, o quanto você recomendaria este sistema para outros colegas?

A análise das respostas obtidas demonstrou resultados amplamente positivos. A maioria dos participantes afirmou que a utilização do sistema foi fácil e intuitiva, conforme apresentado na Figura 17, na qual observa-se que 58,3% dos respondentes marcaram a opção “Concordo totalmente” e 33,3% “Concordo”.

Figura 17. Percepção dos usuários quanto à facilidade e intuição de uso.

Em relação à estabilidade da aplicação, a Figura 18 demonstra que 41,7% dos usuários afirmaram que o sistema nunca apresentou erros ou travamentos, enquanto 50% apenas de forma “Rara”, já 9,1% relataram ocorrência “Ocasional”. Esses resultados sugerem que, embora ainda existam situações pontuais de instabilidade, o comportamento geral da aplicação é considerado confiável pela maior parte dos participantes.

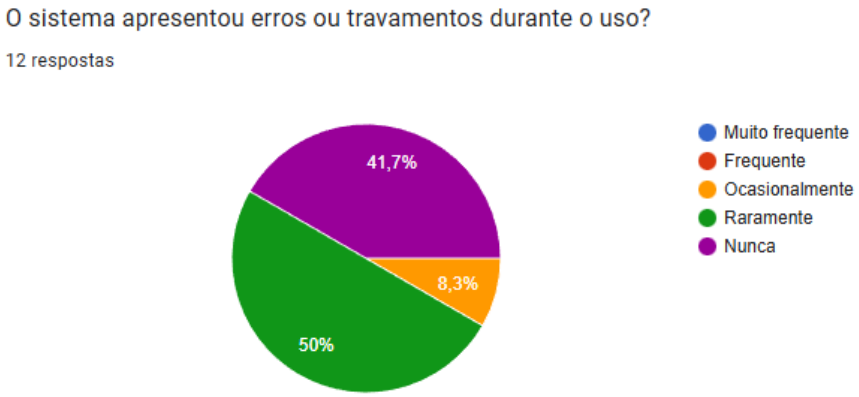

Figura 18. Ocorrência de erros ou travamentos durante o uso.

Com relação ao desempenho, a Figura 19 indica que 58,3% dos participantes “Concordaram” e 33,3% “Concordaram totalmente” que o tempo de resposta do sistema foi satisfatório.

O elevado percentual de respostas positivas demonstra que, mesmo em um ambiente de testes e ainda em fase de implementação de funcionalidades, o sistema consegue oferecer desempenho adequado, sem atrasos perceptíveis que prejudiquem a experiência de uso.

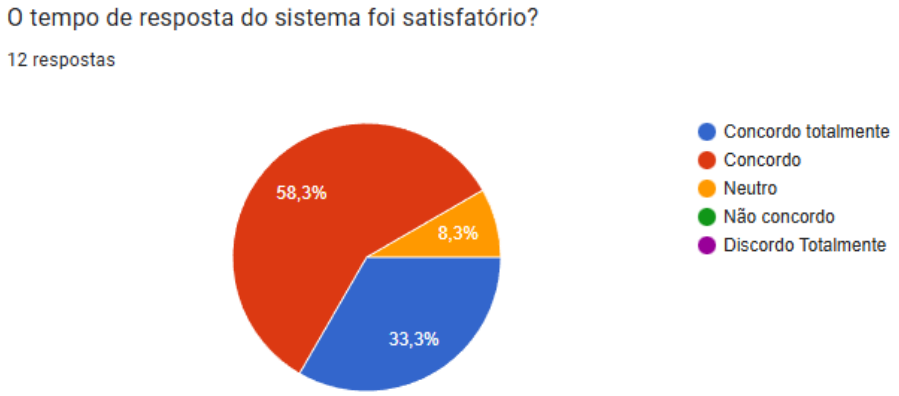

Figura 19. Avaliação do tempo de resposta do sistema.

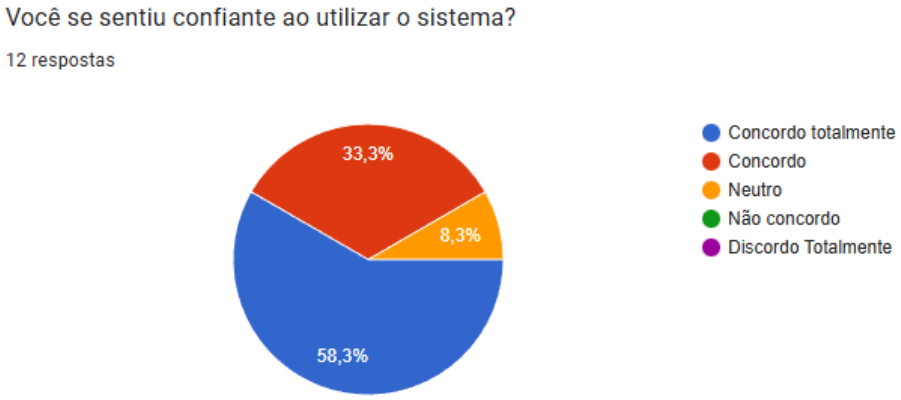

Figura 20. Confiança dos usuários ao utilizar o sistema.

Quanto à confiança na utilização, a Figura 20 demonstra que 58,3% dos usuários "Concordaram totalmente" e 33,3% "Concordaram" que se sentiram confiantes ao utilizar o sistema. A sensação de confiança está diretamente relacionada à usabilidade percebidae à facilidade de navegação, elementos que contribuem para que o usuário compreenda melhor o comportamento da aplicação e reduza a ocorrência de erros durante o uso.

Em relação à recomendação do sistema, a Figura 21 apresenta a distribuição das notas atribuídas pelos participantes na escala de 0 a 10. As notas concentraram-se entre 7 e 10, com destaque para a nota 10, atribuída por 58,3% dos respondentes (7 participantes).

A análise dos resultados demonstrou uma avaliação positiva da plataforma. A maioria dos participantes afirmou que a utilização do sistema foi fácil e intuitiva, destacando a clareza das

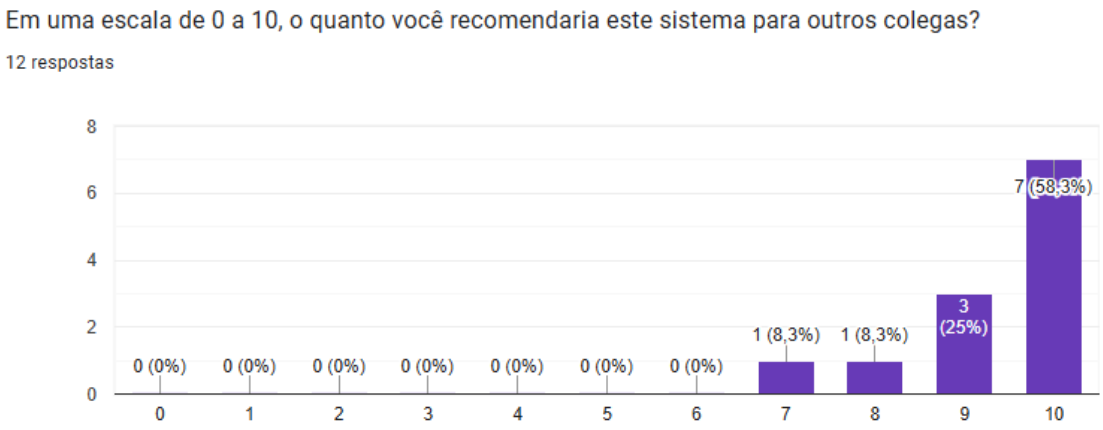

Figura 21. Recomendação do sistema pelos usuários.

ações e a boa organização da interface. Da mesma forma, observou-se que o sistema apresentou baixo índice de erros ou travamentos, evidenciando estabilidade e bom desempenho técnico durante os testes.

No que se refere ao tempo de resposta da plataforma, todos os participantes avaliaram a experiência como satisfatória, destacando que as interações ocorreram de forma fluida e sem interrupções perceptíveis. Essa percepção se relaciona diretamente com a capacidade do sistema de processar, em tempo hábil, as operações de modelagem, validação e geração de consultas SQL, mesmo quando executadas em sequência ou em contextos de maior demanda.

Quanto ao nível de confiança no uso da aplicação, os usuários relataram sentir segurança ao interagir com as funcionalidades disponibilizadas, demonstrando clareza na compreensão das ações realizadas e dos resultados esperados. De modo geral, os participantes afirmaram que o fluxo de uso é intuitivo, favorecendo a aprendizagem inicial e contribuindo para que o usuário se sinta amparado pelo sistema ao longo de suas atividades de modelagem e manipulação do banco de dados.

Na questão sobre as funcionalidades mais apreciadas, foram mencionadas com maior frequência: a geração automática de scripts SQL, a visualização interativa dos relacionamentos entre tabelas e a possibilidade de colaboração em tempo real. Para ilustrar a frequência e relevância das palavras mencionadas pelos participantes, foi construída uma nuvem de palavras, apresentada na Figura 22.

As sugestões coletadas concentram-se em aprimoramentos estéticos e de usabilidade, como a adição de um modo escuro, melhorias visuais no editor e novas opções de exportação de banco de dados. Por fim, ao avaliar a recomendação do sistema em uma escala de 0 a 10, a média obtida foi de 9,3, demonstrando alto nível de aceitação entre os participantes.

Além disso, também foi elaborado um caso de teste interno ao final do desenvolvimento, voltado à validação da sincronização em tempo real entre múltiplos usuários. Assim, estruturou-se uma ficha de teste seguindo o modelo tradicional proposto por Firesmith (1999), contemplando pré-condições, procedimentos detalhados de entrada e saída e as respectivas pós-condições. O caso de teste, representado na Figura 23, descreve um cenário em que três usuários acessam simultaneamente a mesma modelagem e executam ações como criação, movimentação, edição e

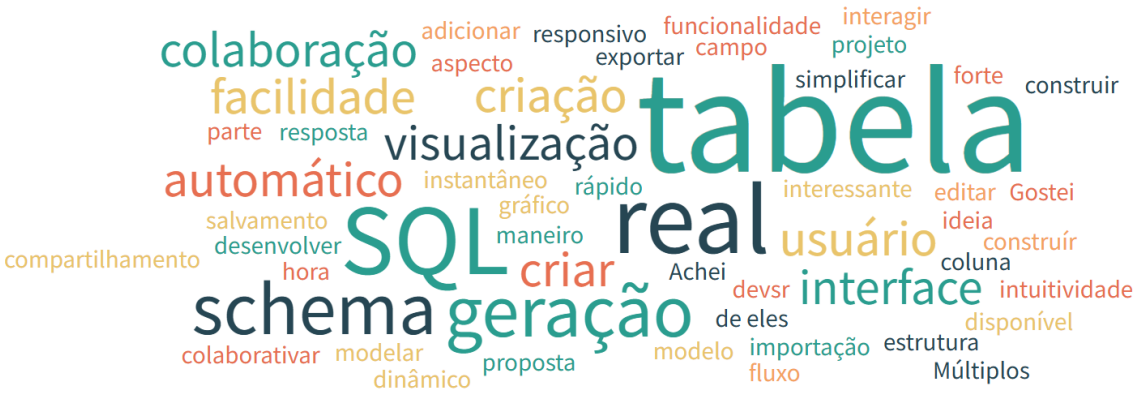

Figura 22. Nuvem de palavras referente às respostas da questão 5

remoção de elementos do diagrama.

Caso de Teste	
Nome	Validação da Sincronização em Tempo Real com 3 Usuários Simultâneos
Ambiente	ColaBD
Ator	Usuário da Plataforma ColaBD
Pré Condições	Três contas diferentes devem ter acesso ao mesmo projeto com acesso simultâneo.
Procedimentos (Entradas e Saídas)	1. Abra a plataforma ColaBD. 2. Usuário 1 acessa a modelagem. 3. Usuários 2 e 3 acessam a mesma modelagem. 4. Usuário 1 adiciona uma nova tabela chamada “CLIENTES”. 5. O sistema deve exibir imediatamente a nova tabela para os usuários 2 e 3. 6. Usuário 2 move a tabela CLIENTES para outra posição no canvas. 7. A nova posição deve ser refletida para todos os usuários. 8. Usuário 3 adiciona uma coluna chamada “id_cliente (int)” dentro da tabela CLIENTES. 9. Todos os usuários devem visualizar a nova coluna.
Pós-Condições	Todas as ações realizadas por cada usuário foram refletidas corretamente aos demais sem perda de informação

Figura 23. Caso de Teste de sincronização em tempo real

Em relação aos projetos relacionados (Seção 2.3), o Quadro 3 apresenta o comparativo

Quadro 3. Comparação entre trabalhos relacionados e a plataforma do TCC.

Critério	DrawSQL	PopSQL	DBDiagram.io	ColaBD
Gratuito	Parcial	Parcial	Parcial	Sim
Modelagem Visual	Sim	Não	Sim	Sim
Exportação SQL	Sim	Não	Sim	Sim
Edição simultânea	Parcial	Parcial	Parcial	Sim
Modelagem Lógica	Sim	Não	Sim	Sim
Editor SQL	Não	Sim	Não	Não
Multi-SGBD	Sim	Sim	Não	Sim

entre sistemas similares ao presente trabalho. Os critérios comparativos foram definidos com base nos requisitos identificados como essenciais para a proposta deste trabalho e cada célula foi preenchida por "Sim", "Não" ou "Parcial", que corresponde se o critério, de cada linha, está presente, ausente ou você deve pagar para possuir a funcionaliade nos sistemas de cada coluna. Os critérios são:

- Gratuito: Quando a plataforma disponibiliza seus recursos sem exigir qualquer tipo de pagamento, permitindo que o usuário explore e utilize suas funcionalidades livremente, sem custos.
- Modelagem Visual: refere-se à capacidade de representar graficamente as estruturas e relacionamentos de tabelas.
- Exportação SQL: indica se a ferramenta permite gerar scripts SQL a partir da modelagem criada.
- Edição simultânea: diz respeito à colaboração em tempo real entre diferentes usuários de diferentes localidades no mesmo diagrama.
- Editor SQL: avalia se o sistema oferece um ambiente integrado para escrita e execução de consultas.
- Multi-SGBD: verifica o suporte a múltiplos sistemas gerenciadores de banco de dados, aumentando a flexibilidade da ferramenta.

Com base na comparação realizada, observa-se que nenhuma das plataformas analisadas atende plenamente a todos os critérios estabelecidos. Enquanto algumas se destacam pela interface amigável ou pela integração com bancos reais, carecem de funcionalidades colaborativas ou suporte multiplataforma. A ferramenta proposta neste trabalho, o ColaBD tem uma abordagem mais completa ao buscar integrar visualização, colaboração, e suporte a múltiplos SGBDs, com perspectivas de evolução para edição simultânea em tempo real.

5. Considerações Finais

O presente trabalho teve como objetivo desenvolver uma plataforma de modelagem e mapeamento de bancos de dados relacional com colaboração em tempo real. Para isso, o desenvolvimento dessa plataforma foi dividido em etapas que envolvem a análise das necessidades no processo de modelagem de dados, o estudo de ferramentas colaborativas e a criação de uma solução interativa.

Contudo, ao desenvolver a plataforma, foram encontrados alguns desafios, como a implementação da comunicação via WebSocket, no qual os eventos enviados e recebidos por meio de diferentes canais do WebSocket, teriam que se agrupar de forma ordenada, tanto no *front-end* para que pudesse ser visível e de forma organizada os *schemas*, quanto no *back-end*, onde os dados seriam preparados e persistidos no banco de dados.

Logo, para futuras melhorias na escolha das tecnologias utilizadas, as ferramentas e linguagens empregadas atualmente poderiam ser substituídas por alternativas mais modernas e performáticas, tanto no *front-end* quanto no *back-end*, com o objetivo de aprimorar o desempenho geral e a escalabilidade da plataforma.

No *front-end* poderiamos ter desenvolvido o trabalho com o framework *React*[35], pois ele possui um suporte melhor para desenvolvimento com *threads* e códigos assíncronos e a utilização da biblioteca *Tailwind*[36] para melhor estilização e customização da página da plataforma, essas duas tecnologias juntas seriam melhores do que as atuais, sendo elas *Angular* e *Angular Material*[37].

No back-end, alternativas como Java[38], Kotlin[39] ou Golang[40] poderiam ser consideradas. Essas linguagens são amplamente reconhecidas por sua eficiência, segurança e robustez. Em especial, o Golang se destaca por sua leveza e rapidez na execução de serviços concorrentes, enquanto Kotlin e Java apresentam ecossistemas maduros e grande compatibilidade com frameworks modernos de desenvolvimento de APIs.

Além disso, sugere-se a implementação de uma página dedicada à gestão de times, na qual o usuário poderá criar, editar, remover, visualizar e participar de equipes existentes. Dessa forma, o usuário terá liberdade para escolher a ação desejada entre as opções disponíveis, tornando a interação com o sistema mais completa e colaborativa.

Continuando com a parte de novas funcionalidades da plataforma, propõe-se a criação de um editor de scripts SQL integrado, permitindo que o usuário escreva e execute comandos SQL e, a partir deles, o sistema gere automaticamente o diagrama visual do banco de dados relacional. Essa funcionalidade uniria a modelagem textual e gráfica, aumentando a flexibilidade da plataforma e facilitando a compreensão e validação da estrutura do banco.

Referências

Ambler, S. W. (2003). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, Indianapolis.

Apple Inc. (2024). Apple. Acessado em: 14 de junho de 2025.

Bjeladinovic, S., Marjanovic, Z., e Babarogic, S. (2020). A proposal of architecture for integration and uniform use of hybrid sql/nosql database components. *Journal of Systems and Software*, 168:110633.

Booch, G., Rumbaugh, J., e Jacobson, I. (2005). *UML – Guia do Usuário*. Elsevier, Rio de Janeiro, 2 edition.

Cherif, A. e Imine, A. (2015). A constraint-based approach for generating transformation patterns. *arXiv preprint arXiv:1512.07684*.

[35]Disponível em: `https://react.dev/`
[36]Disponível em: `https://tailwindcss.com/`
[37]Disponível em: `https://material.angular.dev/`
[38]Disponível em: `https://www.java.com/pt-BR/`
[39]Disponível em: `https://kotlinlang.org/`
[40]Disponível em: `https://go.dev/`

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Date, C. J. (2004). *Introdução a Sistemas de Bancos de Dados*. Campus, Rio de Janeiro, 8 edition.

Ellis, C., Gibbs, S., e Rein, G. (1991). Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58.

Elmasri, R. e Navathe, S. B. (2018). *Sistemas de Banco de Dados*. Pearson, São Paulo, 7 edition.

Figma (2025). Figma: Collaborative interface design tool. Accessed: 2025-06-11.

Firesmith, D. G. (1999). Use case modeling guidelines. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 184–193. IEEE.

Google LLC (2024). Google. Acessado em: 14 de junho de 2025.

Greif, I. e Cashman, P. (1988). Introduction to computer-supported cooperative work. In *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann, San Mateo.

Gruber, M., Lukosch, S., Schneider, K., e Knauss, E. (2017). Real-time collaboration in software engineering. *IEEE Software*, 34(6):30–35.

Grudin, J. (1994). *Computer-supported cooperative work: History and focus*. IEEE Computer Society Press.

Shapiro, M., Preguiça, N., Baquero, C., e Zawirski, M. (2011). Conflict-free replicated data types (crdts). In *Lecture Notes in Computer Science*, volume 6976, pages 386–400. Springer.

Silberchatz, A., Korth, H. F., e Sudarshan, S. (2020). *Sistemas de Banco de Dados*. AMGH Editora, 7 edition.

Sun, C. e Ellis, C. (1998). Operational transformation for real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 59–68. ACM.

Wang, D. L. e Sun, C. (2016). Real-time collaborative programming environments: A survey. *IEEE Transactions on Software Engineering*, 42(12):1072–1090.

A. Integração com a tela: receber e aplicar atualizações

```
constructor(
  private schemaService: SchemaService,
  private socketService: SchemaApiWebsocketService,
  private route: ActivatedRoute
) { }

ngOnInit(): void {
  this.socketService.connectWS(this.route.snapshot.paramMap.get("id"));
  this.subscription.add(
    this.socketService.schemaAtualizadoSubject.subscribe((received) =>
        {
      this.dadosRecebidos = true;
      this.loadJointJSFromWS(received);
    })
  );
}
```

B. Emissão de alterações e prevenção de eco

```
private setupGraphChangeListener() {
    this.graph.on('add', (cell: joint.dia.Cell) => {
      if(!this.dadosRecebidos && this.indexTablesLoaded >
          this.qtTablesLoaded){
        this.addCellAndSend(cell);
      }

      this.indexTablesLoaded += this.indexTablesLoaded <=
          this.qtTablesLoaded? 1 : 0;
    });

    this.graph.on('remove', (cell: joint.dia.Cell) => {
      if(!this.dadosRecebidos){
        this.removeCellAndSend(cell);
      }
    });

    this.graph.on('change:attrs', (cell: joint.dia.Cell) => {
      if(!this.dadosRecebidos){
        this.updateCellAndSend(cell);
      }
    });

    this.graph.on('change:position', (cell: joint.dia.Cell) => {
      if(!this.dadosRecebidos && this.indexTablesLoaded >
          this.qtTablesLoaded){
        this.moveCellAndSend(cell);
      }

      this.indexTablesLoaded += this.indexTablesLoaded <=
          this.qtTablesLoaded? 1 : 0;
    });
}
```

C. Respostas avaliação da plataforma

Questão 5 – Quais funcionalidades do sistema você achou mais interessante?

Resposta 1: “A visualização das tabelas que estão sendo construídas através da interface e o compartilhamento dos schemas.”

Resposta 2: “A colaboração entre usuários em tempo real dentro de um Schema e a importação deles.”

Resposta 3: “A intuitividade foi um aspecto muito forte e o fluxo foi bem simplificado e bem construído. Gostei bastante da interface e como o projeto foi desenvolvido, muito interessante a proposta!.”

Resposta 4: “O salvamento automático na criação das tabelas”

Resposta 5: “A geração de SQL automática e a colaboração em tempo real para vários usuários.”

Resposta 6: “A facilidade para criar e editar tabelas e colunas, e a visualização gráfica das estruturas.”

Resposta 7: “A forma como é criada as tabelas e facilidade em adicionar os campos da tabela”

Resposta 8: “A capacidade de edição simultanea das tabelas, permitindo interações rápidas entre as versões e simplificando a comunicação”

Resposta 9: “As partes colaborativa e geração de SQL”

Resposta 10: “A ideia geral de múltiplos devs interagindo com o mesmo modelo, já era hora de um sistema responsivo em tempo real estar disponível.”

Resposta 11: “Achei muito maneiro a parte de poder exporta o SQL que foi modelado.”

Resposta 12: “Funcionalidade em tempo real, com tempo de resposta instantâneo; Criação dinâmica e rápida das tabelas; Geração de SQL a partir do schema;”

Questão 6 – Como você melhoraria este sistema? Escreva sugestões, críticas e elogios.

Resposta 1: “Incluir ícones mais específicos acercar de atributos únicos; trazer a possibilidade de relacionamento para junto das tabelas, pois está junto com as configurações de visualização; Ao adicionar mais atributos em uma mesma tabela trabalhar com tipos específicos pré-definidos (exemplo, varchar); Melhorar os relacionamento entre as tabelas(forma visual), incluir a tabela intermediária no momento que sugere-se o relacionamento com cardinalidades n - n; o nome deveria ser modelo, ou algo relacionado e não schemas; Trazer a possibilidade de exportar o script para ORM.”

Resposta 2: “Software com uma utilidade enorme. Otimiza muito a etapa de modelagem. No entanto, pouco otimizado. Alguns loadings são mais demorados, mas nada desconfortáveis.”

Resposta 3: “Ajustaria o visual das tabelas, permitindo que o usuário alterasse o seu tamanho.”

Resposta 4: “Deixaria a criação de relacionamento entre as tabelas mais intuitiva”

Resposta 5: “Apenas corrigir os bugs que ocorrem raramente e o sistema já estará completamente funcional.”

Resposta 6: “Poderia ter um jeito mais intuitivo de criar relacionamentos entre tabelas dentro do campo de edição da tabela; Poderia salvar as alterações automaticamente.”

Resposta 7: “Opção de DarkMode - Domínio Próprio - Possibilidade de Edição dos dados do usuários”

Resposta 8: “Trazer o icone de clipes para conectar duas tabelas diretamente no visual de cada tabela”

Resposta 9: “-”

Resposta 10: “Acho que deixaria um pouco mais intuitivo a parte do relacionamento das tabelas, mas de resto está show.”

Resposta 11: “Uma sugestão: adicionar o nome do schema que está sendo editado na barra lateral (ao lado do botão de ”Gerar SQL do schema”, por ex.), para que o usuário consiga se situar melhor.

Parabéns pelo excelente TCC! Ideia inovadora, com o desenvolvimento em tempo real de esquemas de banco de dados e execução fantástica!”

Resposta 12: “A interface é bem simples e fácil de usar e o sistema responde bem de forma geral, apesar de algumas interações levarem um pouco mais de tempo (.e.g exportar SQL ou screenshot do diagrama). Minha única sugestão de ”melhoria”é permitir que a opção ”Criar Relacionamento”permita vincular um campo já existente na tabela em vez se sempre inserir um novo nas tabelas.”