Universidade de Brasília – UnB

Faculdade de Ciências e Tecnologias em Engenharia – FCTE

Engenharia de Software

Estratégias de desenvolvimento orientadas à busca de desempenho na plataforma Brasil Participativo

Autor: Victor Jorge da Silva Gonçalves

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2025

Victor Jorge da Silva Gonçalves

Estratégias de desenvolvimento orientadas à busca de desempenho na plataforma Brasil Participativo

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade de Ciências e Tecnologias em Engenharia – FCTE

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2025

Victor Jorge da Silva Gonçalves
Estratégias de desenvolvimento orientadas à busca de desempenho na plataforma Brasil Participativo/ Victor Jorge da Silva Gonçalves. – Brasília, DF, 2025-
91 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade de Ciências e Tecnologias em Engenharia – FCTE , 2025.

1. otimização. 2. desempenho. I. Prof. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade de Ciências e Tecnologias em Engenharia. IV. Estratégias de desenvolvimento orientadas à busca de desempenho na plataforma Brasil Participativo

CDU 02:141:005.6

Victor Jorge da Silva Gonçalves

Estratégias de desenvolvimento orientadas à busca de desempenho na plataforma Brasil Participativo

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 22 de julho de 2025:

Prof. Dr. Renato Coral Sampaio
Orientador

Prof. Dr. Maurício Serrano
Convidado 1

Prof. Dr. John Lenon Cardoso Gardenghi
Convidado 2

Brasília, DF
2025

Dedico este trabalho à minha mãe, que sempre me apoiou e ajudou durante minha trajetória, sendo a pessoa mais importante em todo o processo.

Agradecimentos

Agradeço ao Prof. Dr. Renato Coral Sampaio por todo o apoio, gentileza e paciência durante o processo de orientação.

Agradeço à Prof. Dr. Carla Rocha pelo apoio e oportunidade de me aprofundar no tema através da minha participação no projeto do Brasil Participativo.

Resumo

Aplicações web modernas geralmente se dividem em três grandes partes: *back-end*, *front-end* e banco de dados. *Frameworks* baseados no padrão de projeto como o *Model View Controller* (MVC) podem experienciar de problemas de escalabilidade e desempenho, principalmente em relação ao uso intensivo do banco de dados e gargalos de I/O para renderização de visões. Um caso real, a plataforma Brasil Participativo, atingiu um número relativamente alto de usuários (um milhão e seiscentos mil), o que revelou diversos pontos de fraqueza e gargalos na arquitetura desenvolvida e na abordagem adotada no desenvolvimento da *gem Decidim*, construída em cima do *framework Ruby on Rails*. Este trabalho tem por objetivo descrever a jornada de desenvolvimento de melhorias em uma das funcionalidades da plataforma, o componente de textos participativos, sob o viés de otimização de desempenho. Para tal, será realizada uma pesquisa sobre boas práticas de desenvolvimento utilizando o *framework Ruby on Rails*, e estratégias de utilização de seus recursos que priorizem o desempenho, principalmente na elaboração das *views*. Por fim, serão realizados testes para verificar o impacto causado no desempenho da ferramenta de textos participativos.

Palavras-chave: Aplicações Web. Padrão de Projeto MVC. Ruby on Rails. Desempenho.

Abstract

Modern web applications are generally divided into three main parts: back-end, front-end, and database. Frameworks based on design patterns such as Model View Controller (MVC) can experience scalability and performance issues, especially related to intensive use of the database and I/O bottlenecks during view rendering. A real-world case, the Brasil Participativo platform, reached a relatively high number of users (one million six hundred thousand), which exposed several weaknesses and bottlenecks in the developed architecture and in the development approach adopted for the Decidim gem, built on top of the Ruby on Rails framework. This work aims to describe the development journey of improvements in one of the platform's features, the participatory texts component, from a performance optimization perspective. To that end, research will be conducted on best development practices using the Ruby on Rails framework, and on strategies for leveraging its features that prioritize performance, especially in the design of views. Finally, tests will be carried out to evaluate the impact on the performance of the participatory texts tool.

Key-words: Web Applications. MVC Design Pattern. Ruby on Rails. Performance

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACID	*Atomicity, Consistency, Isolation, and Durability*
API	*Application Programming Interfaces*
DRY	*Don't Repeat Yourself*
CSS	*Cascading Style Sheets*
FTP	*File Transfer Protocol*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
JSON	*JavaScript Object Notation*
MGI	Ministério da Gestão e Inovação em Serviços Público
MVC	*Model View Controller*
ORM	*Object Relational Mapping*
PPA	Plano Plurianual Participativo
RAM	*Random Access Memory*
SASS	*Syntactically Awesome Style Sheets*
SMTP	*Simple Mail Transfer Protocol*
SGPR	Secretaria Geral da Presidência da República
SNPS	Secretaria Nacional de Participação Social
SQL	*Structured Query Language*
UI	*User Interface*
UX	*User Experience*
WYSIWYG	*What You See Is What You Get*
XML	*Extensible Markup Language*

Sumário

1 Introdução

As aplicações *web* desempenham um papel crucial no cenário digital, oferecendo funcionalidades acessíveis através de navegadores *web* sem a necessidade de instalação local. Essas se diferenciam das aplicações *desktop standalone*, que requerem instalação no dispositivo do usuário para operar. Enquanto as aplicações *desktop* são autossuficientes e independem de uma conexão constante com a internet, as aplicações *web* dependem da comunicação com servidores remotos para processar dados e oferecer serviços (TANENBAUM; KLINT; BOHM, 1978).

O funcionamento de uma aplicação *web* envolve um modelo cliente-servidor, onde o cliente (navegador *web*) solicita recursos ou serviços ao servidor, que processa essas solicitações e retorna os resultados. Em termos de arquitetura, as aplicações *web* são divididas em *back-end*, *front-end* e banco de dados. O *back-end* é responsável pelo processamento das solicitações do cliente, o *front-end* lida com a apresentação e interação do usuário, enquanto o banco de dados armazena e recupera os dados necessários para a aplicação (GOUGH; BRYANT; AUBURN, 2021).

Na construção de aplicações *web*, são empregados padrões de projeto para organizar e estruturar o código de forma eficiente. Um desses padrões é o Modelo-Visão-Controlador (MVC), que divide a aplicação em três componentes principais: a *model* (responsável pelos dados e regras de negócios), a *view* (responsável pela apresentação dos dados ao usuário) e a *controller* (gerencia as interações entre o Modelo e a Visão) (MODEL-VIEW-CONTROLLER. . . , 2009). Um exemplo notável de aplicação *web* que segue o padrão MVC é o *Ruby on Rails*, um *framework* construído sobre a linguagem de programação *Ruby* (THE RAILS FOUNDATION, 2023e). Essa abordagem contribui para a modularidade, manutenção e escalabilidade das aplicações *web*.

Apesar dos benefícios proporcionados pelo padrão MVC na construção de aplicações *web*, é importante destacar que esses sistemas podem enfrentar desafios significativos de escalabilidade, especialmente em cenários de alto tráfego e grande quantidade de usuários. Um exemplo notável é a plataforma Brasil Participativo, do governo federal, que atualmente conta com um milhão e seiscentos mil (1.600.000) usuários. Esta plataforma, desenvolvida utilizando o *framework Ruby on Rails* e a *gem/framework Decidim*, tem enfrentado problemas de desempenho devido ao uso intensivo de suas funcionalidades e aumento substancial na demanda e acesso de usuários.

O crescimento do número de usuários pode sobrecarregar a capacidade do sistema em processar simultaneamente um grande número de solicitações, resultando em lentidão, tempo de resposta prolongado e possíveis falhas na execução de funcionalidades críticas.

Não apenas o volume de acessos pode comprometer a correta operação da plataforma, mas também o volume de dados utilizados em suas funcionalidades. Problemas como esses são indicativos da necessidade de otimização e reestruturação da arquitetura para lidar com o aumento de carga e garantir uma experiência de usuário eficiente e sem interrupções. Esses desafios ressaltam a importância de abordagens proativas na identificação e resolução de questões de desempenho, destacando a relevância de estratégias como a implementação eficaz de cache, otimização de consultas ao banco de dados e reorganização da construção do *front-end*.

1.1 Justificativa

A plataforma Brasil Participativo tonou-se a maior instância do *Decidim* em todo o mundo. Mesmo a comunidade responsável pelo desenvolvimento dessa tecnologia reconheceu que o *framework* não foi desenvolvido com pensamento de tomar proporções tão grandes quanto as que se veem no Brasil. Dessa maneira, notou-se a necessidade de intervenção na arquitetura da plataforma para garantir que esta subsistirá mediante a carga de uso atual e futura no país.

1.2 Objetivos

Este trabalho tem como objetivo explorar oportunidades de reestruturação de funcionalidades que apresentam gargalos de desempenho e que ainda não se adequam ao contexto em que a aplicação se encontra: utilização no cenário nacional seguindo diretrizes de plataformas governamentais brasileiras. Será tomada como foco a funcionalidade de textos participativos, uma vez que existe a necessidade de realizar desenvolvimentos para cumprir requisitos funcionais levantados pelos *stakeholders* e de melhorias de desempenho. Entende-se que os resultados e a metodologia deste trabalho, apesar de serem direcionados para a plataforma Brasil Participativo, sejam replicáveis para outras plataformas *web* que utilizam do modelo arquitetural MVC, e, especialmente, *Ruby on Rails*. Os objetivos gerais, contudo, podem ser sumarizados da seguinte maneira:

- Listar e discutir alguns dos principais problemas de desempenho da plataforma;
- Realizar o desenvolvimento de ajustes na ferramenta de textos participativos e de cadastro de novos usuários de acordo com as necessidades dos envolvidos da plataforma;
- Conduzir o desenvolvimento sob o viés de ganho de desempenho;
- Testar a solução adotada e expor os ganhos de desempenho em relação à versão anterior da funcionalidade.

2 Referencial Teórico

Este capítulo possui como propósito expor e definir tópicos pertinentes sobre o assunto abordado nesse trabalho, a fim de trazer maior contexto e entendimento sobre o universo do assunto. Os conceitos que aqui serão abordados são: Conceitos Gerais (Aplicações *Standalone* e Aplicações *Web*), Componentes Gerais de uma aplicação *Web* (*Front-end*, *Back-end* e Banco de Dados), padrão de projeto *Model View Controller* (MVC), *Ruby on Rails*, e escalabilidade e desempenho.

Com a clarificação desses assuntos, será trazida maior fundação teórica para que se siga com a pesquisa e com o caso de estudo, possibilitando assim maior embasamento durante as discussões referentes à problemática a ser estudada.

2.1 Conceitos Gerais

A seção Conceitos Gerais tem como objetivo introduzir conceitos fundamentais no escopo deste trabalho, a saber Aplicações *Standalone* e Aplicações *Web*. A apresentação destes dois tópicos permite com que sejam destacados aspectos pertinentes ao funcionamento de um *software*, bem como a aplicação Brasil Participativo, através da comparação direta desses dois principais modelos de *software* que operam no dia a dia, desde o computador pessoal de um estudante até os grandes *mainframes* do mercado.

2.1.1 Aplicações *Desktop*

Aplicações *Desktop*, também chamadas de aplicações *standalone* são *softwares* que atingem seu propósito de operacionalidade sem a necessidade de estarem conectados à internet, isto é, estas aplicações rodam em um computador ou dispositivo local que não necessariamente estão conectados à *web*. Uma aplicação *desktop* é autocontida, portanto, todo o código e os recursos utilizados por ela se encontram presentes no dispositivo que a executa.

Geralmente, aplicações *desktop* são específicas à plataforma no qual foi projetada para operar, seja Windows, MacOS ou Linux. Dessa maneira, outro conceito pertinente que surge em meio às aplicações *desktop* é a portabilidade. Portabilidade se refere à possibilidade de uma aplicação ser executada em diferentes ambientes com a mínima quantidade de ajustes que for necessária. Um *software* dito portável pode executar, por exemplo, em Windows e MacOS com pouca ou nenhuma adaptação em seu código fonte, enquanto que, um *software* dito pouco portável, pode precisar ser reescrito do absoluto zero ou ter boa

parte do seu código fonte adaptado para rodar em uma plataforma diferente da qual foi primariamente projetado (TANENBAUM; KLINT; BOHM, 1978).

2.1.2 Aplicações *Web*

Aplicações *Web*, diferentemente de aplicações *desktop*, são executadas no dispositivo do usuário através de outra aplicação, o *Web Browser*. Aplicações *Web* funcionam através do uso do modelo cliente/servidor, onde um dispositivo remoto, que possui os recursos da aplicação, os fornece para o dispositivo cliente, que serão exibidos para o usuário pelo *browser*. Para que uma aplicação *web* execute, é necessário que o dispositivo cliente esteja conectado à internet, ou, no mínimo à uma rede que tenha acesso ao dispositivo servidor. Geralmente, a conexão estabelecida no modelo cliente/servidor é através do protocolo *Hypertext Transfer Protocol* (HTTP) ou sua variante *Hypertext Transfer Protocol Secure* (HTTPS) (CONALLEN, 1999).

2.1.2.1 Modelo Cliente/Servidor

No modelo cliente/servidor, diferentemente de aplicações *desktop* convencionais, o processamento de dados é feito no lado servidor. Portanto, na maioria das vezes, o lado cliente fica isento do processamento pesado de dados, sendo responsável apenas por exibir o resultado final ao usuário. Em geral, o fluxo seguido por uma aplicação cliente/servidor é iniciado pelo dispositivo cliente (usuário final), que solicita ao dispositivo servidor algum recurso; o servidor processa a requisição do cliente realizando o processamento necessário, e em seguida, devolve o recurso em um formato que o lado cliente consiga processar e exibir ao usuário final. Essa comunicação pode ocorrer através de diversos protocolos: *File Transfer Protocol* (FTP) para transferência de arquivos, *Simple Mail Transfer Protocol* (SMTP) para envio e recebimento de e-mails, e o já mencionado HTTP.

Cada protocolo dispõe de particularidades que derivam dos seus propósitos, fazendo com que cada um deles seja aplicável em um contexto específico. Entretanto, uma aplicação pode se beneficiar do uso de vários protocolos diferentes para cada uma de suas funcionalidades, isso a depender do propósito de cada um de seus módulos. Uma aplicação *web* moderna, geralmente dispõe de funcionalidades para gerenciar recursos e exibi-los aos usuários utilizando o protocolo HTTP, em contrapartida, a própria entidade usuário pode ser considerada um recurso, onde, cada usuário tem um e-mail vinculado, permitindo que a aplicação envie informações utilizando o protocolo SMTP em algum outro momento oportuno.

A adoção do modelo cliente/servidor pode proporcionar à aplicação *web* diversas características importantes, sendo a principal delas a centralização dos dados no lado servidor. Uma vez que os dados estejam centralizados no lado servidor, máquinas com mais recursos computacionais podem ser alocadas para que o processamento ocorra mais

rapidamente e de forma desacoplada do ambiente do lado cliente, permitindo maior portabilidade da aplicação (OLUWATOSIN, 2014).

2.2 Componentes Gerais de uma Aplicação *Web*

Uma aplicação *web* moderna em geral pode ser vista dividida em duas partes: *back-end* e *front-end*. Ao contrário de aplicações *standalone* convencionais que possuem a lógica de negócio altamente acoplada com a lógica de exibição, aplicações *web* tendem a ter uma divisão mais clara nesse aspecto, permitindo com que a lógica de negócio não necessariamente interfira na renderização. Vale notar ainda a grande semelhança dessa abordagem com o modelo cliente/servidor (GONG et al., 2020). Outro importante componente no contexto de aplicações *web* é o banco de dados, que cumpre o papel de persistir informações da aplicação. A seção Componentes Gerais de uma Aplicação *Web* possui como objetivo introduzir cada um desses conceitos e elucidar seus papeis, trazendo maior clareza para cada um de seus papeis e como estes influenciam no funcionamento da aplicação *web*, e consequentemente no seu desempenho.

2.2.1 *Front-end*

O *front-end* é a parte da aplicação *web* responsável pela exibição dos dados e da interface de usuário. Idealmente, o *front-end* lida apenas com a renderização de páginas, a lógica de *design* da aplicação, sistema de rotas de páginas e recursos estáticos (GONG et al., 2020).

2.2.2 *Back-end*

O *back-end* é a parte da aplicação *web* que lida com toda a lógica de negócio contida no escopo do *software*. Todos os dados, bem como o processamento desses dados é realizado nessa camada. Do ponto de vista do funcionamento da aplicação como um todo, o *back-end* possui como trabalho responder as requisições do usuário que vêm da camada do *front-end*. Uma vez que o usuário pode enviar e requisitar dados, o *back-end* deve ser capaz, através de um conjunto de métodos, processar a requisição e devolver uma resposta no menor tempo possível (ADAM; BESARI; BACHTIAR, 2019).

A comunicação entre *back-end* e *front-end* pode ser realizada de diversas maneiras e com vários protocolos. Uma das formas mais adotadas como mecanismo de comunicação em aplicações *web* modernas é a exposição de funções do *back-end* por meio de *Application Programming Interfaces* (APIs). As APIs podem ser vistas como interfaces de comunicação que visam abstrair a lógica de funcionamento por detrás de uma aplicação, expondo suas funcionalidades através de *endpoints* que podem ser acessados por

outras aplicações sem que a aplicação cliente precise saber como estas operam (GOUGH; BRYANT; AUBURN, 2021).

2.2.3 Banco de Dados

Banco de Dados, nesse contexto, uma forma reduzida do termo Sistema Gerenciador de Banco de Dados, é uma classe de *software* que lida com a manipulação e persistência de dados. Um banco de dados tem como objetivo fornecer aos seus usuários uma fonte de dados centralizada, com qualidade, integridade e segurança (FRY; SIBLEY, 1976). Existem diversos tipos de bancos de dados, porém, são relevantes para este trabalho: os bancos de dados relacionais e os bancos de dados de chave-valor.

2.2.3.1 Banco de Dados Relacionais

Bancos de dados relacionais são bancos cujo dados são guardados no formato de tabelas, também denominadas relações. As tabelas são compostas por colunas e linhas, semelhante a planilhas, onde cada linha representa um registro, denominado tupla, na tabela. Bancos de dados relacionais permitem a criação de relacionamentos entre tabelas, sendo essa a motivação do termo "relacional". A maioria dos bancos de dados permitem manipular e consultar seus dados através do uso da linguagem de consulta *Structured Query Language* (SQL) (JATANA et al., 2012).

2.2.3.2 Transações ACID

No contexto de banco de dados, um importante conceito é o de transações. Uma transação é um agrupamento de operações realizadas pelo banco de dados, sendo a transação um conceito atômico, isto é, a menor instrução que o banco de dados realizará a mando do usuário. Existem características que um banco de dados pode ou não respeitar na sua implementação, estas são: atomicidade, consistência, isolamento e durabilidade (ACID). As propriedades ACID podem ser definidas da seguinte maneira:

- Atomicidade: as operações envolvidas em uma única transação são executadas como uma só, implicando em que, caso uma falhe, a transação como um todo será dada como falha.

- Consistência: os dados presentes no banco de dados devem permanecer consistentes após a execução da transação.

- Isolamento: estados intermediários da transação não são visíveis por outras transações concorrentes, implicando que, uma transação não interferirá em outra.

- Durabilidade: quando uma transação é executada e chega ao fim, seus efeitos são persistentes. Caso ocorra interrupções ou falhas no banco de dados, uma transação completa não será afetada.

(YU, 2009).

2.2.3.3 *PostgreSQL*

Segundo a documentação, o PostgreSQL é um gerenciador de banco de dados relacional *open-source* que implementa e incrementa o padrão da linguagem SQL. O PostgreSQL implementa todos os princípios ACID desde 2001, além de oferecer *plugins* que permitem o uso de outras funcionalidades não convencionais no contexto de banco de dados relacionais (GROUP, 2023).

2.2.3.4 *Redis*

Bancos de dados chave-valor são sistemas gerenciadores de banco de dados que armazenam seus dados de forma não relacional utilizando de chaves e valores. Cada registro consistirá em uma chave e em um valor, podendo este ser armazenado tanto na memória RAM, quanto no disco. Uma das principais características de bancos de dados chave-valor é a sua velocidade quando comparado com bancos relacionais. O *Redis* é um banco de dados chave valor *open-source* extremamente rápido que guarda seus dados em forma persistente no disco ou na memória RAM (CARLSON, 2013).

2.3 Padrão de Projeto *Model-View-Controller*

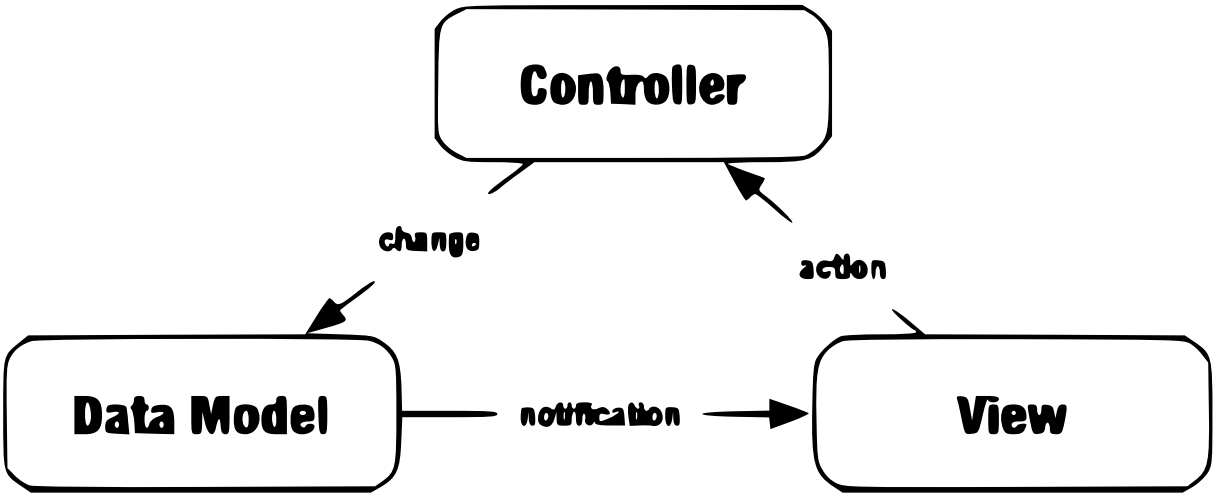
O MVC é um padrão de projeto que é considerado como um dos mais importantes na área da ciência da computação, pois este pode ser interpretado como algo mais próximo de uma filosofia de desenvolvimento de arquitetura de código, do que de um padrão de projeto que resolve instâncias de problemas de um domínio específico. O MVC possui uma grande abertura de interpretação e aplicação, podendo ser adotado desde subsistemas até aplicações inteiras. Essencialmente, o MVC é definido em três entidades: a *model*, a *view*, e a *controller* (MODEL-VIEW-CONTROLLER. . . , 2009).

As próximas subseções têm como objetivo descrever cada uma dessas entidades e qual papel desempenham dentro do contexto do padrão de projeto MVC.

2.3.1 *Model*

A *model* é a entidade do MVC que possui como preocupação encapsular informações do domínio do problema a ser resolvido pela aplicação. Um objeto de dados *model* deve saber guardar, encapsular e abstrair os dados. Vale observar também que

Figura 1 – Comunicação entre as entidades do MVC



Fonte: (MODEL-VIEW-CONTROLLER. . . , 2009)

cada entidade idealmente não deve extrapolar suas funções, portanto, uma *model* deve saber serializar seus dados, mas não deve saber como implementar uma funcionalidade de "salvar como"(MODEL-VIEW-CONTROLLER. . . , 2009).

2.3.2 *View*

Em poucas palavras, a *view* é a entidade do MVC que apresenta informações na tela para o usuário final. Porém, a *view* desempenha muitos outros papeis; na verdade, ela é a ponte entre o usuário e o sistema. Dessa forma, a *view* é responsável por captar o *input* do usuário e transformá-lo em comandos para o sistema processar. Outro aspecto importante é o escopo que uma *view* pode atuar. No desenvolvimento de aplicações com o modelo MVC, objetos de *view* podem ser generalistas ao ponto de saberem renderizar e lidar qualquer com qualquer tipo de texto, imagem ou mídia, ou podem ser específicos ao ponto de saberem renderizar apenas determinados objetos do sistema definidos no código pelo programador (MODEL-VIEW-CONTROLLER. . . , 2009).

2.3.3 *Controller*

A *controller* é a entidade do MVC que implementa as ações do sistema. Como já mencionado, as *models* sabem gerenciar seus dados a nível de persistência e abstração, as *views* são responsáveis por coletar o *input* do usuário para disparar as ações, estas que residem nas *controllers*. Uma ação realizada por uma *controller*, por exemplo, seria o comando "salvar como"de um dado armazenado dentro de uma *model* (MODEL-VIEW-CONTROLLER. . . , 2009). Em geral, a comunicação realizada pelas entidades no modelo MVC pode ser visualizada na Figura 1.

2.4 *Ruby on Rails*

O *Ruby on Rails*, ou simplesmente *Rails*, é um *framework* de aplicação *web* que fornece tudo o que é necessário para a criação de aplicações que seguem o modelo MVC e que utilizam de bancos de dados para armazenamento de suas estruturas e dados. O *Rails* traz para cada uma das entidades do modelo MVC uma interface que provê as funcionalidades necessárias para que estas sejam implementadas em uma aplicação *web*.

O *Rails* traz duas interfaces para que o desenvolvedor consiga criar as *models* no domínio de sua aplicação: o *Active Record* e o *Active Model*. Para as *views*, o *Rails* fornece a *Action View*, que disponibiliza funcionalidades de geração de novas *views* na aplicação. Para as *controllers*, o *Rails* fornece o *Action Controller*. Cada uma dessas interfaces é explorada em mais detalhes a seguir (THE RAILS FOUNDATION, 2023e).

2.4.1 *Active Record*

O *Active Record* é o "M"do modelo MVC no contexto do *Rails*. De fato, o *Active Record* é um padrão de projeto que precede o *Rails*, este foi descrito por *Martin Fowler* em seu livro *Patterns of Enterprise Application*. No contexto de aplicações *Rails*, o Active Record é caracterizado por sua responsabilidade de representar regras de negócio e dados, sendo uma ponte direta entre as *models* da aplicação e o banco de dados, disponibilizando uma interface de *Object Relational Mapping* (ORM) (THE RAILS FOUNDATION, 2023d).

2.4.1.1 *Active Record como uma interface de ORM*

O *Active Record* provê, por padrão, uma interface de comunicação entre os objetos da aplicação com bancos de dados relacionais. Esta interface permite que o programador não necessite de escrever instruções SQL diretamente no código fonte da aplicação para buscar e persistir dados das *models* no banco de dados (THE RAILS FOUNDATION, 2023d). Além de prover esse mecanismo de interação com o banco, o *Active Record* fornece diversas outras funcionalidades que permitem a configuração de validações sobre atributos da *model*, além de prover mecanismos de *callbacks* que são ativados automaticamente para os eventos básicos do ciclo de vida do objeto: criar, salvar, atualizar e destruir.

2.4.2 *Active Model*

O *Active Model* possibilita incorporar funcionalidades do *Active Record* em uma classe *Ruby* pura, sem a intenção de persistir seus atributos no banco de dados. O *Rails* provê maneiras de se incluir separadamente cada uma dessas funcionalidades, permitindo com que a classe herde apenas os comportamentos desejados do *Active Record* (THE RAILS FOUNDATION, 2023c).

2.4.3 *Action View*

As *Action Views* no *Rails* são usadas para renderizar o resultado de uma ação ou requisição processada pelo sistema. Em termos dos fundamentos de uma aplicação cliente/servidor, a função de uma *Action View* é processar as informações a serem retornadas pelo sistema e compilá-las em um formato compreensível pelo usuário. O *Rails* facilita essa renderização por meio de três componentes chave: *templates*, *partials* e *layouts*.

2.4.3.1 *Templates*

Os *templates* atuam como um guia para a exibição das informações resultantes de uma ação. Por padrão, um *template* da *Action View* pode ser renderizado em diversos formatos, destacando-se entre eles: o *HyperText Markup Language* (HTML), o *JavaScript Object Notation* (JSON) e o *Extensible Markup Language* (XML).

2.4.3.2 *Partials*

As *partials*, ou o termo mais completo *template partials*, são exatamente o que o nome sugere: *templates* parciais, ou seja, fragmentos de código que são extraídos para um arquivo separado e que podem ser reutilizados em diversas partes. O propósito das *partials* é dividir o processo de renderização em partes para um melhor controle e permitir a reusabilidade, o que fomenta a filosofia do *Don't Repeat Yourself* (DRY) (THE RAILS FOUNDATION, 2023b).

2.4.4 *Action Controller*

O *Action Controller* é uma parte essencial do padrão de arquitetura MVC, representando a letra "C". Ele gerencia as solicitações, processa dados provenientes da *model* e gera a saída apropriada para exibição ao usuário. Em aplicações *Rails* convencionais, o *Action Controller* recebe a solicitação, interage com a *model* para obter ou salvar dados, e utiliza a *view* para criar uma saída em HTML ou qualquer outro formato necessário. Ou seja, o *Action Controller* atua como intermediário entre os dados das *models* e a apresentação ao usuário por meio das *Action Views*. No *Rails*, é possível definir *actions* dentro das *controllers* por meio de funções que recebem os dados da requisição e realizam as operações necessárias. Cada *action* tem um *template* de *view* correspondente, fortalecendo a cooperação entre o *Action Controller* e a *Action View* (THE RAILS FOUNDATION, 2023a).

2.5 Desempenho e Escalabilidade

A desempenho de uma aplicação *web* pode ser avaliada de várias maneiras. Duas das maneiras mais conhecidas de aferir o nível de desempenho de uma aplicação é através do número de requisições processadas por segundo, também chamado de *throughput* ou *requests/second*, e o tempo gasto pela aplicação para responder a uma requisição. Uma boa aplicação visa manter seu *throughput* alto e seu *response time* baixo, de tal forma que a experiência do usuário seja positiva, a carga de uso seja suportada e a aplicação seja confiável em momentos de pico de acesso, evitando indisponibilidade. Sabe-se que 80% do tempo gasto por uma aplicação no processamento de uma requisição é durante a execução de operações no banco de dados, e na montagem da resposta para o usuário, com a renderização de *templates* e criação do conteúdo compilado (JUGO; KERMEK; MEŠTROVIĆ, 2014).

A aplicação de soluções web em diversas áreas de negócio, como em vendas online, destaca o conceito de escalabilidade em contextos de *software*. Com o aumento da demanda e do acesso de usuários, é crucial que uma aplicação web possa expandir-se mediante a adição de recursos computacionais. Embora não haja um consenso claro sobre a definição de escalabilidade, geralmente é compreendida como a capacidade de uma aplicação aumentar seu *throughput* por meio da incorporação eficiente de recursos de hardware. Essa capacidade é uma propriedade do sistema de *software*, fortemente influenciada pela arquitetura adotada. Se a arquitetura não permitir o aumento do *throughput* em um determinado nível de demanda com a incorporação de mais recursos, ela é considerada não escalável (WILLIAMS; SMITH, 2004).

2.6 Decidim

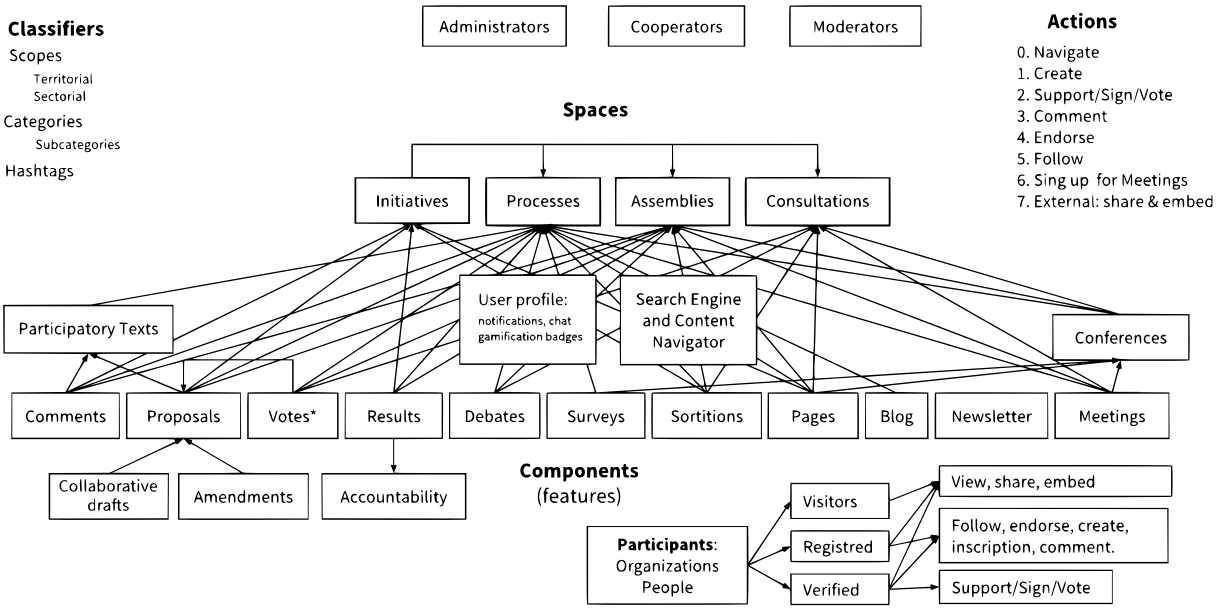
A plataforma Brasil Participativo foi desenvolvida com o *framework Ruby on Rails* utilizando a *gem Decidim*. Segundo a própria documentação disponibilizada pela comunidade de contribuidores do *Decidim*, a *gem* é descrita como um *framework* que visa possibilitar que qualquer pessoa possa criar uma plataforma *web* para ser utilizada como rede política para participação democrática. O *Decidim* fornece a organizações a capacidade de criar processos para planeamento estratégico, orçamento participativo, concepção colaborativa de regulamentos, espaços urbanos e processos eleitorais (METADECIDIM, 2023a).

2.6.1 Arquitetura do Decidim

O *Decidim* possui várias entidades em sua arquitetura. As mais importantes para entendimento de seu funcionamento são os espaços participativos e os componentes, referidos na documentação como *participatory spaces* e *components*, respectivamente. Uma

visão macro das entidades do *Decidim* pode ser visto na Figura 2. O *Decidim* armazena dados de cada uma dessas estruturas utilizando a abordagem padrão de aplicações *Rails like*, isto é, utilizando o *Active Record* e persistindo os dados em um banco de dados relacional, o *PostgreSQL*.

Figura 2 – Arquitetura do *Decidim*



Fonte: (METADECIDIM, 2023b)

2.6.1.1 Participatory Spaces

Os *participatory spaces* são *frameworks* que definem como a participação será realizada, os canais ou meios pelos quais cidadãos ou membros de uma organização podem processar solicitações ou coordenar propostas e tomar decisões. Iniciativas, processos, assembleias e consultas são todos espaços participativos, estes referidos na documentação como *initiatives*, *participatory processes*, *assemblies* e *consultations*, respectivamente. Exemplos específicos incluem: uma iniciativa cidadã para alterar diretamente uma regulamentação; uma assembleia geral ou conselho de trabalhadores; um orçamento participativo, planejamento estratégico ou processo eleitoral; um referendo ou uma convocação para votar "Sim"ou "Não"para mudar o nome de uma organização (METADECIDIM, 2023b).

A nível de código, cada implementação de um novo *participatory space* é feita em um submódulo separado em uma *gem* própria. Uma relação entre cada um dos espaços participativos já existentes por padrão no *Decidim* e o seu respectivo submódulo pode ser vista na Tabela 1.

Tabela 1 – Módulos dos espaços participativos padrões do *Decidim*

Espaço Participativo	Módulo
Assembleias	decidim-assemblies
Consultas ou Eleição	decidim-elections
Iniciativas	decidim-initiatives
Processos Participativos	decidim-participatory_processes

Fonte: (METADECIDIM, 2023b)

Tabela 2 – Módulos dos componentes padrões do *Decidim*

Componente	Módulo
Comentários	decidim-comments
Propostas	decidim-proposals
Debates	decidim-debates
Pesquisas	decidim-surveys
Sorteios	decidim-sortitions
Blogs	decidim-blogs
Reuniões	decidim-meetings

Fonte: (METADECIDIM, 2023b)

2.6.1.2 Components

Os *components* são os mecanismos participativos que permitem uma série de operações e interações entre os usuários da plataforma dentro de cada um dos espaços participativos. No *Decidim* há disponível por exemplo os seguintes componentes: comentários, propostas, debates, pesquisas, sorteios, blogs e reuniões. Outros componentes que se baseiam nos componentes básicos são: textos participativos, responsabilidade e conferências (METADECIDIM, 2023b).

Da mesma forma que ocorre com os espaços participativos, os componentes são implementados através de submódulos em formato de novas *gems*. Na Tabela 2 é possível conferir o nome do módulo de cada componente fornecido por padrão pelo *Decidim*.

2.6.1.3 Participatory Texts

Os *Participatory Texts*, em português: textos participativos, é uma funcionalidade do componente *proposals* que permite o uso de propostas como parágrafos de texto. O seu uso pode ser variado, mas em geral é utilizado para discutir de forma participativa: normativas, planos e outros tipos de texto (METADECIDIM, 2025a).

2.7 Brasil Participativo

A Plataforma Brasil Participativo é uma iniciativa do governo federal voltada para a promoção da participação social. Desenvolvida em *software* livre com o apoio de diversos parceiros, incluindo a Dataprev, a comunidade Decidim-Brasil, o Ministério da Gestão e Inovação em Serviços Públicos (MGI) e a Universidade de Brasília (UnB), ela foi criada sob a responsabilidade da Secretaria Nacional de Participação Social da Secretaria Geral da Presidência da República (SNPS/SGPR).

A plataforma tem como propósito possibilitar que a população contribua ativamente na criação e melhoria das políticas públicas. Uma de suas primeiras iniciativas foi o Plano Plurianual Participativo, assinado pela SGPR e pelo Ministério do Planejamento e Orçamento (MPO). Durante o período de 11 de maio a 16 de julho de 2023, a plataforma permitiu a coleta de propostas da sociedade e a priorização de programas e propostas para o Plano Plurianual (PPA) 2024-2027.

A participação ativa na etapa digital do PPA atingiu mais de um milhão e quatrocentos mil (1.400.000) usuários, conquistando o título de maior experiência de participação social na internet realizada pelo governo federal. A plataforma continuará evoluindo, permitindo que conselhos nacionais criem suas páginas, ministérios realizem consultas públicas e órgãos federais promovam a participação da população na definição de decretos, portarias e outras ações.

Essa abertura à participação digital representa um marco importante para a democracia, possibilitando que cidadãos influenciem diretamente nas decisões governamentais. A Plataforma Brasil Participativo oferece a oportunidade de criação de perfis individuais para facilitar a participação ativa dos interessados (PARTICIPATIVO, 2023).

A plataforma no presente momento se apoia totalmente nas funcionalidades disponibilizadas pela *gem Decidim*, preocupando-se apenas em implementar em sua instância correções de *bugs*, customizações de estilo, comportamento de regras de negócio, e pequenas funcionalidades extras. De fato, a maior parte do código executado na plataforma não é visto em seu repositório oficial, pois este está abstraído dentro da *gem Decidim*. O repositório oficial do Brasil Participativo está disponível no *Gitlab* no endereço: https://gitlab.com/lappis-unb/decidimbr/decidim-govbr. Já o repositório oficial do *Decidim* está disponível no *Github* no endereço: https://github.com/decidim/decidim.

2.8 Considerações Finais do Capítulo

Este capítulo visou esclarecer conceitos relevantes sobre aplicações *web* e seu funcionamento, a fim de permitir ser introduzida de maneira clara a abordagem de otimização de desempenho da plataforma Brasil Participativo, esta desenvolvida utilizando a *gem*

Decidim que utiliza o *framework Ruby on Rails*.

Para compreender os desafios enfrentados pela plataforma e perceber a aplicabilidade das soluções propostas, é fundamental entender o funcionamento de aplicações cliente/servidor e o fluxo de operação de uma aplicação MVC, especialmente no contexto do *framework Ruby on Rails*.

3 Metodologia

Este capítulo explica a metodologia adotada neste trabalho, descrevendo suas etapas, ferramentas utilizadas e os perfis envolvidos na resolução da problemática central. O trabalho segue o formato de estudo de caso, com características de pesquisa aplicada mista.

3.1 Síntese do Referencial Teórico

A fundamentação teórica presente neste trabalho foi feita com objetivo de trazer o conhecimento necessário para completo entendimento dos tópicos envolvidos na solução do problema de desempenho e adequação das ferramentas de textos participativos e de cadastro de novos usuários do Brasil Participativo. Os tópicos envolvidos na execução deste trabalho incluem desde técnicas organizacionais de desenvolvimento de *software* até a abordagens de desenvolvimento sob o rigor de requisitos de desempenho mais restritos, especialmente com o uso do *framework Ruby on Rails*.

3.2 Ferramentas

Para a execução desse trabalho foi adotado o uso de diversas ferramentas:

3.2.1 Visual Studio Code

O Visual Studio Code (VSCode) é um *software* editor de textos de código aberto amplamente utilizado para codificação de sistemas e aplicações. No VSCode é possível trabalhar com o repositório de código da aplicação de forma fácil e integrada, com capacidades de *syntax highlighting*, *auto-formatting* e visualizações do gerenciador de versões Git. Esse editor ainda conta com uma grande gama de extensões da comunidade para facilitar e melhorar a qualidade do desenvolvimento. Uma das extensões utilizadas foi o RubyLSP, que permite a verificação do código em tempo real, capturando erros de sintaxe, além de fornecer um mecanismo de crítica do código considerando as regras de *lint* definidas no projeto.

3.2.2 Git

O Git é uma ferramenta gratuita de código aberto de gestão de versionamento distribuído de código, que pode ser utilizado desde pequenas até grandes aplicações. O Git é amplamente utilizado para gerenciamento de projetos modernos de código.

3.2.3 Yarn

O Yarn é um gerenciador de pacotes de código aberto para gerenciar a distribuição de dependência em projetos JavaScript. Seu objetivo é ajudar no processo de instalação, atualização, configuração e remoção de dependências de pacotes.

3.2.4 RuboCop

O RuboCop é um analisador de estilo de código e formatador *Ruby* que se baseia no guia de estilo de código *Ruby Style Guide*, que é guiado pela comunidade de desenvolvedores. O uso do RuboCop juntamente do RubyLSP acelerou o desenvolvimento e a legibilidade do código, trazendo em tempo real sugestões de refatoração de acordo com os padrões da comunidade.

3.2.5 Nokogiri

O Nokogiri é uma *gem Ruby* usada para manipular documentos HTML e XML. A biblioteca conta com um grande conjunto de funções que auxiliam o processamento de HTML, permitindo com que o desenvolvedor não precise se preocupar com problemas específicos de representação de XML e HTML no *Ruby*.

3.2.6 Jodit

O Jodit é um editor de textos para navegadores. Este editor segue o formato *What You See is What You Get* (WYSIWYG), que basicamente permite com que o usuário insira o texto com a formatação desejada e com a confiança de que essa formatação será persistida e exibida posteriormente.

3.2.7 LibreOffice Writer

O LibreOffice Writer é uma ferramenta de edição de textos de código aberto similar ao Microsoft Word. O LibreOffice Writer permite a escrita de textos, exportação em formato PDF e afins. O LibreOffice Writer é utilizado para trabalhar com os arquivos do formato *docx*, que é utilizado para a primeira confecção dos Textos Participativos pelos usuários.

3.2.8 Webpack e Webpacker

O Webpack é uma ferramenta de empacotamento escrita em JavaScript usada para realizar a transformação de ativos servidos pela aplicação para um formato compreendido pelos navegadores web. Os ativos da aplicação podem incluir desde imagens, arquivos de estilo até código JavaScript. O Webpacker, por sua vez, é uma extensão disponível para o

Ruby on Rails que realiza a interface de comunicação entre a aplicação, que é escrita em Ruby, com o Webpack.

3.2.9 Notion

Em geral, o Notion pode ser definido como uma aplicação *web* utilizada para anotações. O Notion conta com a possibilidade de se trabalhar em conjunto em espaços de trabalho colaborativos. O Notion pode servir como um repositório de ideias e anotações na nuvem, facilitando a recuperação e organização.

3.2.10 ApacheBench

O ApacheBench é uma ferramenta de linha de comando utilizado para realizar testes de carga em aplicações *web*. Essencialmente ela serve para mostrar quantas requisições por segundo a aplicação é capaz de entregar. Sua simplicidade de uso é um de seus pontos fortes.

3.2.11 Draw.io

O Draw.io é uma ferramenta de criação de diagramas online gratuita, permitindo a criação de diagramas clássicos como UML, ER e afins.

3.3 Abordagem

A abordagem deste trabalho pode ser dividida em duas vertentes: a abordagem organizacional, e a abordagem técnica. A abordagem organizacional diz respeito à forma como o trabalho foi conduzido durante o tempo, e a divisão das atividades. A abordagem técnica demonstra como os problemas de *software* foram tratados para se chegar no resultado esperado.

3.3.1 Abordagem Organizacional

A fim de alcançar os objetivos destacados foi adotada a metodologia de desenvolvimento de *software* ágil (*agile*). A rotina de execução de trabalho foi dividida em *sprints*, onde em cada uma delas um conjunto de funcionalidades e atividades serão realizadas. No fim de cada *sprint*, o que foi feito é validado com o time de produto e se necessário com os *stakeholders* do projeto.

3.3.2 Abordagem Técnica

A abordagem técnica será o desenvolvimento orientado à busca por desempenho. Nesse contexto, partes do sistema que devem receber ajustes serão previamente analisadas em termos de desempenho através da checagem das requisições usando da própria ferramenta de *logs* do *framework*. Uma boa ferramenta de logs fornecerá ao desenvolvedor detalhes de tempo gasto por cada componente chave do sistema. Nesse momento é importante que as mesmas condições do ambiente produtivo sejam replicadas no ambiente local de desenvolvimento sempre que possível.

Caso o ajuste em uma determinada parte da funcionalidade envolva a melhoria de desempenho, após a realização das modificações, a mesma análise é conduzida para validar a qualidade dos ajustes e seguir adiante.

3.4 Compilação de Resultados

Como artefato final deste trabalho, será feita a compilação dos resultados obtidos com os ajustes realizados na plataforma do Brasil Participativo sobre o uso das ferramentas de texto participativo e de cadastro de novos usuários. Para demonstrar de forma quantitativa os resultados alcançados, será feita uma tabela comparando os dados de desempenho da aplicação em testes de simulação realizados com a ferramenta ApacheBench.

4 Desenvolvimento

Neste capítulo será apresentado de forma detalhada a maneira como se deu o desenvolvimento dos ajustes nas funcionalidades de textos participativos e de cadastro de novos usuários do Brasil Participativo.

Na Seção 4.1 são levantados alguns dos problemas de desempenho da plataforma. Na Seção 4.2 são apresentados o contexto e a motivação central para a escolha do foco deste trabalho, além de listar de forma clara os objetivos e os requisitos a serem trabalhados.

A Seção 4.3 descreve em pormenores as atividades realizadas em cada *sprint*, abordando aspectos de desenvolvimento e decisões técnicas.

Na Seção 4.4 são apresentadas as apurações dos testes de carga das funcionalidades desenvolvidas, exibindo quantitativamente os ganhos de desempenho. Um resumo do desenvolvimento e dos resultados deste capítulo é apresentado na Seção 4.5.

4.1 Problemática

No ano de 2023 o Brasil Participativo acumulou um milhão e quatrocentos mil usuários únicos durante a execução do PPA. O alto volume de acesso de usuários e dados utilizados no período resultou na exibição dos primeiros problemas de desempenho da plataforma, que foram observados pelos usuários através da indisponibilidade do serviço durante a situação de alta demanda.

Após os incidentes de indisponibilidade, a empresa responsável pela infraestrutura da aplicação no ambiente produtivo, a Dataprev, conduziu uma análise utilizando dos registros das ferramentas de observabilidade adotadas no sistema. Em contato direto com a equipe de desenvolvimento do Lab Livre pelo canal de comunicação do Telegram, eles disponibilizaram um relatório indicando alguns dos problemas da aplicação e suas possíveis soluções.

Os problemas listados foram: sobreutilização do banco de dados e a indisponibilidade do mesmo, falta do uso de índices, *memory leak* e lentidão em páginas específicas. Cada um deles será discutido em mais detalhes a seguir.

4.1.1 Sobreutilização do Banco de Dados e Indisponibilidade

O acesso a páginas de propostas e criação de novos usuários gera enorme utilização do banco de dados. Por exemplo, um dos problemas notados foi o uso do comando SQL `ILIKE` para desambiguação de *nicknames* na plataforma, que ocorre durante o envio de

informações para efetuação do cadastro na plataforma. Foi constatado que o `ILIKE` requer bastante processamento do banco de dados, atingindo tempos na casa de dois segundos.

Requisições que demandam tanto tempo do banco de dados, aliadas à alta demanda de usuários, fizeram com que o PostgreSQL atingisse o limite de conexões em seu *pool* em vários momentos. No ambiente produtivo, o tempo de espera limite para obtenção de uma conexão do *pool* do banco de dados é de cinco segundos, porém diversas requisições retornaram com erro para os usuários por exceder esse tempo limite.

4.1.2 Ausência do Uso de Índices em Consultas Ofensoras na Plataforma

As sete principais consultas mais ofensoras da plataforma poderiam ter seu desempenho melhorado através do uso de índices em suas tabelas no banco de dados. Através da observação do uso da plataforma no ano de 2023, um padrão de consultas recorrentes foi observado. A Dataprev iniciou um trabalho interno para implementação dos índices no banco de dados do ambiente produtivo. A criação desses índices pode ser proposta para a comunidade *Decidim* posteriormente, com o intuito de torná-la padrão no projeto.

4.1.3 Memory Leak do Processo Ruby

Com o passar do tempo, a memória alocada para a execução da aplicação *Rails* aumenta sem dar qualquer indício de recuo, o que implicou em um trabalho extra por parte da equipe de infraestrutura para reiniciar a aplicação sempre que ela atingisse uma determinada quantidade de uso de memória.

4.1.4 Lentidão em Páginas sob Condições Específicas

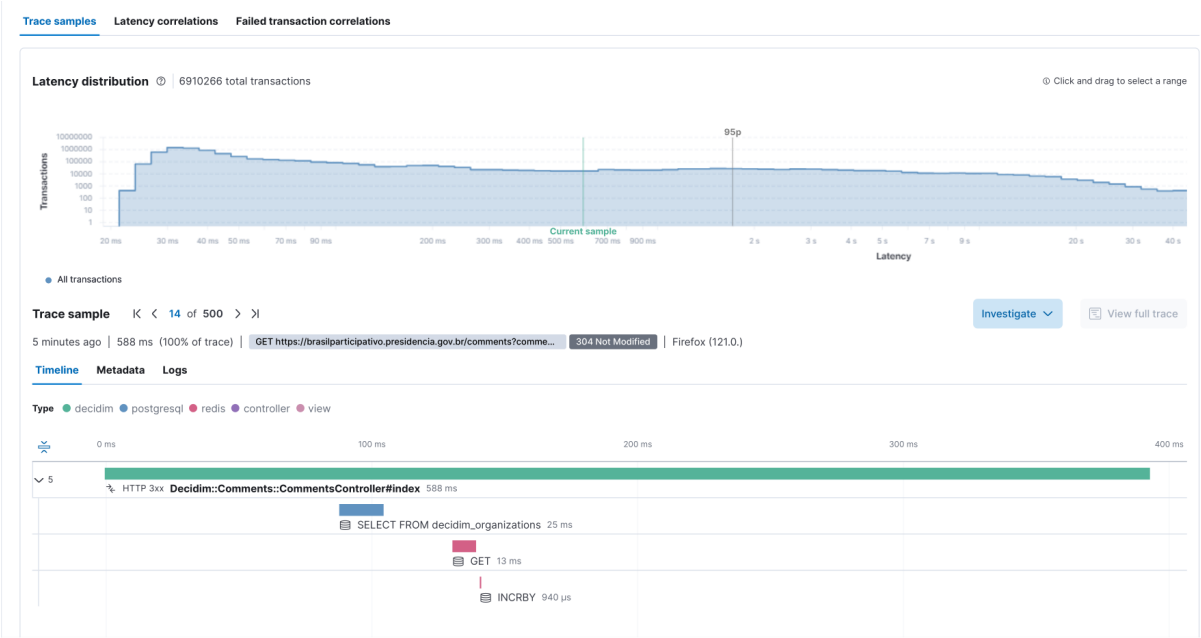
Durante a navegação pelas páginas da plataforma, foi notada lentidão em algumas delas. Essa lentidão ocorria na escala de segundos, e em casos mais graves na escala de minutos. Notavelmente, algumas das páginas que demonstraram baixo desempenho foram: a exibição de propostas com um alto número de comentários, e a exibição de textos participativos que possuíam algumas centenas de parágrafos em sua composição.

Uma das evidências fornecidas pode ser vista na Figura 3. Ela demonstra o desempenho da controladora de comentários da aplicação. A ferramenta utilizada para síntese dessa informação foi o Elastic APM. Contudo, o problema não teve sua causa raiz diagnosticada.

4.2 Condução e Foco do Desenvolvimento

O Brasil Participativo diariamente recebe atualizações solicitadas pelos *stakeholders* da Presidência da República, e elas são executadas pelo time de desenvolvimento do

Figura 3 – Relatório da controladora de comentários gerado pelo Elastic APM



Fonte: Dataprev

Laboratório de Competência em Software Livre da UnB (Lab Livre).

Durante o ano de 2024 surgiu para o Brasil Participativo a necessidade de adequar a funcionalidade de textos participativos vinda do *Decidim*. Esta adequação deveria cobrir os requisitos que foram levantados pelos *stakeholders* e que posteriormente foram trabalhados pelo time de produto do Lab Livre.

Destaca-se o fato de que algumas rodadas de ajustes já tinham sido feitas pelo time de desenvolvimento do Lab Livre no começo do ano de 2024. Entretanto, problemas de desempenho que já eram percebidos na versão original da funcionalidade se mostraram persistentes mesmo após algumas entregas realizadas, dado que o objetivo não era refatoração da ferramenta nem muito menos a busca por ganho de desempenho.

Mediante às oportunidades de intervenção no código fonte da plataforma por solicitação dos *stakeholders*, e munido dos insumos de análise disponibilizados pela Dataprev, o desenvolvimento teve como foco a realização de ajustes na ferramenta de textos participativos do Brasil Participativo. Além disso, o problema no cadastro de novos usuários também será abordado, uma vez que ele foi um dos pontos considerados prioritários pela equipe do Lab Livre.

4.2.1 Objetivos

O problema observado na ferramenta de textos participativos pode ser caracterizado pelo baixo desempenho computacional, onde a renderização da página de visualização

e edição do texto leva minutos para ser feita, a depender da quantidade parágrafos presentes. Outro notável problema de desempenho é a lentidão da página para responder ao comando do usuário, uma vez renderizada. Comandos como o *scroll* e clique do mouse podem levar segundos para serem processados pelo navegador devido ao alto volume de conteúdo presente na página.

Sendo assim, com a correta execução da análise, estudo e ajustes na ferramenta de textos participativos do Brasil Participativo, espera-se que alguns dos resultados sejam:

- Adição de novas funcionalidades na ferramenta de texto participativo;
- Satisfação dos requisitos funcionais levantados;
- Melhora perceptível no desempenho da ferramenta, reduzindo o tempo gasto pelo servidor para processar requisições do usuário;
- Aplicação das melhorias em ambiente produtivo do Brasil Participativo.

Apesar do foco do estudo ser o cumprimento dos objetivos listados acima, entende-se também que um dos frutos deste trabalho é a exposição da metodologia de análise e desenvolvimento de funcionalidades em aplicações *web* MVC com preocupação em desempenho, especialmente *Ruby on Rails*. É esperado que que as etapas de trabalho aqui exploradas sejam reproduzíveis para outros casos semelhantes onde se há a necessidade de realizar o desenvolvimento cumprindo com requisitos de desempenho mais rigorosos.

4.2.2 Organização do Desenvolvimento

A execução das atividades de trabalho ocorreu em múltiplas entregas, onde o período entre uma entrega e outra tinha duração média de um mês.

Foi levantada uma série de requisitos cuja origem foi: as necessidades expressadas pelos *stakeholders* da Presidência, e também as características de uma funcionalidade semelhante ao texto participativo disponível na plataforma Participa+ Brasil.

O time de produto do Lab Livre foi responsável por processar os insumos das solicitações dos usuários e separá-los em requisitos entregáveis, sintetizando o "*backlog*"de trabalho que pode ser conferido na Seção 4.2.3.

A comunicação com o time de produto e de desenvolvimento do Lab Livre foi realizada de forma presencial e por vezes de forma remota. Em média, os rituais de *status report* foram realizados com o gestor do time de desenvolvimento duas vezes por semana. Durante esses encontros eram debatidos impeditivos no cumprimento de requisitos, reajustes de cronogramas e formalização de decisões técnicas.

Ao final de cada *sprint*, foi feita uma *release* contendo as alterações feitas. Essas *releases* foram disponibilizadas no ambiente de testes do Brasil Participativo, o Lab Decide.

Após o lançamento de cada *release*, rodadas de testes de aceitação foram conduzidas pelo time de produto. Como resultado desses testes, eram levantadas atividades de correção de defeitos e ajustes nas funcionalidades já entregues caso houvesse necessidade. Essas necessidades de correção eram adicionadas ao *backlog*, e que na maioria das vezes já eram priorizadas para a próxima *sprint*.

Os *feedbacks* sobre cada *release* eram dados através de reuniões presenciais e remotas com o time de produto, e algumas vezes com o gestor do time de desenvolvimento do Lab Livre. Após o fornecimento de *feedbacks*, o time de produto priorizava o que deveria ser executado na próxima *sprint*.

4.2.3 Requisitos Iniciais

No contexto político que se encontrava a plataforma Brasil Participativo, havia a necessidade de trazer características de uma funcionalidade utilizada com mesmo propósito do texto participativo na aplicação Participa+ Brasil. O Participa+ Brasil também é uma plataforma de participação social do governo federal, e nela o usuário consegue submeter um novo texto copiando o conteúdo de um editor de textos externo, como o Google Docs e o Microsft Word, e colando em um editor de textos próprio da plataforma. Dessa forma, os requisitos iniciais foram definidos como:

- Diminuir o tempo de renderização da página de visualização e de edição do texto participativo;
- Possibilitar a criação de textos participativos através de um editor de textos próprio do Brasil Participativo, contando com o uso do comando copiar e colar, trazendo texto de fontes como o Microsoft Word e Google Docs;
- Refatoração da página de visualização do rascunho (edição de parágrafos), trazendo-a para o mesmo padrão de *design* que a tela de visualização;
- Refatorar a jornada de criação de textos participativos de acordo com a especificação do time de produto.

Além dos requisitos específicos do texto participativo, foi levantada também a necessidade de resolver problemas na funcionalidade de cadastro de usuários:

- Resolver o problema de lentidão e indisponibilidade da funcionalidade de cadastro de usuários exibido durante o PPA.

4.3 Execução do Desenvolvimento

No Brasil Participativo, a ferramenta de textos já vinha recebendo modificações para tentar se adequar aos requisitos que eram acordados junto dos *stakeholders*. No ponto de partida deste trabalho, a ferramenta exibia comportamentos que impactavam negativamente a experiência de usuário, apesar de funcionar.

A tela de edição do rascunho do texto participativo demonstrava bastante lentidão para carregar e operar sobre os parágrafos. O mesmo ocorria com a tela de visualização do texto publicado, onde textos com algumas centenas de parágrafos faziam com que o usuário experimentasse navegações cada vez mais demoradas.

A criação de textos se dava unicamente pelo envio de arquivos no formato `docx`. Esse envio não trazia corretamente as formatações de texto, tais como negrito e itálico.

4.3.1 Etapas de Desenvolvimento

No total houveram quatro *sprints* dedicadas para o desenvolvimento de ajustes no texto participativo, cada uma delas com duração média de um mês. O trabalho foi guiado pelos requisitos específicos levantados para aquele momento de projeto, além dos *feedbacks* fornecidos pelos usuários e pelo time de produto do Lab Livre. Além disso, houve uma quinta *sprint* dedicada para resolver o problema de desempenho na ferramenta de cadastros de usuários.

4.3.2 Sprint 1

A *Sprint* 1 pode ser caracterizada pelo início dos trabalhos, onde houve foco nos três requisitos que eram centrais na proposta de melhoria do componente:

- **Requisito 1 da Sprint 1**: Diminuir o tempo de renderização da página de visualização do texto participativo;
- **Requisito 2 da Sprint 1**: Possibilitar a criação de textos participativos através de um editor de textos próprio do Brasil Participativo, contando com o uso do comando copiar e colar, trazendo texto de fontes como o Microsoft Word e Google Docs;
- **Requisito 3 da Sprint 1**: Refatoração da página de visualização do rascunho (edição de parágrafos), trazendo-a para o mesmo padrão de *design* que a tela de visualização.

Em acordo com o time de produto do Lab Livre, o objetivo central era fazer um *Minimum Viable Product* (MVP) para servir como uma espécie de prova de conceito, onde analisariam os resultados obtidos para eventualmente tomar uma decisão de manter

ou não a mesma estratégia, a fim satisfazer as necessidades levantadas pelos *stakeholders*. As próximas seções revelam detalhes da abordagem utilizada no desenvolvimento de cada requisito que foi listado acima.

4.3.2.1 Requisito 1 da Sprint 1

Aplicações *Ruby on Rails* possuem um sistema de *logs* que permitem que o desenvolvedor obtenha informações da execução de vários tipos e em vários níveis de detalhamento, sendo eles:

- *debug*;
- *info*;
- *warn*;
- *error*;
- *fatal*;
- *unknown*.

Por padrão, aplicações *Rails* executadas em modo de desenvolvimento terão seu nível de *log* como *debug*, e quando executada em modo produtivo será *info* (THE RAILS FOUNDATION, 2025).

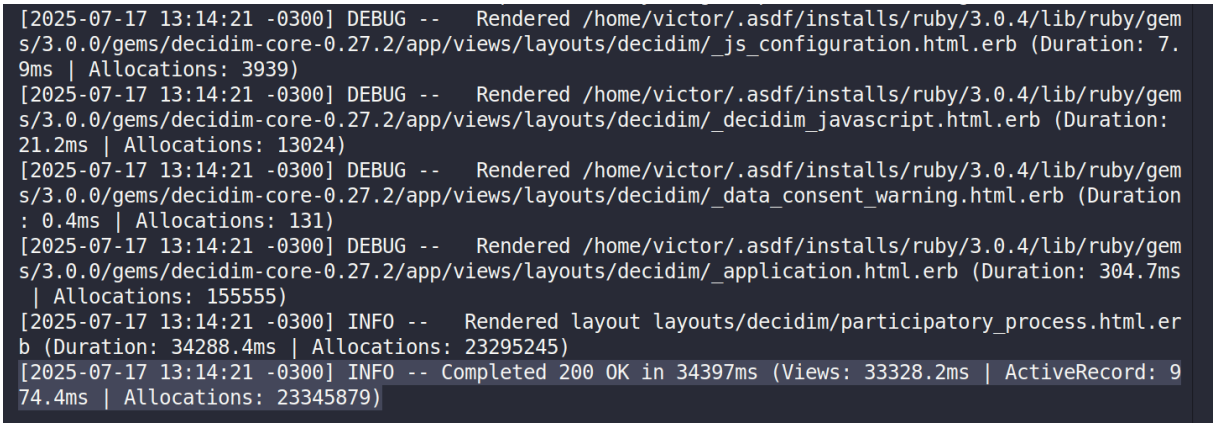
Na condução dessa análise, o Brasil Participativo foi executado em modo desenvolvimento com nível de *logs* como *debug*. Com esse nível de informações, é possível visualizar na saída padrão do Puma, o servidor *web*, quais consultas SQL estão sendo executadas, quais *views* estão sendo renderizadas pelo *Rails* durante a requisição, e os acessos ao ActiveStorage. Alguns desses registros são acompanhados pelo tempo gasto na execução e em alguns casos a quantidade de alocações também é informada.

Para realizar o primeiro diagnóstico, foi criado um texto participativo com em torno de 600 parágrafos, em seguida foi acessada a página de visualização do texto. A Figura 4 mostra os *logs* apresentados pela aplicação no processamento da requisição. Nota-se que a maior parte do tempo foi utilizado na renderização da *views* e suas *partials*, correspondendo a 96,7% do tempo total gasto para processar a requisição.

O Decidim possui um mecanismo de representação para UI chamado de *View Models*, também conhecido como *cells* (em português: células). Uma célula é um objeto que representa fragmentos da UI, podendo conter desde uma página inteira até um container de comentário ou imagem, por exemplo (METADECIDIM, 2025b).

A célula `ParticipatoryTextProposalCell` é usada para realizar a renderização dos parágrafos do texto participativo. Nessa célula, são utilizados dois arquivos de *view*:

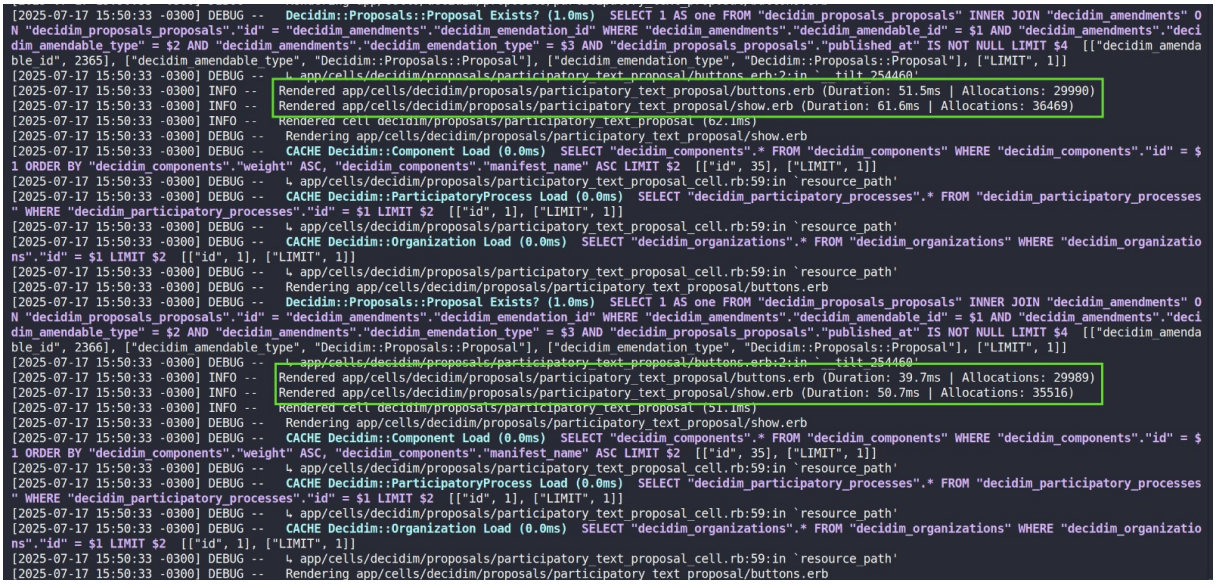
Figura 4 – Captura de tela dos *logs* apresentados pelo *Rails*



Fonte: Próprio autor

`show.erb` e `buttons.erb`. Para realizar a renderização de cada parágrafo, o Decidim fará duas chamadas de renderização de *partials*, uma para o arquivo `show.erb` e outra para o arquivo `buttons.erb`, que é referenciado pelo arquivo `show.erb`. Esse comportamento pode ser percebido através do recorte das mensagens de *log* no processamento da requisição, que é mostrado na Figura 5.

Figura 5 – Captura de tela dos *logs* evidenciando a utilização de *partials*



Fonte: Próprio autor

No repositório oficial do *Ruby on Rails* existe uma *issue* que relata problemas de desempenho com o uso de *partials* em contextos em que há aninhamento, destacando que a renderização de HTML planos acontece com maior agilidade. A *issue* pode ser acessada em <https://github.com/rails/rails/issues/41452>.

Dessa forma, a primeira tentativa de ajustes realizada foi a substituição do uso

das células na renderização de textos participativos pela renderização *inline* do conteúdo do parágrafo diretamente no arquivo principal da *view*:

`decidim/proposals/proposals/participatory_texts/participatory_text.html.erb`

Após realizar a substituição, foi observado novamente os *logs* do processamento da requisição de renderização da página do texto participativo. O tempo de processamento total da requisição diminuiu de 34.397 milissegundos para 1.253 milissegundos, representando uma redução de 96,36%, e que foi considerada como suficiente para o requisito. Uma captura de tela dos *logs* é exibida na Figura 6.

Para elucidar a alteração, dado o enfoque deste trabalho na otimização de desempenho, o código fonte será demonstrado em capturas de tela. Contudo, as alterações estão disponíveis no repositório oficial do Brasil Participativo no *Merge Request* em <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/560>.

A versão anterior à modificação pode ser vista na Figura 7, e a versão ajustada para melhoria de desempenho pode ser vista na Figura 8.

Figura 6 – Captura de tela dos *logs* após substituição das células

```
t]
[2025-07-17 16:22:26 -0300] DEBUG --   Rendered /home/victor/.asdf/installs/ruby/3.0.4/lib/ruby/gems/3.0.0/gems/d
ecidim-core-0.27.2/app/views/decidim/shared/_authorization_modal.html.erb (Duration: 0.1ms | Allocations: 34)
[2025-07-17 16:22:27 -0300] DEBUG -- [Webpacker] Everything's up-to-date. Nothing to do
[2025-07-17 16:22:27 -0300] DEBUG -- [Webpacker] Everything's up-to-date. Nothing to do
[2025-07-17 16:22:27 -0300] DEBUG --   Rendered /home/victor/.asdf/installs/ruby/3.0.4/lib/ruby/gems/3.0.0/gems/d
ecidim-core-0.27.2/app/views/layouts/decidim/_js_configuration.html.erb (Duration: 9.2ms | Allocations: 3940)
[2025-07-17 16:22:27 -0300] DEBUG --   Rendered /home/victor/.asdf/installs/ruby/3.0.4/lib/ruby/gems/3.0.0/gems/d
ecidim-core-0.27.2/app/views/layouts/decidim/_decidim_javascript.html.erb (Duration: 20.6ms | Allocations: 7957)
[2025-07-17 16:22:27 -0300] DEBUG --   Rendered /home/victor/.asdf/installs/ruby/3.0.4/lib/ruby/gems/3.0.0/gems/d
ecidim-core-0.27.2/app/views/layouts/decidim/_data_consent_warning.html.erb (Duration: 0.2ms | Allocations: 127)
[2025-07-17 16:22:27 -0300] DEBUG --   Rendered /home/victor/.asdf/installs/ruby/3.0.4/lib/ruby/gems/3.0.0/gems/d
ecidim-core-0.27.2/app/views/layouts/decidim/_application.html.erb (Duration: 191.7ms | Allocations: 67692)
[2025-07-17 16:22:27 -0300] INFO --   Rendered layout layouts/decidim/participatory_process.html.erb (Duration: 1
187.5ms | Allocations: 1072762)
[2025-07-17 16:22:27 -0300] INFO -- Completed 200 OK in 1253ms (Views: 1139.4ms | ActiveRecord: 54.1ms | Allocati
ons: 1099366)
```

Fonte: Próprio autor

4.3.2.2 Requisito 2 da Sprint 1

Para cumprir com o segundo requisito, foi adicionado no Brasil Participativo, o editor de textos Trix que é do tipo WYSIWYG. O Trix é um editor de textos pensado para o uso de envio de mensagens, escrita de comentários, artigos e listas. Ele foi criado pela 37signals e está presente em algumas aplicações modernas, como por exemplo o Basecamp, que é uma ferramenta de gestão de projetos também da 37signals. A escolha desse editor de texto se deu pela sua facilidade de integrar com uma aplicação *Rails*, e sua arquitetura baseada em eventos; permitindo com que customizações sejam feitas com pouco código.

Figura 7 – Código fonte da *view* antes da substituição da célula

```
<div class="br-container-lg">
  <%= render partial: "decidim/shared/component_announcement" %>
  <h1 class="participatory-text-title">
    <%= render_participatory_text_title(@participatory_text, @proposals.first) %>
  </h1>
  <% if @proposals.count > 0 %>
  <div class="participatory_texts_buttons">
    <%= render partial: "decidim/shared/export_modal" %>
  </div>
  <% end %>
  <ul class="participatory-text-list" id="proposals-text">
    <% @proposals.each do |proposal| %>
      <% if !proposal.comments_section? && !proposal.is_hidden %>
        <%= cell("decidim/proposals/participatory_text_proposal", proposal, context: { current_user: current_user }) %>
      <% end%>
    <% end %>

    <% if participatory_text_comments(@proposals).present? %>
      <%= comments_for participatory_text_comments(@proposals) %>
    <% end %>

    <% @proposals.each do |proposal| %>
      <% if proposal.comments.length > 0 && !proposal.comments_section? %>
        <%= comments_for proposal, {participatory_text_initial_page: true} %>
      <% end %>
    <% end%>
  </ul>
</div>

```

Fonte: Próprio autor

Um requisito implícito no requisito 2 era a necessidade de inserir imagens no texto. Por padrão, o Trix posiciona a imagem no texto como um documento "pendente", portanto, é necessário que a aplicação guarde a imagem remotamente e forneça ao Trix um *link* permanente.

Para realizar isso, a *controller* `Decidim::Govbr::AttachmentsController` foi criada no Brasil Participativo. Esta *controller* possui como papel: receber um arquivo, salvá-lo no `storage` da aplicação e devolver a URL permanente para aquele arquivo em uma resposta no formato JSON. Essa *controller* expôs a *action* `create` na rota `/attachments`. A *action* verifica antes de processar a requisição se o emissor é um usuário administrador logado. O código da *controller* é exibido na Figura 9.

Outro desafio relacionado à inserção do texto através do comando copiar e colar é separar corretamente cada parágrafo de texto. O *input* do usuário, apesar de parecer apenas texto, na verdade é enviado para o *backend* como uma `string` HTML, como por exemplo:

`<h1>`Título do Texto`</h1> <div>` Parágrafo! Aqui vem o conteúdo do parágrafo do usuário.`<br>` Aqui é outro parágrafo.`<br> </div>`

Para processar o *input* HTML e transformar em uma estrutura mais flexível de se trabalhar, foi adotada a *gem* Nokogiri. Com o uso do Nokogiri, o texto HTML é transformado em uma estrutura de dados semelhante a uma árvore, onde cada *tag* HTML

Figura 8 – Código fonte da *view* ajustado para desempenho

```
<div class="br-container-lg">
  <%= render partial: "decidim/shared/component_announcement" %>
  <h1 class="participatory-text-title">
    <%= render_participatory_text_title(@participatory_text, @proposals.first) %>
  </h1>
  <% if @proposals.count > 0 %>
  <div class="participatory_texts_buttons">
    <%= render partial: "decidim/shared/export_modal" %>
  </div>
  <% end %>
  <ul class="participatory-text-list" id="proposals-text">
    <% comment_icon = icon "comment-square", class: "icon--small", aria_label: t("comments", scope: "
decidim.proposals.participatory_text_proposal.buttons"), role: "img" %>

    <% @proposals.find_each do |participatory_text| %>
      <% participatory_text_title = translated_attribute(participatory_text.title) %>
      <% participatory_text_body = translated_attribute(participatory_text.body) %>

      <li class="participatory-text-item">
        <%= content_tag(participatory_text.is_interactive ? :a : :div, href: current_component_engine_router.proposal_path(
participatory_text)) do %>
          <h5><strong>$ <%= participatory_text.position %></strong></h5>

          <% if participatory_text.image? && participatory_text.image_attachment.attached? %>
            <%= content_tag(:img, nil, src: participatory_text.image_attachment.url) %>
          <% elsif participatory_text.table? %>
            <%== table_from_str(participatory_text_body) %>
          <% elsif translated_attribute(participatory_text_body).include?("<table>") %>
            <%== decidim_sanitize_editor(simple_format(participatory_text_body, {}, sanitize: false)) %>
          <% else %>
            <%== decidim_sanitize_editor(simple_format(participatory_text_body)) %>
          <% end %>

          <% if component_settings.comments_enabled? %>
            <%= link_to current_component_engine_router.proposal_path(participatory_text, anchor: "comments") do %>
              <button class="br-sign-in secondary small" type="button" data-trigger="login">
                <i class="fa-solid fa-message"></i>
                <span class="d-sm-inline">
                  <%= participatory_text.comments_count %>
                </span>
              </button>
            <% end %>
          <% end %>
        <% end %>
      </li>
    <% end %>

    <% if participatory_text_comments(@proposals).present? %>
      <%= comments_for participatory_text_comments(@proposals) %>
    <% end %>

    <% @proposals.each do |proposal| %>
      <% if proposal.comments_count > 0 && !proposal.comments_section? %>
        <%= comments_for proposal, { participatory_text_initial_page: true } %>
      <% end %>
    <% end%>
  </ul>
</div>

```

Fonte: Próprio autor

Figura 9 – Código fonte da *controller* `Attachments`

```ruby
# frozen_string_literal: true

module Decidim
  module Govbr
    class AttachmentsController < ApplicationController
      before_action :authenticate_user!

      def create
        return render(json: { success: false }) unless current_user.admin?

        @blob = ActiveStorage::Blob.create_and_upload!(
          io: params[:file],
          filename: params[:file].original_filename,
          content_type: params[:file].content_type
        )

        render(
          json: {
            signed_id: @blob.signed_id,
            url: Rails.application.routes.url_helpers.rails_blob_url(@blob, only_path: true)
          },
          status: :ok
        )
      end
    end
  end
end
```

Fonte: Próprio autor

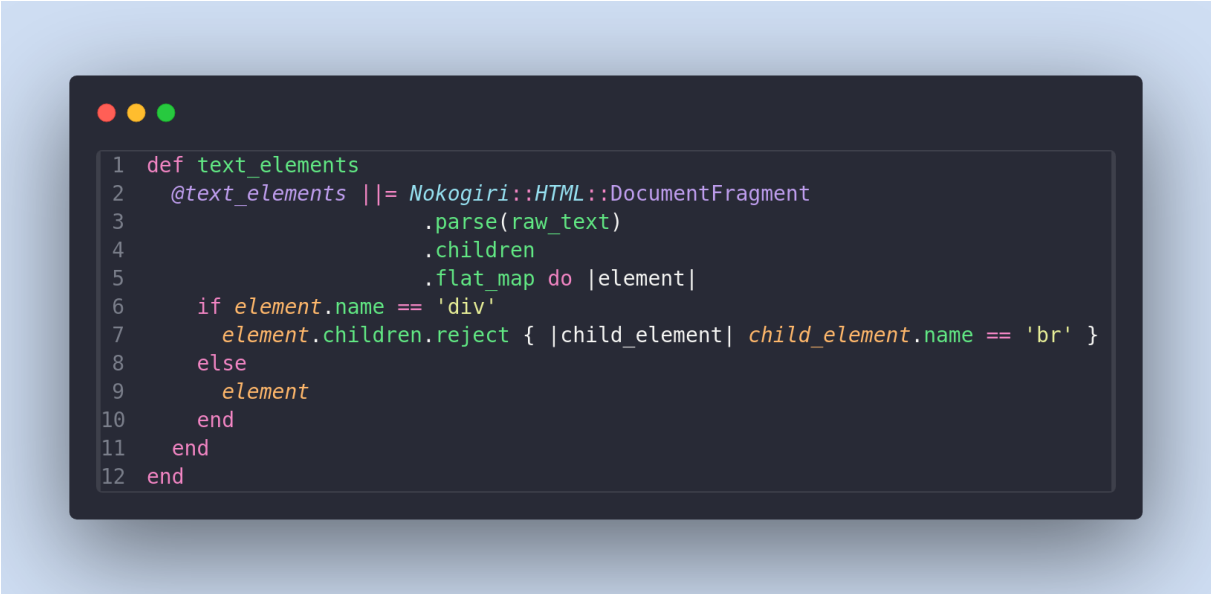
se torna um nó, com seus elementos filhos, que podem ser outras *tags* ou texto puro, estando disponíveis no atributo `#children`.

A fim de separar o texto, depois de realizar o *parsing* do HTML, cada elemento (*tag*) de primeiro nível era considerado um parágrafo. Entretanto, quando o usuário digita um parágrafo e logo em seguida uma quebra de linha para digitar outro parágrafo, o Trix envolvia ambos os parágrafos em uma única *tag* `<div>`, separando os parágrafos por quebra linha com a *tag* `<br>`. Para contornar isso, foi adicionada uma verificação no nó, e caso este seja um elemento do tipo `<div>`, ele não será utilizado como um parágrafo, mas sim os seus elementos filhos diretos, desde que não seja uma *tag* `<br>`. O código de divisão pode ser conferido na Figura 10.

Depois de resolvido o problema de divisão dos parágrafos, havia a gestão das imagens presentes no texto. Dado que cada imagem é enviada para o `storage` do *Rails* já no momento em que o usuário está digitando o conteúdo do texto no *frontend*, era necessário de alguma maneira saber vincular cada imagem a cada parágrafo.

Um parágrafo na ferramenta de textos participativos é um objeto do tipo `Proposal`, que também é utilizado para representar propostas na plataforma. No Brasil Participa-

Figura 10 – Método para realizar a divisão dos parágrafos do input do usuário



```ruby
def text_elements
  @text_elements ||= Nokogiri::HTML::DocumentFragment
                       .parse(raw_text)
                       .children
                       .flat_map do |element|
    if element.name == 'div'
      element.children.reject { |child_element| child_element.name == 'br' }
    else
      element
    end
  end
end
```

Fonte: Próprio autor

tivo, foi feito pela equipe de desenvolvimento do Lab Livre uma customização que permite que cada proposta tenha relacionamento com nenhum ou vários arquivos. Sendo assim, uma das etapas de pré-processamento do *input* é descobrir as imagens que já estão no `storage` do *Rails* e vinculá-las a cada proposta (parágrafo).

A primeira etapa é feita através da extração do atributo `signed_id` dos nós Nokogiri que são do tipo `figure`. Na integração de envio de arquivos do Trix para o `storage` do *Rails*, além da URL da imagem, a *controller* `Attachments` envia o atributo `signed_id`, que funciona como um identificador para o arquivo. Esse atributo é vinculado ao elemento de imagem do Trix, e fica disponível no atributo `data-trix-attachment`.

Depois de extrair o `signed_id`, a próxima etapa é a criação de fato dos parágrafos. Como dito, cada parágrafo é salvo em um objeto do tipo `Proposal`. Nesse momento o parágrafo é salvo podendo seu tipo ser:

- Título (enumerador `section`), caso seja `<h1>`;
- Subtítulo (enumerador `sub_section`), caso seja `<h2>` e `<h3>`;
- Imagem (enumerador `image`), caso seja `<figure>`;
- Artigo (enumerador `article`), caso seja qualquer outra *tag* HTML.

Caso o parágrafo seja do tipo imagem, o atributo `signed_id` é usado para localizar o arquivo no `storage` e realizar o vínculo.

4.3.2.3 Requisito 3 da Sprint 1

A página de visualização do rascunho refatorada, que é a página utilizada para fazer edições no texto submetido, tinha como propósito parecer o máximo possível com a tela de visualização do texto já publicado. Segundo o time de produto do Lab Livre, esta abordagem traria melhor experiência ao usuário administrador, que no momento da edição do rascunho teria ideia de como o texto se pareceria com o que o usuário vê.

Sendo assim, a primeira ideia foi a de extrair o padrão visual da *view* utilizada na renderização do texto, e utilizá-la em outra página dedicada para a edição do texto. Ora, o Decidim segmenta o sistema em três contextos diferentes: o `system admin`, `admin` e `public`, cada um deles tem o seu próprio `namespace` dentro do código-fonte, fazendo uma separação entre os estilos, controle de permissões, *views* e *controllers*.

A página de visualização do texto fica no contexto `public`, enquanto a página de edição do texto fica no contexto `admin`. Para que a transição entre a edição e a visualização do texto fosse irrisória, uma das decisões foi a de implementar a página de edição refatorada no contexto `public`. Dessa maneira, o *layout* e a lógica de renderização dos parágrafos, já com a otimização de desempenho, foi separada em uma *partial*, que é renderizada tanto pela página de visualização quanto pela de edição.

A *partial* criada (`_lightweight_text_render.html.erb`) faz verificações sobre o estado do texto, e o usuário que a está acessando. Caso o usuário seja um administrador e o texto ainda não tenha sido publicado, botões de ações aparecerão para manipular o texto. A página de edição do texto com os seus botões de ação pode ser vista na Figura 11.

A página de edição conta com cinco ações possíveis:

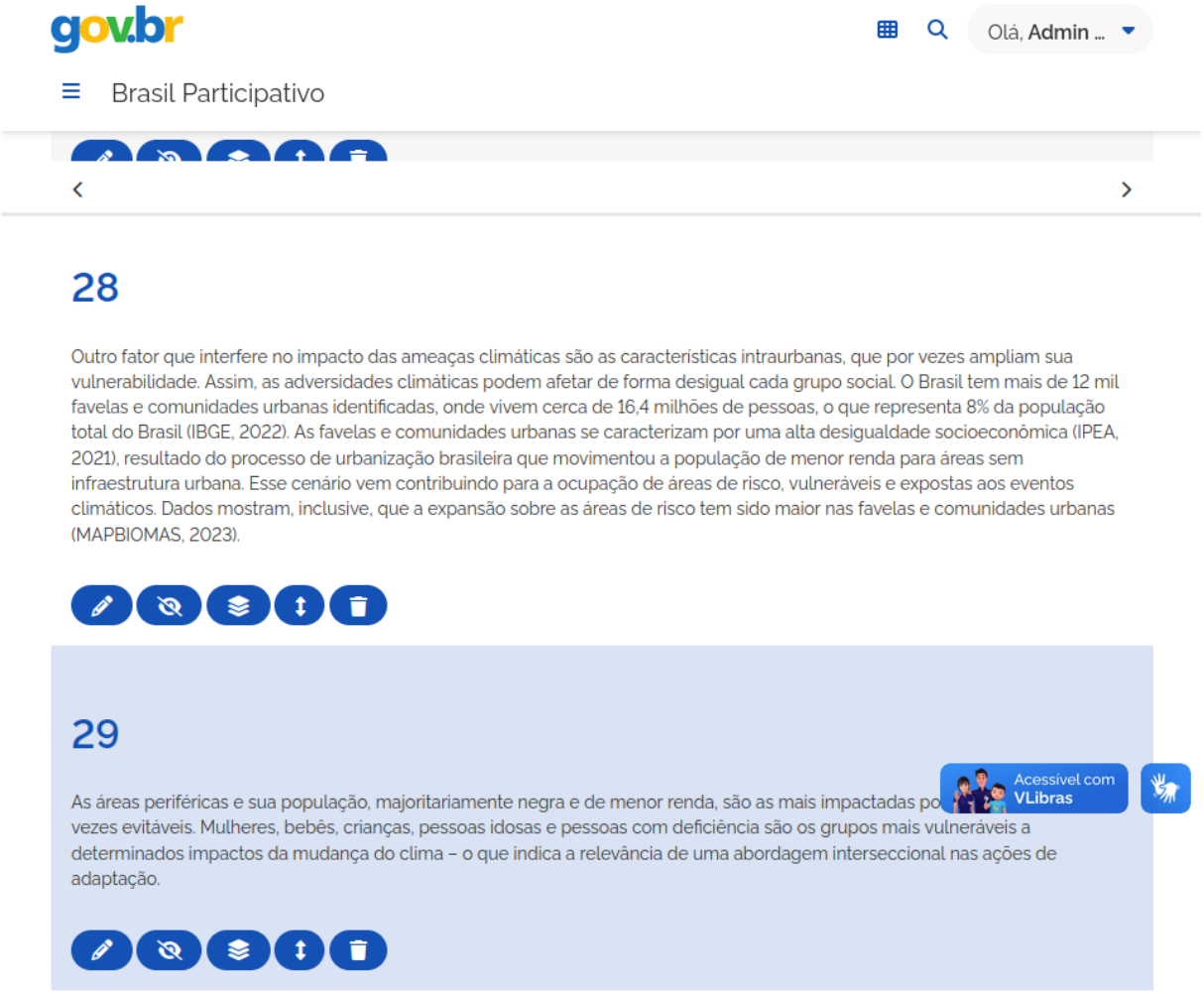
- Editar conteúdo do texto;
- Ocultar parágrafo;
- Mesclar parágrafos;
- Mover parágrafo;
- Remover parágrafo.

O *Merge request* contendo as alterações descritas acima pode ser encontrado em <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/561>.

4.3.3 Sprint 2

A *sprint* 2 foi caracterizada pela realização de ajustes e correção de comportamentos inesperados que surgiram durante os testes de aceitação das funcionalidades entregues

Figura 11 – Tela de edição de textos refatorada para o escopo `public`



Fonte: Próprio autor

na *sprint* 1, conduzidos pelo time de produto. Os trabalhos começaram assim que a lista de ajustes necessários foi levantada, e pode ser sumarizada nos seguintes requisitos:

- **Requisito 1 da Sprint 2**: Corrigir problema onde um parágrafo que possua trechos em itálico e negrito é erroneamente dividido em mais de um parágrafo;
- **Requisito 2 da Sprint 2**: Corrigir problema de imagens quebradas quando copiadas do Google Docs ou Microsft Word;

4.3.3.1 Requisito 1 da Sprint 2

Alguns parágrafos que possuíam itálico e negrito em seu conteúdo, eram erroneamente divididos em múltiplos parágrafos. Por exemplo o *input* HTML:

seria dividido em cinco parágrafos:

```
<h1>Título do Texto</h1>
<div>
Parágrafo! <strong>Aqui vem o conteúdo</strong>
do parágrafo do usuário.<br>
Aqui é outro parágrafo.<br>
</div>
```

- `<h1>Título do Texto</h1>`;
- `Parágrafo!`;
- `<strong>Aqui vem o conteúdo</strong>`;
- `do parágrafo do usuário.`;
- `Aqui é outro parágrafo.`.

Isso acontece devido ao fato de que no Nokogiri, texto puro também é tratado como um nó na árvore do documento XML/HTML. Ao fazer a tratativa de separar as `<div>` exibida na Figura 10, o método `#children` separará o primeiro parágrafo em três, pois entende-se que a *tag* `strong` é um elemento independente na árvore. A Figura 12 ilustra o comportamento.

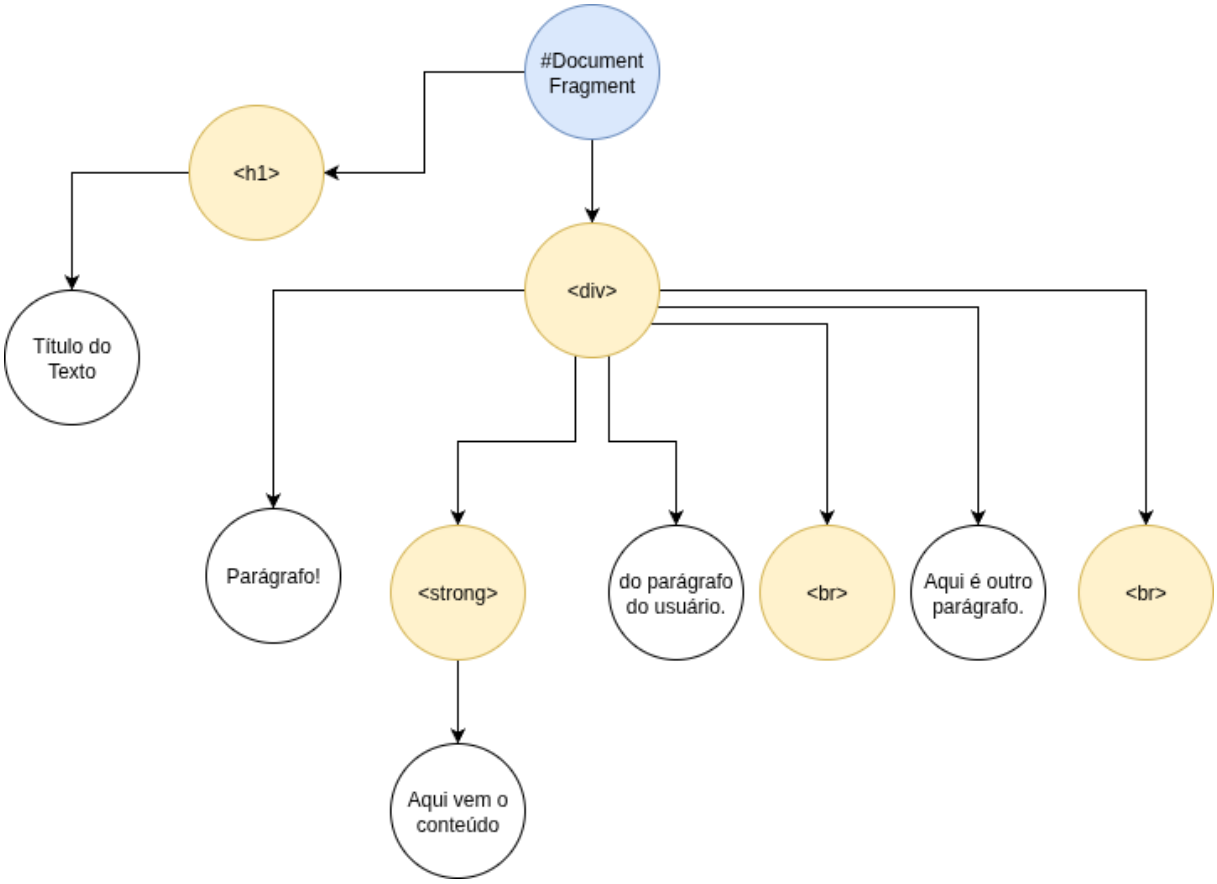
Para resolver esse problema, foi criado um método para fazer a extração dos parágrafos com base nos filhos diretos de uma `<div>`, onde dois ou mais filhos podem ser considerados pertencentes ao mesmo parágrafo. O critério para dividir os nós filhos entre mais de um parágrafo é o aparecimento de um elemento `<br>` entre eles. A Figura 13 mostra como a divisão ocorre na visualização de árvore. Caso seja encontrado um elemento `<div>`, seus filhos serão agrupados até que se encontre o elemento divisor (`<br>`). Os elementos são analisados da esquerda para direita. Esse método foi inserido diretamente na classe `Nokogiri::XML::Nodeset` na inicialização da aplicação. A Figura 14 mostra a implementação e o uso.

4.3.3.2 Requisito 2 da Sprint 2

Ao utilizar o comando copiar em editores de texto como o Google Docs, na maioria das vezes o conteúdo da imagem em si não era copiado, mas sim um *link* para onde ela está guardada. Para esses casos, a imagem (`<img>`) já era inserida no texto com o seu atributo `src` preenchido, e não era automaticamente enviada para o `storage` do *Rails*, pois o seu conteúdo (`blob`) sequer estava lá.

Para resolver esse problema, uma das possibilidades seria fazer uma requisição para a fonte da imagem, para em seguida enviá-la para o `storage` usando a *controller* `Attachments`. Mas, essa alternativa tinha dois problemas: barreiras com o *Cross-Origin Resource Sharing* (CORS), e disponibilidade limitada do recurso.

Figura 12 – Diagrama representando a árvore do HTML após o *parsing* do Nokogiri



Fonte: Próprio autor

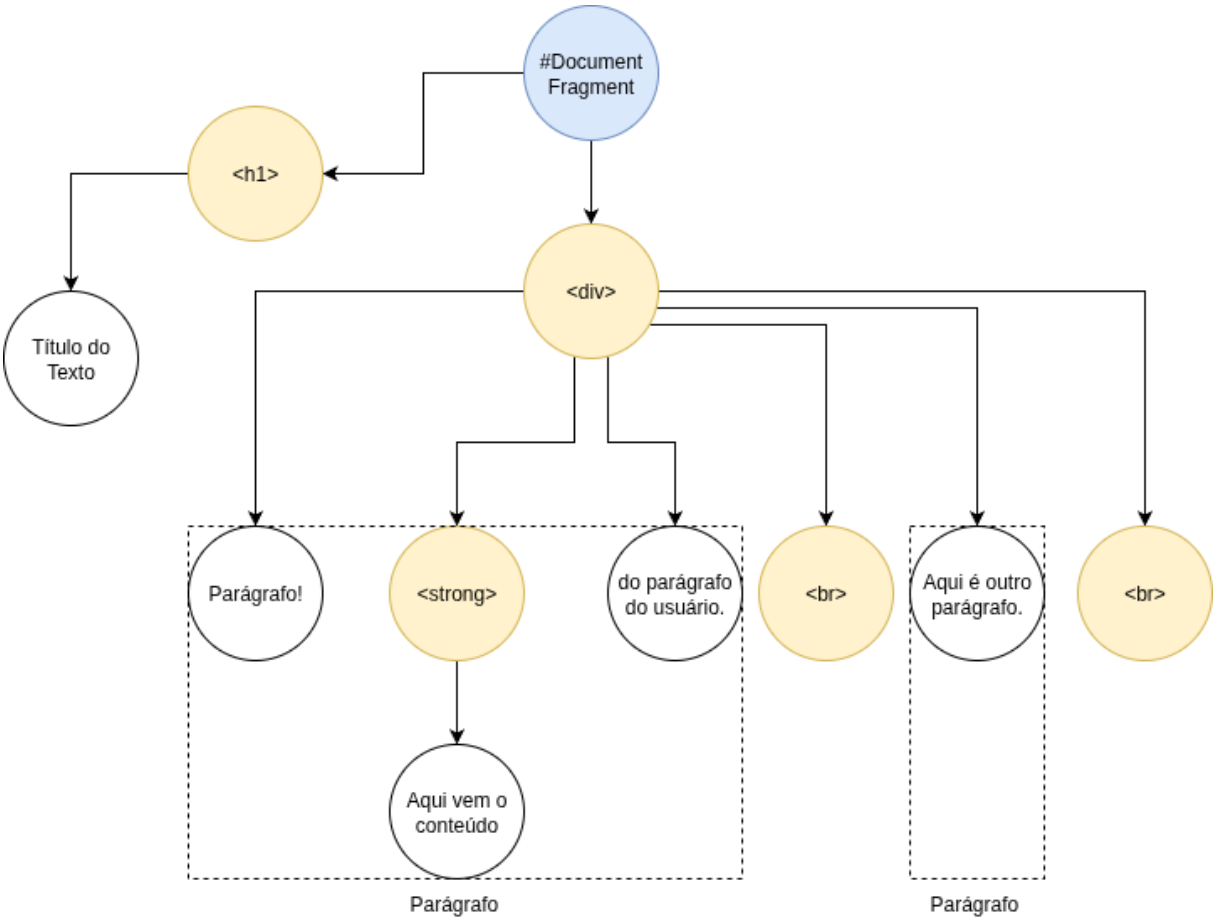
CORS é um mecanismo para controle de requisições entre domínios distintos quando a requisição é feita pelo *browser*, fazendo com que nem sempre o Brasil Participativo tenha permissão para baixar imagens de outras fontes automaticamente (MDN WEB DOCS, 2025a).

Outra possibilidade seria deixar o elemento HTML que representa a imagem intocado e referenciando o *link* externo mesmo depois de salvar o parágrafo no banco de dados. Porém, algumas fontes possuem mecanismos de *throttle* para impedir que uma determinada origem faça requisições demais, além de que não há garantias de que o recurso estará disponível a qualquer instante que se queira buscá-lo.

Portanto, para garantir a consistência na inserção de imagens no texto, o usuário não conseguirá mais inserir imagens através do comando copiar e colar. Será necessário manualmente baixar a imagem no computador e inseri-la no corpo do texto, seja arrastando, seja utilizando a funcionalidade de upload de arquivos do Trix.

Para realizar esse comportamento, foi escrito uma nova customização no Trix para observar o evento `trix-before-paste`, que é disparado antes do conteúdo ser inserido no *input*. Caso seja detectado algum elemento de imagem no conteúdo colado, este será

Figura 13 – Diagrama representando o agrupamento de elementos filhos separados por `<br>`



Fonte: Próprio autor

substituído por uma mensagem de aviso, além de alertar o usuário com uma mensagem do próprio navegador de que ele deve seguir outro procedimento para realizar a inserção daquela imagem. Na Figura 15 é exibido o código JavaScript que realiza essa verificação.

4.3.4 Sprint 3

A *sprint* 3 foi o momento em que houve maior foco em questões de usabilidade e experiência do usuário. Nessa *sprint* houve a participação do time de *design* do Lab Livre para auxiliar na definição dos elementos visuais e na organização do fluxo de uso da ferramenta. Em geral, os requisitos da *sprint* foram definidos como:

- **Requisito 1 da Sprint 3**: Adequar os componentes visuais da tela de visualização e edição do texto de acordo com o protótipo elaborado pelo time de *design*;
- **Requisito 2 da Sprint 3**: Ajustar o fluxo de uso da funcionalidade de acordo com a jornada mapeada pelo time de produto.

Figura 14 – Definição e uso do método `split`

```ruby
# Dentro do parser do input HTML
def text_elements
  @text_elements ||= Nokogiri::HTML::DocumentFragment
                     .parse(raw_text)
                     .children
                     .flat_map do |element|
    if element.name == 'div'
      element.children.split('br') # Alterado de .reject para .split
    else
      element
    end
  end
end

# Dentro do initializer
Nokogiri::XML::NodeSet.class_eval do
  def split(element_name)
    result = []
    last_cut_pos = -1

    each_with_index do |element, idx|
      next unless element.name == element_name

      if last_cut_pos != idx - 1
        new_node = Nokogiri::XML::Node.new('p', document)
        new_node.children = slice((last_cut_pos + 1)..(idx - 1))
        result << new_node
      end

      last_cut_pos = idx
    end

    if last_cut_pos < length - 1
      new_node = Nokogiri::XML::Node.new('p', document)
      new_node.children = slice((last_cut_pos + 1)..(length))
      result << new_node
    end

    result
  end
end

```

Fonte: Próprio autor

Figura 15 – Código JavaScript para remover imagens inseridas através do copiar e colar

```
document.querySelector('trix-editor').addEventListener('trix-before-paste', function(event) {
  const html = event.paste.html;
  const tempDiv = document.createElement('div');
  tempDiv.innerHTML = html;
  const images = tempDiv.querySelectorAll('img');

  if (images.length > 0) {
    alert("
Parece que você está tentando colar imagens além de texto. É necessário que você as baixe pri
meiro no seu computador e as arraste manualmente para o editor de textos do Brasil Participat
ivo!
")

    images.forEach(img => {
      const placeholder = document.createElement('div');
      placeholder.className = 'image-placeholder';
      placeholder.innerHTML =
`⚠️ Por favor, arraste a imagem baixada no seu computador para cá. Em seguida apague esta lin
ha`
;
      img.replaceWith(placeholder);
    });

    event.paste.html = tempDiv.innerHTML;
  }
});
```

Fonte: Próprio autor

4.3.4.1 Requisito 1 da Sprint 3

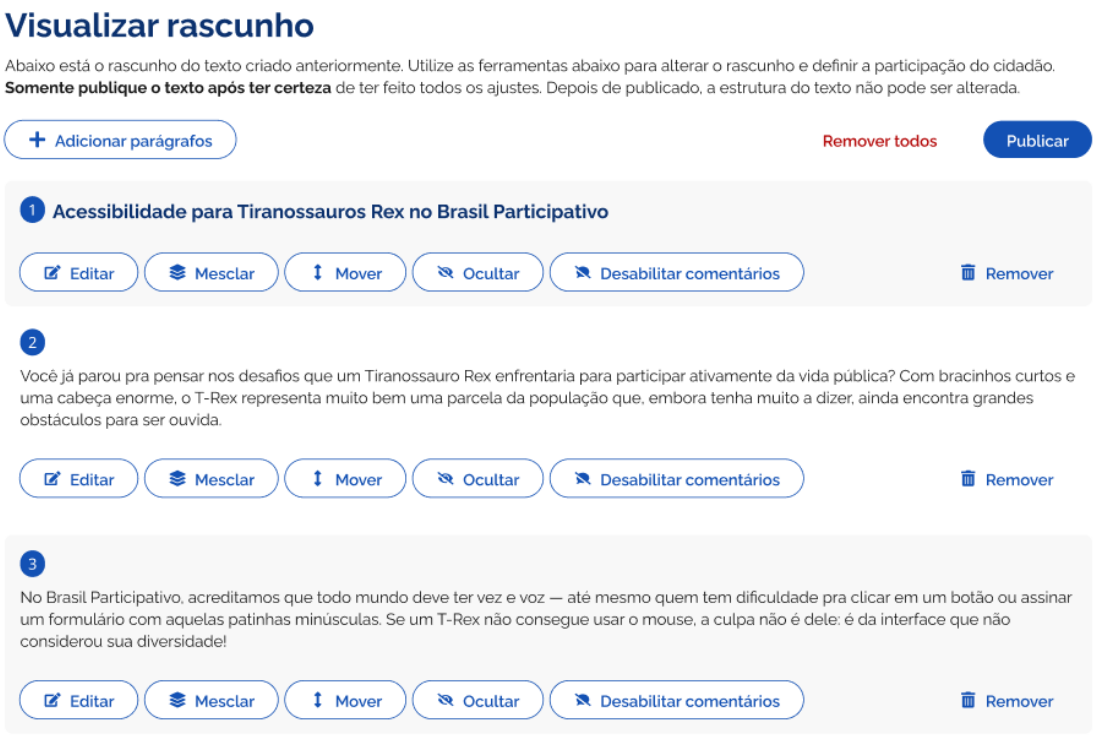
Na Figura 16, pode-se conferir uma captura de tela do protótipo criado pelo time de *design* na ferramenta Figma. Com intuito de preservar o nível de desempenho obtido pela refatoração realizada na *sprint* 1, as modificações visuais foram majoritariamente feitas através do uso de CSS, com pouquíssima alteração no HTML.

A estilização foi feita principalmente através de regras aplicadas a classes HTML utilizando o *Syntactically Awesome Style Sheets* (SASS), uma linguagem de extensão para CSS. As regras de estilo foram codificadas no arquivo `participatory-text.scss`.

Para utilizar da estilização por classes, na própria *partial* `_lightweight_text_render`, para cada parágrafo a ser renderizado, é instanciado um `Array` para guardar quais classes HTML são aplicáveis naquele contexto. Para isso, são feitas verificações sobre o tipo, visibilidade e estado do parágrafo, além de checar se a *partial* está sendo renderizada no modo edição ou visualização. Na Figura 17 é demonstrado de forma resumida o código utilizado.

Com essa abordagem, foram adicionadas apenas verificações com *if/else* para dinamicamente alterar as classes do elemento `<li>` que envolve o conteúdo do parágrafo. Essa

Figura 16 – Protótipo da tela de rascunho



Fonte: Time de *design* do Lab Livre - UnB

Figura 17 – Exemplo de código fonte para estilização dinâmica com classes

```
1  <% dynamic_classes = [] %>
2  <% dynamic_classes << "section" if participatory_text.section? %>
3  <% dynamic_classes << "sub-section" if participatory_text.sub_section? %>
4  <% dynamic_classes << "edit" if editing %>
5  <% dynamic_classes << "hidden-for-users" if participatory_text.is_hidden %>
6  <% dynamic_classes << "published" if participatory_text.published? %>
7
8  <%= content_tag(:li, class: "participatory-text-item #{dynamic_classes.join(' ')}") do %>
9    <% # Conteúdo do parágrafo vem aqui... %>
10 <% end %>
```

Fonte: Próprio autor

estratégia evita o uso desnecessário de *partials*, que poderia trazer novamente gargalos de desempenho para a *view*, e evita também duplicação de código, algo que as *partials* se propõe a resolver.

Um comparativo do antes e depois do visual da tela de edição do texto pode ser conferido através das figuras: Figura 18, que mostra o estado anterior às modificações, e Figura 19 que mostra o estado após as modificações.

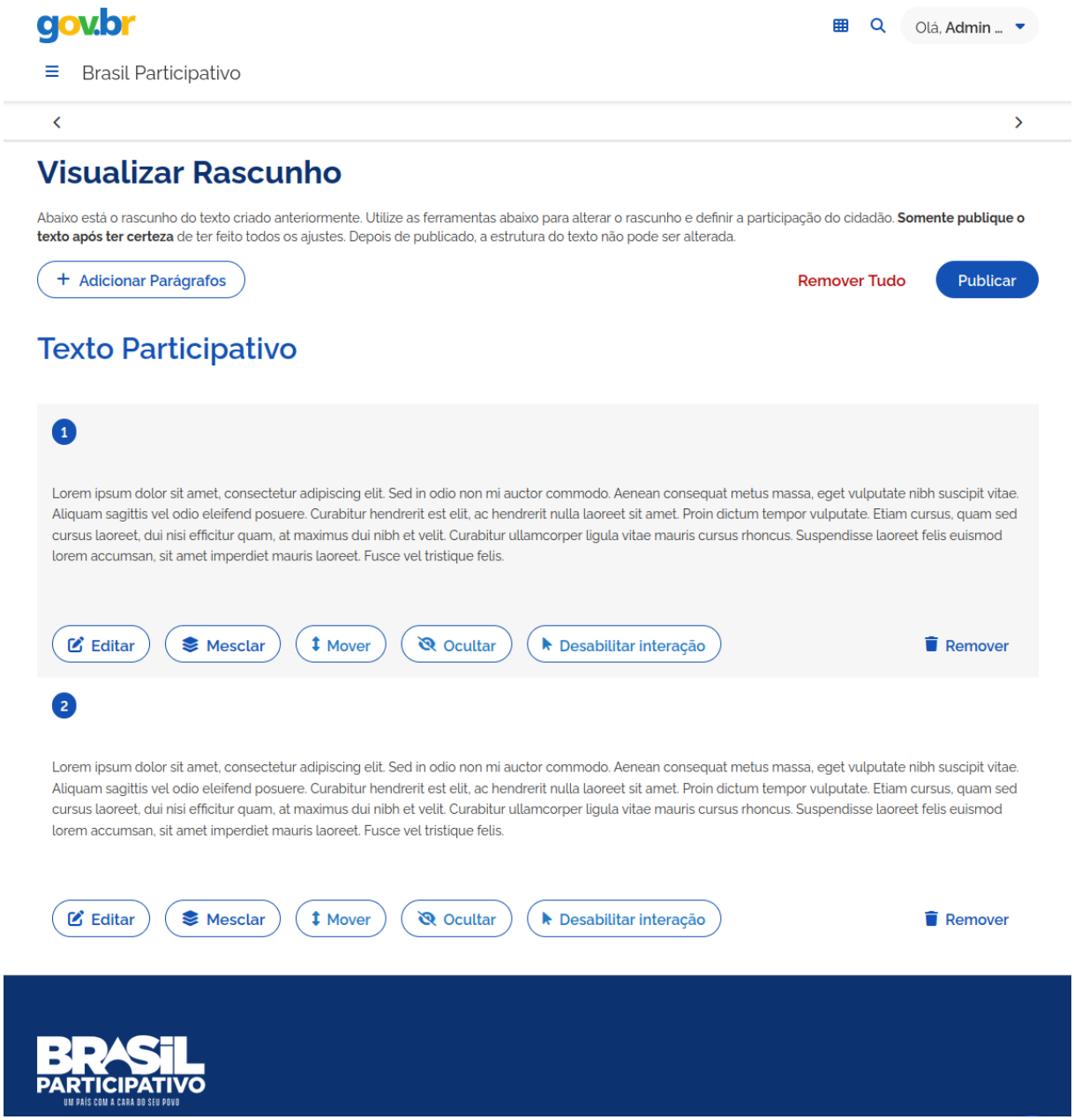
Figura 18 – Tela de edição de textos antes da adequação visual

Fonte: Próprio autor

4.3.4.2 Requisito 2 da Sprint 3

O fluxo de uso da ferramenta de textos participativos ainda possuía sobreposições com a versão anterior à adequação da página de edição do texto. Por conta de a tela de edição estar junta da tela de visualização do texto no contexto `public`, várias transições ocorriam entre contexto `admin` para contexto `public`, que são visualmente diferentes, durante a jornada de uso da funcionalidade. Para que o usuário administrador acessasse a

Figura 19 – Tela de edição de textos depois da adequação visual



Fonte: Próprio autor

página de edição do texto, era preciso primeiramente navegar pela página do componente no painel *admin*, para só então clicar em um botão para o redirecionar para a tela desejada.

Sendo assim, foram realizados ajustes para que o usuário seja direcionado diretamente para páginas que façam sentido para o estado atual do texto participativo. Essas modificações foram feitas apenas com o uso de redirecionamentos nas *controllers* com o uso do método `redirect_to`. Dessa forma, esse ajuste não teve impacto no requisito de desempenho, dado que bastava fazer uma verificação sobre o estado do texto participativo e realizar ou não o *redirect* para outra página.

O trabalho realizado na *sprint* também contou com a participação do time de desenvolvimento do Lab Livre. As alterações podem ser conferidas na íntegra nos *Merge Requests*:

1. <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/603>
2. <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/612>
3. <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/613>
4. <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/615>
5. <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/628>

4.3.5 Sprint 4

Após todos os ajustes já realizados no *frontend* para se adequar aos padrões visuais expostos no protótipo feito pelo time de *design* do Lab Livre, a quarta *sprint* foi caracterizada por um único requisito:

- **Requisito 1 da Sprint 4**: Adicionar suporte à criação e edição de tabelas nos parágrafos do texto participativo.

No repositório oficial do Trix há uma *issue*, já fechada, onde usuários questionam se há planos de adicionar o suporte a tabelas; até a data deste trabalho não houve indícios de que mantenedores do projeto irão atuar nessa demanda. A *issue* mencionada pode ser acessada em <https://github.com/basecamp/trix/issues/539>.

4.3.5.1 Requisito 1 da Sprint 4

Com a limitação do Trix em relação ao uso de tabelas, anteriormente foi acordado com o time de produto do Lab Livre que estas seriam inseridas no texto no formato de imagens. Entretanto, em reuniões de *feedback* com os *stakeholders*, viu-se que era impraticável em termos de experiência do usuário e usabilidade proceder dessa maneira.

Foram feitas reuniões de descoberta com o time responsável pela plataforma Participa+ Brasil, onde existe funcionalidade semelhante ao texto participativo, e que existe o suporte a tabelas. Como saída dessa reunião foram sugeridos alguns editores de texto, dentre eles: Jodit, CKEditor 5 e o TinyMCE. No Participa+ Brasil é utilizado o editor Jodit. Com base na premissa de que alguns usuários poderiam migrar do Participa+ Brasil para o Brasil Participativo, o Jodit foi cotado como potencial escolha.

Para garantir que o Jodit haveria compatibilidade com o Brasil Participativo, foi realizado um teste localmente, trocando o Trix pelo Jodit na página de inserção de conteúdo para criação de texto participativo. O teste foi realizado inserindo na própria *view* da página *links* para o JavaScript e o CSS do Jodit, conforme descrito na própria documentação do editor. Para validar a viabilidade de uso do editor, foram feitas várias provas inserindo tabelas, imagens e textos formatados através do comando de copiar e colar, todos os textos vindos do Google Docs, Microsoft Word e LibreOffice Writer. Foram manualmente persistidos alguns parágrafos que continham tabelas, e posteriormente exibidos com sucesso na tela de visualização do texto participativo.

Com o sucesso desses testes, o desenvolvimento finalmente seguiu com foco total para a implantação do Jodit no Brasil Participativo.

Como próximo passo, era preciso, da mesma forma como foi feito para o Trix, achar formas de persistir as imagens inseridas no editor no `storage`. Um dos problemas com a persistência de imagens realizada pelo Trix era de que o *upload* da imagem era feito no momento em que o usuário a inseria no texto, estando à mercê de problemas de rede, ou impedindo que o texto fosse escrito *offline* para ser enviado posteriormente.

Em termos de desempenho em relação ao uso de armazenamento, havia também o agravante de que o usuário poderia inserir imagens e apagá-las logo em seguida. Dessa forma a imagem nunca seria associada a um parágrafo de texto, permanecendo órfã no `storage` da aplicação até que fosse removida manualmente ou por alguma automação.

O Jodit possui a opção `uploader.insertImageAsBase64`, que guarda o conteúdo da imagem no formato de Data URL na própria estrutura do HTML. Data URL é uma maneira de embutir o conteúdo de arquivos de forma *inline* em documentos (MDN WEB DOCS, 2025b). Com isso, o atributo `src` de *tags* `<img>` é preenchido com o próprio conteúdo da imagem codificado em Base64, ao invés de ser preenchido com um *link* para local onde ela estaria guardada.

Dessa maneira, ao invés de tentar guardar a imagem no `storage` logo que for inserida no texto, pode-se guardá-la apenas quando o usuário submeter o texto para criação, resolvendo então o problema de desempenho no uso de armazenamento.

Sendo assim, o próximo passo seria, então, fazer o *parsing* do *input* HTML e realizar a divisão dos parágrafos. Diferentemente do Trix que por padrão inseria novos

parágrafos envoltos de *tags* `<div>`, da forma como foi configurado, no Jodit os parágrafos são envoltos de *tags* `<p>`, e a cada quebra de linha uma nova *tag* `<p>` é criada para servir de container para o texto. O único problema é a formatação de alguns textos quando copiados do Google Docs e do Microsft Word. Alguns utilizam *tags* `<p>` para servir de container para *tags* `<img>`, além de em alguns momentos o texto de vários parágrafos estar envolto de uma única *tag* `<strong>`. Esse comportamento foi classificado neste trabalho como "*tags* modificadoras de texto sendo utilizadas como `<div>`", e guiou o desenvolvimento do *parser* para suportar a migração para o Jodit.

A versão final do *parser* de textos participativos executa três passos:

1. Substituir sequência de caracteres de espaço em branco ou não imprimíveis por um único espaço no *input* HTML ainda em formato `string`;
2. Realizar a transformação para a estrutura do Nokogiri;
3. Separar parágrafos removendo aqueles que estivem em branco.

A Figura 20 demonstra o código fonte da etapa de *parsing* do *input*.

Figura 20 – Código fonte da função que realiza o *parsing* do HTML vindo do Jodit

```ruby
def parsed_text_elements
  return @parsed_text_elements if @parsed_text_elements

  raw_text.gsub!(/\s+/, ' ')
  parsed_text = Nokogiri::HTML::DocumentFragment.parse(raw_text)
  @parsed_text_elements = split_paragraphs(parsed_text).reject { |paragraph| empty_paragraph?(paragraph) }
end
```

Fonte: Próprio autor

O método `split_paragraphs` é um método também criado à mão para satisfazer os casos de bordas classificados como "*tags* modificadoras de texto sendo utilizadas como `<div>`". Para resolver esse problema, foi utilizada uma estratégia recursiva para percorrer a árvore de nós do Nokogiri e determinar quais nós pertencem ao mesmo parágrafo ou não.

A verificação começa no nó raiz, que é o próprio `DocumentFragment`. Se o nó que está sendo avaliado nesse momento for uma `<div>` ou o `DocumentFragment`, recursivamente chama a verificação para todos os seus nós filhos retornados pelo método `children`, pegando o resultado e compilando em um `Array` achatado, isto é, um `Array` que não possui elementos do tipo `Array` dentro de si. A mesma coisa também é feita caso o nó seja

um modificador de texto sendo usado como `<div>`. Caso contrário, é retornado um `Array` envolvendo o próprio nó apenas. A ideia central por trás desse algoritmo é a de eliminar elementos que sejam uma `<div>` ou que ajam como uma `<div>`, e escolher ramos da árvore para serem parágrafos. O código fonte dessa verificação pode ser visto na Figura 21.

Figura 21 – Código fonte do método que realiza a divisão de parágrafos do Jodit

```ruby
def split_paragraphs(node)
  return [] unless node

  if node.name.downcase.in?(%w(div #document-fragment)) || text_modifier_being_used_as_div?(node)
    node.children.flat_map { |child| split_paragraphs(child) }
  else
    [node]
  end
end
```

Fonte: Próprio autor

O algoritmo para definir se um determinado nó está sendo usado como `<div>` também é de caráter recursivo. A análise começa no primeiro nó passado como parâmetro. Primeiro se verifica se o nó é uma *tag* modificadora de texto, se não for, o algoritmo para e a resposta é falso. Caso contrário, o algoritmo segue em frente e verifica se o nó atual possui filhos, se não tiver, ele para e a resposta é falso. Caso contrário, o algoritmo segue em frente e passa a avaliar todos os filhos diretos do nó que está sendo avaliado. Recursivamente o algoritmo irá avaliar para cada filho se este é uma *tag* modificadora sendo usada como `<div>`. Se ao menos um de seus filhos for uma *tag* modificadora sendo usada como `<div>`, o próprio nó também será considerado, e o algoritmo retorna verdadeiro.

No contexto de uso do Jodit, considera-se como *tags* modificadoras de texto: `<p>`, `<title>`, `<span>`, `<br>`, `<text>`, `<strong>`, `<b>`, `<em>`, `<i>`, `<u>`, `<s>`, `<del>`, `<mark>`, `<small>`, `<sup>`, `<code>`, `<pre>`, `<img>` e `<a>`. É importante saber que a *tag* `<img>` está entre elas pois uma imagem tem o mesmo valor semântico que um texto puro dentro de um parágrafo, nesse contexto. Isto é, uma imagem não é fator determinante para dividir um parágrafo. O código fonte dessa verificação pode ser visto na Figura 22.

Depois de implementado a funcionalidade de copiar e colar texto, foram realizadas algumas experimentações para checar o nível de desempenho. Primeiro foi submetido um texto copiado do LibreOffice Writer com 106.055 caracteres. Na Figura 24 é exibida uma captura de tela dos *logs* da aplicação demonstrando o tempo gasto pela requisição. Nota-se que foram gastos 18 segundos para processar a requisição, sendo 4 segundos gastos pelo banco. O texto foi dividido em 381 parágrafos

Figura 22 – Código fonte do método que verifica se modificador de texto está sendo usado como `<div>`

```ruby
def text_modifier_being_used_as_div?(node)
  return false unless tag_is_a_text_modifier_or_img?(node)
  return false if node.children.empty?

  used_as_div = false
  node.children.each do |child|
    used_as_div ||= if tag_is_a_text_modifier_or_img?(child)
                      if child.children.present?
                        text_modifier_being_used_as_div?(child)
                      else
                        false
                      end
                    else
                      true
                    end
  end

  used_as_div
end

def tag_is_a_text_modifier_or_img?(node)
  node.name.downcase.in?(
  %w(p title span br text strong b em i u s del mark small sup code pre img a))
end
```

Fonte: Próprio autor

Figura 23 – Tempo de execução para criação de texto com 106.055 caracteres e com 381 parágrafos

```
[2025-07-19 02:18:57 -0300] DEBUG --   ↳ app/commands/decidim/proposals/admin/create_participatory_text_from_copy_and_paste.rb:
44:in `create_proposals_from_parsed_text'
[2025-07-19 02:18:57 -0300] INFO -- [WISPER] Decidim::Proposals::Admin::CreateParticipatoryTextFromCopyAndPaste#1118160 publish
ed ok to Decidim::EventRecorder#[] with no arguments
[2025-07-19 02:18:57 -0300] INFO -- Redirected to http://localhost:3000/admin/participatory_processes/programas/components/36/m
anage/participatory_texts
[2025-07-19 02:18:57 -0300] INFO -- Completed 302 Found in 18317ms (ActiveRecord: 4418.6ms | Allocations: 9994863)
```

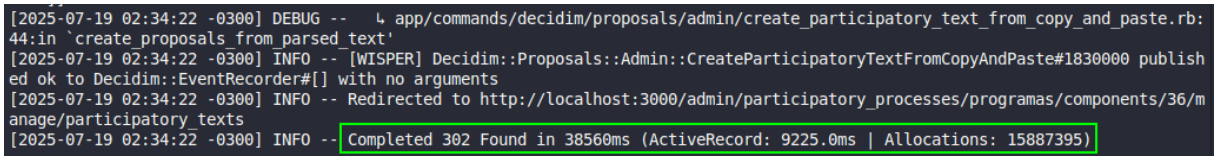
Fonte: Próprio autor

Depois foi submetido um texto, também copiado do LibreOffice Writer, com 157.883 caracteres. Na Figura 24 é exibida uma captura de tela dos *logs* da aplicação demonstrando o tempo gasto pela requisição. Nota-se que foram gastos 38 segundos para processar a requisição, sendo 9 segundos gastos pelo banco. O texto foi dividido em 647 parágrafos.

Um dos problemas arquiteturais da ferramenta de textos participativos do Decidim é o uso de propostas como objeto para persistência dos parágrafos. No Decidim as propostas possuem diversos relacionamentos com outros objetos como: votos, seguidores, categoria, referência, múltiplos autores e também *logs* de auditória, algo que traz *overhead* desnecessário para a funcionalidade.

A fim de melhorar a experiência do usuário, evitando que o mesmo enfrente demoras para criar textos com copiar e colar, principalmente para textos com muitos parágrafos,

Figura 24 – Tempo de execução para criação de texto com 157.883 caracteres e 647 parágrafos



Fonte: Próprio autor

fica como oportunidade para próximos trabalhos a implementação de mecanismos de criação de textos participativos de forma assíncrona. Isto é, a requisição de criação do texto é respondida instantaneamente, redirecionando o usuário para tela onde verá os parágrafos enquanto a criação é processada em segundo plano pela aplicação. Uma possível melhoria também seria a separação do componente de textos participativos do componente de propostas, reduzindo significativamente o *overhead* de informações inutilizadas.

Apesar destes problemas serem conhecidos e terem possíveis soluções mapeadas, com a limitação de tempo de quatro *sprints*, os resultados se limitarão aos já aqui apresentados.

O *merge request* com as alterações dessa *sprint* está disponível em <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/641>.

4.3.6 Sprint para Ajustes na Ferramenta de Cadastro de Usuários

O fluxo de trabalho contou com quatro *sprints* dedicadas para realização do desenvolvimento sobre a ferramenta de textos participativos. Uma quinta *sprint* foi dedicada unicamente para resolução do problema de desempenho no cadastro de usuários na plataforma. O único requisito da *sprint* é descrito como:

- **Requisito 1 da Sprint**: Resolver o problema de lentidão e indisponibilidade da funcionalidade de cadastro de usuários exibido durante o PPA.

4.3.6.1 Requisito 1 da Sprint

Um dos problemas levantados pela Dataprev nas análises conduzidas por eles foi o impacto causado pelo uso do operador `ILIKE` durante o processamento de requisições de cadastro de novos usuários. Dito isto, a abordagem para resolução deste problema foi a de encontrar maneiras de atingir o mesmo resultado da funcionalidade de desambiguação sem utilizar o operador ILIKE na consulta SQL.

A *model* utilizada para representação de usuários é a `Decidim::User`, essa *model* herda da *model* `Decidim::UserBaseEntity`. A `UserBaseEntity` inclui um módulo chamado `Nicknamizable`, que adiciona validações sobre o `nickname` do usuário, além do

método `disambiguate` que recebe como entrada um apelido e retorna o mesmo apelido intocado caso não existam apelidos iguais, ou retorna o apelido com uma variação numérica ao final caso contrário. A definição do método pode ser vista na Figura 25.

Figura 25 – Código fonte do método original de desambiguação

```ruby
def disambiguate(name, scope)
  candidate = name

  2.step do |n|
    return candidate if Decidim::UserBaseEntity.where("nickname ILIKE ?", candidate.downcase)
                                               .where(scope)
                                               .empty?

    candidate = numbered_variation_of(name, n)
  end
end
```

Fonte: Próprio autor

Ora, dado que inicialmente a desambiguação é case insensitive, e o *nickname* candidato era transformado para minúsculo antes de realizar a consulta no banco, se o *nickname* presente no banco também estivesse no mesmo *case* do candidato, bastaria fazer uma comparação direta com o operador de igualdade `=`. Para isso, foi utilizado o operador `LOWER` disponível no PostgreSQL para compor a consulta. A Figura 26 demonstra a nova versão do método de desambiguação com essa alteração.

Figura 26 – Código fonte do método desambiguação modificado

```ruby
def disambiguate(name, scope)
  candidate = name

  2.step do |n|
    return candidate unless nickname_already_exists?(candidate, scope)

    candidate = numbered_variation_of(name, n)
  end
end

def nickname_already_exists?(candidate, scope)
  Decidim::UserBaseEntity.where("LOWER(nickname) = ?", candidate.downcase)
                         .where(scope)
                         .present?
end
```

Fonte: Próprio autor

A fim de comparação, foi executado no ambiente produtivo do Brasil Participativo ambas as consultas SQL utilizadas no método `disambiguate` com o comando `EXPLAIN ANALYZE`. As consultas receberam um parâmetro extra `LIMIT 1` para não trazer todas as tuplas que satisfazem a condição. Isso foi necessário pois a ferramenta utilizada para executar a consulta no ambiente produtivo foi a `Rails DB`, que é executada no próprio *frontend* da aplicação.

Na Figura 27 é mostrada a consulta executada no ambiente produtivo utilizando `ILIKE`, enquanto na Figura 28 é mostrada a que utiliza `LOWER`.

Figura 27 – Consulta SQL com *Explain Analyze* utilizando `ILIKE`

```
1 EXPLAIN ANALYZE
2 SELECT "decidim_users".*
3 FROM "decidim_users"
4 WHERE nickname ILIKE 'nickname-teste'
5 LIMIT 1
```

Fonte: Próprio autor

Figura 28 – Consulta SQL com *Explain Analyze* utilizando `LOWER`

```
1 EXPLAIN ANALYZE
2 SELECT "decidim_users".*
3 FROM "decidim_users"
4 WHERE (LOWER(nickname) = 'nickname-teste')
5 LIMIT 1
```

Fonte: Próprio autor

Na Figura 29 é exibido o resultado da consulta utilizando `ILIKE`, e na Figura 30 é exibido o resultado da consulta utilizando a comparação de igualdade com a função `LOWER`. A plataforma no momento da consulta contava com mais de 1 milhão e 600 mil usuários.

Nota-se que a consulta que utiliza o `LOWER` gasta cerca de 25% menos tempo para ser processada em comparação à consulta que utiliza o `ILIKE`. Dado fato de que até o presente momento deste trabalho não houve cenários de uso da plataforma na mesma escala que foi vista com o PPA em 2023, esta solução foi considerada como satisfatória.

Figura 29 – Resultado da consulta SQL com *Explain Analyze* utilizando `ILIKE`

QUERY PLAN
Limit (cost=0.00..730.10 rows=1 width=1028) (actual time=1580.927..1580.928 rows=0 loops=1)
-> Seq Scan on decidim_users (cost=0.00..116816.54 rows=160 width=1028) (actual time=1580.925..1580.926 rows=0 loops=1)
Filter: ((nickname)::text ~~* 'nickname-teste'::text)
Rows Removed by Filter: 1617958
Planning Time: 0.330 ms
Execution Time: 1580.968 ms

Fonte: Próprio autor

Figura 30 – Resultado da consulta SQL com *Explain Analyze* utilizando `LOWER`

QUERY PLAN
Limit (cost=0.00..15.09 rows=1 width=1028) (actual time=1183.891..1183.893 rows=0 loops=1)
-> Seq Scan on decidim_users (cost=0.00..120818.65 rows=8004 width=1028) (actual time=1183.889..1183.890 rows=0 loops=1)
Filter: (lower((nickname)::text) = 'nickname-teste'::text)
Rows Removed by Filter: 1617958
Planning Time: 0.216 ms
Execution Time: 1183.942 ms

Fonte: Próprio autor

Mas, ainda existem pontos para serem explorados em futuras oportunidades, como por exemplo o uso de índices sobre o *nickname* em *lowercase*.

As modificações realizadas nessa *sprint* estão disponíveis no *merge request* em <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/merge_requests/41>

4.4 Testes de Resultados

Os requisitos centrais das entregas realizadas neste trabalho eram sobre melhorar o desempenho das páginas de visualização e de edição do texto participativo. Sendo assim, os testes de carga serão executados para medir quantitativamente os resultados obtidos pelas modificações realizadas na ferramenta de textos participativos do Brasil Participativo que visaram o ganho de desempenho.

É importante observar que as funcionalidades de *parsing* de texto do usuário através do comando copiar e colar são completamente novas na plataforma, o que impossibilita um comparativo.

Para a execução dos testes, será utilizada a ferramenta de linha de comando *Apache HTTP Server Benchmarking Tool* (ApacheBench). Os testes serão executados em um computador com sistema operacional Linux, distribuição Ubuntu 24.04.2 LTS. As configurações de *hardware* podem ser vistas na Tabela 3. Os testes utilizarão a aplicação do Brasil Participativo com execução em modo produção. O protocolo utilizado será HTTP, e o próprio *Rails* servirá os *assets*. O *web-server Puma* será executado em *single mode* com cinco *threads*.

Tabela 3 – Especificação do hardware de teste

Processador	AMD Ryzen 5 7430U (6 Cores/ 12 Threads)
Placa de Vídeo	Radeon Graphics 2GB (Integrado)
Memória RAM	24 GB DDR4

Os testes consistirão em submeter sob carga as páginas de visualização e edição do texto participativo.

Serão levantados dados para comparação com a realidade do cotidiano e os resultados dos testes apresentados pelo ApacheBench, trazendo a noção do impacto causado pelas modificações realizadas. Para trazer variabilidade nos testes, serão executados tanto os testes com volumetria de parágrafos histórica, quanto com volumetria de parágrafos extremas.

Para ambos os cenários de teste, cada parágrafo terá seu conteúdo preenchido com um texto de exemplo "*Lorem Ipsum*"de 563 caracteres cada. O conteúdo utilizado será:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed in odio non mi auctor commodo. Aenean consequat metus massa, eget vulputate nibh suscipit vitae. Aliquam sagittis vel odio eleifend posuere. Curabitur hendrerit est elit, ac hendrerit nulla laoreet sit amet. Proin dictum tempor vulputate. Etiam cursus, quam sed cursus laoreet, dui nisi efficitur quam, at maximus dui nibh et velit. Curabitur ullamcorper ligula vitae mauris cursus rhoncus. Suspendisse laoreet felis euismod lorem accumsan, sit amet imperdiet mauris laoreet. Fusce vel tristique felis.

A ideia central é verificar o quão bem estas páginas da plataforma possivelmente operarão no dia-dia e em cenários de alta demanda. Para os cenários extremos, será adotada a volumetria de 2 mil parágrafos por texto participativo, e 10 administradores utilizando a ferramenta paralelamente.

A versão anterior às modificações descritas neste trabalho utilizada para comparação nos testes é a *tag* 1.5.0, que está disponível no repositório oficial do Brasil Participativo em <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr/-/tags/1.5.0>.

4.4.1 Teste de Carga de Visualização do Rascunho

4.4.1.1 Contexto

A fim de obter um parâmetro para comparação com teste de carga da página de edição de rascunho, foi executada no ambiente produtivo do Brasil Participativo uma consulta SQL para descobrir o dia que houve mais criações de componentes de texto participativo. A consulta considera o período entre o primeiro dia do ano de 2024 até o último dia do mês de maio de 2025, admitindo apenas componentes que foram publicados e que são textos participativos. A consulta SQL pode ser conferida na Figura 31, e o seu resultado é demonstrado na Tabela 4.

Figura 31 – Consulta SQL para classificação dos dias com maior carga de criação de textos participativos

```sql
SELECT DATE(created_at) created_at_date, COUNT(id) components
FROM decidim_components dc
WHERE EXISTS (
      SELECT id FROM
        decidim_proposals_proposals
      WHERE decidim_component_id = dc.id
  )
  AND settings->'global'->>'participatory_texts_enabled' = 'true'
    AND manifest_name = 'proposals'
    AND created_at >= '2024-01-01' AND created_at < '2025-06-01'
    AND published_at IS NOT NULL
GROUP BY created_at_date
ORDER BY components DESC
LIMIT 5;

```

Fonte: Próprio autor

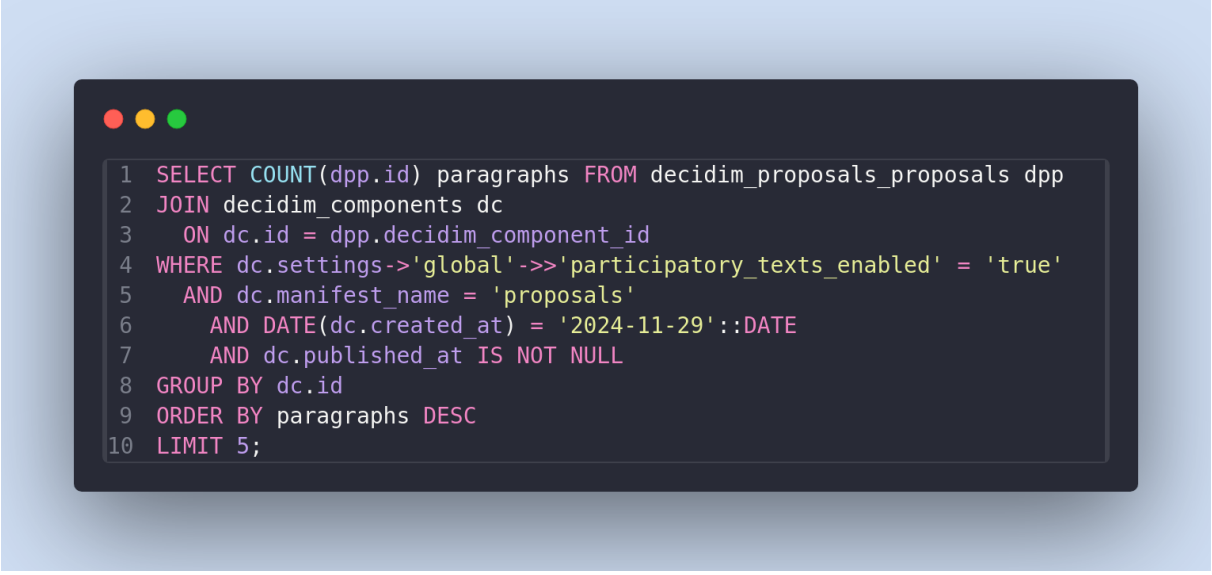
Tabela 4 – Top 5 dias com maior volume de textos participativos criados

Data	Textos participativos criados
29/11/2024	45
30/11/2024	24
02/12/2024	17
22/01/2025	15
09/10/2024	12

Fonte: Próprio autor

No dia 29 de novembro de 2024, houve 45 componentes de texto participativo criados na plataforma. Uma segunda consulta foi executada para descobrir a classificação dos cinco textos participativos dessa data com a maior quantidade de parágrafos. A consulta SQL pode ser conferida na Figura 32. Como resultado, houve textos com 124, 85, 80, 76 e 74 parágrafos.

Figura 32 – Consulta SQL para classificação dos textos participativos com maior quantidade de parágrafos



```sql
SELECT COUNT(dpp.id) paragraphs FROM decidim_proposals_proposals dpp
JOIN decidim_components dc
  ON dc.id = dpp.decidim_component_id
WHERE dc.settings->'global'->>'participatory_texts_enabled' = 'true'
  AND dc.manifest_name = 'proposals'
    AND DATE(dc.created_at) = '2024-11-29'::DATE
    AND dc.published_at IS NOT NULL
GROUP BY dc.id
ORDER BY paragraphs DESC
LIMIT 5;
```

Fonte: Próprio autor

Com base nessas informações, será considerado um dia com pico de uso com 50 componentes sendo criados, cada um deles com 130 parágrafos. Será levado em conta que o usuário administrador possivelmente trabalha na plataforma entre os horários das nove horas da manhã até meio dia (12 horas), e das 13 horas até às 18 horas, significando uma janela de trabalho de oito horas. O que significa em aproximadamente de seis a sete componentes criados por hora.

Assumindo um possível pior cenário, também por convenção, será considerado que o administrador realiza em média no mínimo todas as cinco operações em cada parágrafo, isto é: editar, mesclar, mover, ocultar e desabilitar interações. Para este teste será ignorado o fato de que a operação de mesclagem diminui a contagem de parágrafos em uma unidade. A realização de cada operação acarreta na atualização da página de visualização do rascunho, portanto considerando os seis a sete componentes por hora (nesse caso será considerado sete, simplesmente), a quantidade de 150 parágrafos e as cinco operações por parágrafo, tem-se que a página de visualização do rascunho será renderizada 5.250 vezes em uma hora, ou 87 vezes por minuto, ou 1,46 vezes por segundo.

4.4.1.2 Execução e Resultados

Os testes foram realizados disparando 100 requisições com o fator de paralelismo igual a cinco, isto é, ocupando todas as cinco threads do servidor cada uma com um único usuário. Os resultados podem ser visualizados na Tabela 5

Tabela 5 – Teste da página de edição do texto

Métrica	Versão anterior	**Nova versão**
Tamanho da página	931.752 bytes	712.295 bytes
Tempo gasto no teste	109 segundos	18 segundos
Requisições por segundo	0,91	5,38
Tempo médio por requisição	1.097 milissegundos	185 milissegundos

Fonte: Próprio autor

4.4.1.3 Testes Extremos

Os testes extremos foram realizados com 10 requisições com fator de paralelismo igual a 10. O número de requisições foi escolhido como 10 para garantir que os testes ocorressem em uma escala de tempo na casa de alguns minutos, evitando problemas de expiração de sessão, dado que o teste foi realizado usando um Cookie de sessão de usuário administrador extraído do navegador manualmente. O texto participativo utilizado possui 2 mil parágrafos. Os resultados podem ser conferidos na Tabela 6.

Tabela 6 – Teste extremo da página de edição do texto

Métrica	Versão anterior	**Nova versão**
Tamanho da página	14.027.611 bytes	5.888.914 bytes
Tempo gasto no teste	158 segundos	20 segundos
Requisições por segundo	0,06	0,48
Tempo médio por requisição	15.837 milissegundos	2.084 milissegundos

Fonte: Próprio autor

Considerando ambos os testes, nota-se que em média houve uma redução de 84,9% do tempo gasto para processar a requisição.

4.4.2 Teste de Carga de Visualização do Texto

4.4.2.1 Execução e Resultados

Os testes foram realizados disparando 100 requisições com o fator de paralelismo igual a cinco. Os resultados podem ser visualizados na Tabela 7. Para esse teste não se faz necessário o uso de Cookie de sessão de usuário, uma vez que a página é pública e possível de visualizar sem estar logado na plataforma.

4.4.2.2 Testes Extremos

A execução dos testes extremos da página de visualização do texto seguiu a configuração de 10 requisições com fator de paralelismo igual a 10. O resultado pode ser

Tabela 7 – Teste da página de visualização do texto

Métrica	Versão anterior	**Nova versão**
Tamanho da página	437.372	528.236
Tempo gasto no teste	455 segundos	15 segundos
Requisições por segundo	0,22	6,61
Tempo médio por requisição	4.551 milissegundos	151 milissegundos

Fonte: Próprio autor

verificado na Tabela 8.

Tabela 8 – Teste extremo da página de visualização do texto

Métrica	Versão anterior	**Nova versão**
Tamanho da página	2.510.864 bytes	3.239.958 bytes
Tempo gasto no teste	503 segundos	16 segundos
Requisições por segundo	0,02	0,61
Tempo médio por requisição	50.345 milissegundos	1.644 milissegundos

Fonte: Próprio autor

Nota-se pelos resultados que houve uma redução de aproximadamente 96,7% do tempo de processamento da requisição.

4.4.3 Resultados Gerais

As alterações abordadas neste trabalho foram devidamente incluídas no código fonte do Brasil Participativo e utilizadas no ambiente produtivo. Até o presente momento a ferramenta está sendo utilizada e posta à prova. Considera-se como um sucesso o atingimento de todos os requisitos levantados dado fato que foi realizada uma cerimônia de validação juntamente com o time de produto do Lab Livre.

4.5 Resumo do Capítulo

Neste capítulo foram discutidos os principais problemas de desempenho do Brasil Participativo apontados pela Dataprev. Dentre esses problemas se destacaram: sobreutilização do banco de dados, falta de uso de índices, *memory leak* e lentidão em páginas específicas.

Como foco de desenvolvimento deste trabalho, foram escolhidas as funcionalidades de textos participativos e também de cadastro de usuários, onde ambas apresentaram problemas de desempenho na análise realizada pela Dataprev.

A ferramenta de textos participativos já estava cotada para receber desenvolvimentos no ano de 2024. Portanto, vários requisitos foram levantados junto dos *stakeholders* do projeto e também do time de produto. As medidas adotadas para satisfação desses requisitos foram diversas, e são resumidas a seguir:

1. **Requisito 1**: diminuir o tempo de renderização da página de visualização e de edição do texto participativo. **Solução**: remover o uso de *partials* em laços de repetição, dando preferência para renderização de conteúdo *inline*;

2. **Requisito 2**: possibilitar a criação de textos participativos através de um editor de textos próprio do Brasil Participativo, contando com o uso do comando copiar e colar, trazendo texto de fontes como o Microsoft Word e Google Docs. **Solução**: utilizar o editor de texto Jodit, munido de um algoritmo de detecção de parágrafos e de um sistema de persistência de imagens no `storage`;

3. **Requisito 3**: refatoração da página de visualização do rascunho (edição de parágrafos), trazendo-a para o mesmo padrão de design que a tela de visualização. **Solução**: utilizar a mesma *view* usada para renderização do texto para o usuário público, acrescida de botões de ações para o usuário administrador;

4. **Requisito 4**: refatorar a jornada de criação de textos participativos de acordo com o acordado com o time de produto. **Solução**: adicionar chamadas de `redirect_to` nas *actions* envolvidas, levando o usuário para uma página mais adequada de acordo com o estado atual do componente;

5. **Requisito 5**: resolver o problema de lentidão e indisponibilidade da funcionalidade de cadastro de usuários exibido durante o PPA. **Solução**: trocar o uso do operador `ILIKE` do SQL pela combinação da função `LOWER` e operador de igualdade.

Um dos problemas notáveis que foi resolvido neste trabalho é a detecção de parágrafos dentro de um texto estruturado no formato HTML. Editores de texto como o Microsoft Word e Google Docs utilizam organizações de *tags* que tornam complexa a elaboração de uma solução.

Como saída foi adotado o uso da biblioteca `Nokogiri` para converter o HTML do formato de texto (*string*) para uma estrutura de dados em formato de árvore. Foi desenvolvido um algoritmo recursivo para percorrer essa estrutura de dados e decidir quais fragmentos deveriam ser considerados como parágrafos ou não. Um nó e seus filhos eram considerados um parágrafo caso ele não fosse o elemento raiz da árvore, ou não fosse uma `<div>`, ou não fosse uma *tag* de texto contendo múltiplos parágrafos independentes dentro de si.

Sobre a síntese do trabalho em geral, houve cinco *sprints* de desenvolvimento, cada uma delas com um mês de duração. O trabalho de desenvolvimento foi realizado em parceria com o time de produto e também com o time de desenvolvimento do Lab Livre. O time de produto esteve a cargo principalmente da realização da validação das entregas, enquanto o time de desenvolvimento, principalmente o gestor, esteve a cargo de fazer a validação técnica.

Na Tabela 9 são exibidos os ganhos de desempenho para as páginas de edição e visualização de texto participativo, e também da consulta SQL usada para desambiguação de *nicknames* utilizada no cadastro de novos usuários.

Tabela 9 – Ganhos de desempenho após as modificações

Funcionalidade	**Redução do tempo**
Tela de visualização do Texto Participativo	96,7%
Tela de edição do Texto Participativo	84,5%
Consulta de verificação de existência de *nickname*	25%

5 Considerações do Autor

Neste capítulo são abordados assuntos pertinentes à execução do trabalho na perspectiva do autor. Durante o desenvolvimento, diversos problemas surgiram trazendo desafios extras para cumprimento dos objetivos levantados.

Na Seção 5.1 é abordado o assunto sobre retrabalho e redefinição de requisitos, na Seção 5.2 é levado a debate a forma como as funcionalidades foram testadas e validadas durante o ciclo de desenvolvimento, e, por fim, na Seção 5.3 são apresentados alguns detalhes de como funcionou o esquema de colaboração para o Brasil Participativo junto dos times do Lab Livre e também dos *stakeholders*.

5.1 Mudança de Requisitos e Retrabalho

Sob a ótica da engenharia de *software*, um dos problemas que surgiram durante as etapas de desenvolvimento foi o retrabalho. Nesse contexto, entende-se como retrabalho o ato de refazer atividades já finalizadas com o intuito de implementar mudanças de requisitos, seja por conta de alterações de contexto do projeto, seja por conta da melhor definição de outros requisitos que não estavam tão claros, ou seja pela simples mudança de desejo do cliente. Esse tipo de atividade geralmente acarreta no aumento do custo e do tempo de desenvolvimento do projeto, podendo representar 50% ou mais do valor total (JAYATILLEKE; LAI, 2021).

Neste trabalho, no início foi definido como premissa que os usuários administradores fariam a inserção de tabelas dentro do texto no formato de imagens. Porém, após a entrega da funcionalidade disponibilizando o Trix como editor de textos, alguns dos usuários *stakeholders* que participaram dos testes de aceitação, juntamente do time de produto do Lab Livre, concluíram que era inviável em termos de usabilidade e experiência de usuário inserir tabelas como imagens. Diversos textos de exemplo utilizados nos testes possuíam dezenas de tabelas. Considerando a jornada de uso do texto participativo, essas tabelas ainda poderiam sofrer modificações após a publicação do conteúdo, o que seria extremamente difícil de se realizar, dada essa limitação.

Para sanar esse problema, foi necessário substituir o Trix pelo editor de textos Jodit, e também redesenhar a solução de detecção de parágrafos, uma vez que o algoritmo de detecção utilizado para o Trix não cobria corretamente os cenários avaliados durante o uso do Jodit. Essa alteração constituiu grande parte do trabalho aqui realizado, e por alguns momentos direcionou os esforços de desenvolvimento mais para a resolução de desafios de implementação do que para a busca por melhoria de desempenho na ferramenta.

Esse caso exemplifica a necessidade de uma boa definição de requisitos e premissas de desenvolvimento. Apesar de haver um levantamento prévio de requisitos, decisões tomadas no início ou durante a etapa de desenvolvimento devem ser acordadas junto dos *stakeholders*, a fim de garantir que elas não firam as necessidades dos usuários.

5.2 Testes de Aceitação

Ao final de cada *sprint* de trabalho, o desenvolvedor liberava para testes as alterações feitas. Em geral, isso era comunicado ao gestor do time de desenvolvimento, que assim que possível disponibilizava uma *release* de testes no Lab Decide, o ambiente de testes do Brasil Participativo. Apesar do desenvolvedor ter a possibilidade de realizar testes de aceitação das funcionalidades com base nos requisitos, as rodadas de testes definitivas eram feitas pelo time de produto de forma isolada. O time de produto muitas vezes também contava com a participação de alguns *stakeholders* para terem papel opinativo sobre funcionamento da melhoria entregue.

Entretanto, na execução desses testes, os envolvidos nem sempre conheciam as limitações e a forma de uso das funcionalidades lançadas, além de não estarem conscientes das decisões técnicas e funcionais tomadas ao longo do desenvolvimento. A participação do desenvolvedor nos testes de aceitação acontecia para eventuais esclarecimentos de dúvidas, e muita das vezes de forma tardia. Essas situações geralmente levavam o responsável pelo teste a interpretar um comportamento da funcionalidade como uma inconsistência ou como um defeito, mesmo que não fosse o caso, o que resultava em desacordos entre os times e/ou *stakeholders*. Além disso, esses desacordos geravam atrasos no projeto, e poderiam potencialmente comprometer a confiança sobre a qualidade dos artefatos entregues pelo desenvolvedor.

Como melhoria de processo, pode-se adotar uma colaboração ativa do desenvolvedor durante os testes, trazendo maior celeridade na atividade e enriquecendo o fornecimento de *feedbacks*. Nota-se também a necessidade de que os profissionais envolvidos no projeto estejam em constante comunicação, para garantir que as decisões tomadas sejam do conhecimento de todos os interessados.

5.3 Rotina de trabalho no Lab Livre

O projeto Brasil Participativo é executado hoje primariamente pelo Lab Livre da UnB. Durante a execução deste trabalho, além do desenvolvimento relacionado ao texto participativo, foram feitas pelo autor inúmeras outras funcionalidades, correção de defeitos e atividades de engenharia de *software* para assuntos secundários da plataforma. Portanto, apesar do formato aqui apresentado se assemelhar bastante a uma rotina que

segue metodologias ágeis com foco total na ferramenta de textos, os pacotes de trabalho executados não seguiram uma padronização em termos de esforço e de cronograma, uma vez que era necessário executar múltiplas outras tarefas juntamente destas relacionadas aos textos participativos e ao cadastro de usuários.

O time do Lab Livre, principalmente para esta atividade, era dividido entre produto, *design* e desenvolvimento. O time de produto era responsável por gerir os requisitos, realizar a tomada de decisões e acordar prazos de entrega com os *stakeholders*. O time de *design* era responsável pela elaboração da UX/UI da ferramenta, onde, em geral, era discutido com o time de produto quais resultados eram desejáveis para a aplicação, levando em conta as necessidades dos usuários. O time de desenvolvimento era responsável pela criação do código fonte da plataforma, buscando alcançar os requisitos funcionais e não funcionais levantados pelo time de produto e de *design*.

Neste trabalho, vale destacar que as atividades foram apresentadas sob a ótica de um desenvolvedor. Portanto, detalhes minuciosos de execução das atividades dos demais times não foram abordados.

6 Considerações Finais

Neste capítulo é apresentado um resumo dos objetivos, desafios e soluções deste trabalho. Na Seção 6.1 é apresentado o resumo da obra, e na Seção 6.2 são discutidos potenciais pontos de atuação para trabalhos futuros.

6.1 Atividades e Resultados

Este trabalho teve como objetivo demonstrar uma jornada de desenvolvimento de funcionalidades levando em consideração requisitos levantados por *stakeholders* e outros times, tendo como foco principal o desenvolvimento de funcionalidades com requisitos de desempenho mais rigorosos.

Com o surgimento de problemas na plataforma Brasil Participativo em 2023 e com a análise realizada pela Dataprev, um plano de atividades foi traçado considerando também as necessidades levantadas pelos *stakeholders*. As funcionalidades de textos participativos e cadastro de usuários foram tomadas como objetos de estudo e de melhorias.

A metodologia adotada se baseou no desenvolvimento contínuo. Uma série de requisitos iniciais foram definidos, e em cada ciclo de desenvolvimento novos requisitos eram priorizados para serem trabalhados. Em suma, os requisitos foram:

- Diminuir o tempo de renderização da página de visualização e de edição do texto participativo;
- Possibilitar a criação de textos participativos através de um editor de textos próprio do Brasil Participativo, contando com o uso do comando copiar e colar, trazendo texto de fontes como o Microsoft Word e Google Docs;
- Refatoração da página de visualização do rascunho (edição de parágrafos), trazendo-a para o mesmo padrão de *design* que a tela de visualização;
- Refatorar a jornada de criação de textos participativos de acordo com o acordado com o time de produto;
- Resolver o problema de lentidão e indisponibilidade da funcionalidade de cadastro de usuários exibido durante o PPA.

Durante a jornada de desenvolvimento, uma série de modificações foram realizadas desde as páginas de exibição de conteúdo do texto até mesmo na forma como o usuário administrador submete o material na ferramenta.

Na primeira *sprint* foi posta à prova as principais ideias de solução, removendo o uso indiscriminado de *partials* do *frontend*, e adicionando um editor de textos *rich text* para envio de texto. Juntamente, foi criado um algoritmo de *parser*, a fim de separar parágrafos, de tal forma que cada um deles seja exibido e receba interações de forma isolada. Melhorias de interface também foram conduzidas para que a ferramenta seguisse o padrão de estilo `Gov.Br` já adotado na plataforma.

Na segunda e terceira *sprint* houve ajustes de usabilidade, aparência e também a correção de defeitos no *parser* da ferramenta. Pode-se dizer que esses dois ciclos foram utilizados para polir o que foi entregue já no primeiro.

Na quarta *sprint*, surgiu a necessidade de implementar de forma diferente a edição e criação de textos, de tal maneira que fosse possível manipular tabelas nativamente na plataforma. O editor de textos inicialmente disponibilizado, o Trix, foi substituído pelo editor de textos Jodit; e com isso foi também necessária a substituição do algoritmo de *parser*.

E, por fim, na quinta *sprint* foi abordado o problema de lentidão na criação de novos usuários, realizando ajustes em um ponto específico: a desambiguação de *nicknames*, melhorando em 25% o desempenho da consulta raiz do problema.

Ao final, foram conduzidos testes de carga para avaliar os ganhos de desempenho com as alterações propostas no trabalho. Os resultados foram promissores, com reduções de tempo gasto de até 96%.

6.2 Trabalhos Futuros

No ponto em que se encerraram as atividades mencionadas neste trabalho, nota-se que existem diversas aberturas para melhorias, correções e evoluções da plataforma e das funcionalidades trabalhadas. De forma não exaustiva as oportunidades de atuação são discutidas a seguir.

6.2.1 Texto Participativo

Com o uso do Jodit para captura do *input* do usuário, existe a possibilidade, e também necessidade de adicionar validação sobre o conteúdo a ser processado e guardado no banco de dados. O texto submetido pelo usuário é enviado no formato HTML, e dado que essa linguagem permite, além de estruturar textos, especificar *scripts* e estilo em seu conteúdo, é necessário que seja feito um pós-processamento no lado *backend*. O Jodit foi configurado de forma que atributos de estilo e *scripts* sejam removidos após a colagem de textos advindos de fontes externas. Mas, nada impede que um usuário mal-intencionado sequestre uma conta de um usuário administrador para submeter HTML com conteúdo

malicioso.

Com a remoção de *tags* de estilo e *script* do texto, algumas inconsistências surgem ao copiar e colar textos vindos do Google Docs, por exemplo. Após observar o comportamento do copiar durante os testes, notou-se que a estilização do texto, como negrito, itálico e espaçamento é feita com o uso das *tags* de estilo. Isto é, um texto negrito, por exemplo, ao invés de ser representado através de uma *tag* HTML, era representado através de atributos CSS. Para esses casos, o conteúdo colado trazia o texto inteiro envolto sob uma *tag* `<strong>`, dando a impressão de que tudo está em negrito, mesmo que no Google Docs isso não fosse o caso.

Na perspectiva de segurança, existe também a necessidade da criação de um mecanismo inteligente para validação das imagens submetidas. Hoje as imagens são enviadas *inline* no corpo do texto no formato `Data URL`. Uma vez que o texto é processado pelo *backend*, essas imagens são transformadas em `blobs` e enviadas ao `ActiveStorage` sem muita preocupação com seu conteúdo e tamanho.

Ainda sobre imagens, uma possível melhoria seria a criação de um mecanismo de envio assíncrono para o *backend*. Com uma *controller* dedicada para o processamento dessas imagens, as rotinas de validação e persistência aconteceriam de forma granular e independente, removendo do *parser* um pouco da sua complexidade. Para que essa melhoria seja efetiva, também é necessário que seja implementado um mecanismo de limpeza do `storage`, de tal forma que imagens órfãs sejam removidas posteriormente, dado que uma imagem pode ser submetida e apagada logo em seguida sem nem mesmo persistir o texto no banco de dados.

A criação de textos grandes ainda sofre com o problema de lentidão, uma vez que é necessário separar cada parágrafo em um objeto único para ser salvo no formato de proposta no banco de dados. Esses objetos possuem relacionamentos com outras entidades e também levam à criação de outros objetos para guardar metadados de autoria e auditoria. Visando o ganho de desempenho, idealmente cada parágrafo do texto não deveria utilizar o modelo de propostas para operar, mas sim um modelo próprio, ou uma estrutura de dados que permita sua manipulação e exibição de forma rápida.

6.2.2 Cadastro de Usuários

No cadastro de usuários, com a troca do `ILIKE` pelo uso da função `LOWER`, observou-se uma melhora de desempenho. Porém, existem formas atingir resultados ainda melhores na desambiguação de *nicknames* através do uso de índices ou removendo a função `LOWER` da consulta. A adoção de um índice poderia melhorar o desempenho da consulta, garantindo que a função `LOWER` não precise ser executada para cada tupla. Outra forma seria remover o uso do `LOWER`, mas traria consigo uma série de cuidados necessários: garantir que os

nicknames são submetidos ao banco sempre em minúsculo, e os *nicknames* já existentes estejam em minúsculo.

6.2.3 Vazamento de Memória e Indisponibilidade do Banco de Dados

Pela análise da Dataprev, foi visto que o vazamento de memória é um problema real, e que exige esforço do time de infraestrutura para corrigir essa falha de tempos em tempos. Outro problema constatado foi a indisponibilidade do banco em diversos momentos de alta carga de acesso de usuários. Existem ferramentas que podem ser adotadas na camada do *backend* para potencialmente mitigar esses problemas.

Para melhorar o sistema de *pool* do banco, pode-se explorar a utilização de ferramentas como o `pg-bouncer`. E, para aprimorar o uso de memória, pode-se examinar o uso de outros mecanismos de alocação de memória, como, por exemplo, o `jemalloc`.

Referências

ADAM, B. M.; BESARI, A. R. A.; BACHTIAR, M. M. Backend server system design based on rest api for cashless payment system on retail community. In: IEEE. *2019 International Electronics Symposium (IES)*. [S.l.], 2019. p. 208–213. Citado na página 27.

CARLSON, J. L. *Redis in Action*. Shelter Island, NY: Manning Publications Co., 2013. ISBN 9781935182054. Citado na página 29.

CONALLEN, J. Modeling web application architectures with uml. *Communications of the ACM*, ACM New York, NY, USA, v. 42, n. 10, p. 63–70, 1999. Citado na página 26.

FRY, J. P.; SIBLEY, E. H. Evolution of data-base management systems. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 8, n. 1, p. 7–42, 1976. Citado na página 28.

GONG, Y. et al. The architecture of micro-services and the separation of frond-end and back-end applied in a campus information system. In: IEEE. *2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*. [S.l.], 2020. p. 321–324. Citado na página 27.

GOUGH, J.; BRYANT, D.; AUBURN, M. *Mastering API Architecture*. [S.l.]: "O'Reilly Media, Inc.", 2021. Citado 2 vezes nas páginas 23 e 27.

GROUP, P. G. D. *About PostgreSQL*. 2023. Accessed on December 4, 2023. Disponível em: <https://www.postgresql.org/about/>. Citado na página 29.

JATANA, N. et al. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, v. 1, n. 6, p. 1–5, 2012. Citado na página 28.

JAYATILLEKE, S.; LAI, R. A method of assessing rework for implementing software requirements changes. 1 2021. Disponível em: <https://opal.latrobe.edu.au/articles/journal_contribution/A_method_of_assessing_rework_for_implementing_software_requirements_changes/14504886>. Citado na página 81.

JUGO, I.; KERMEK, D.; MEŠTROVIĆ, A. Analysis and evaluation of web application performance enhancement techniques. In: SPRINGER. *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings 14*. [S.l.], 2014. p. 40–56. Citado na página 33.

MDN WEB DOCS. *Cross-Origin Resource Sharing (CORS)*. 2025. Acesso em: 17 de julho de 2025. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS>. Citado na página 57.

MDN WEB DOCS. *data: URLs*. 2025. Acesso em: 18 de julho de 2025. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS>. Citado na página 65.

METADECIDIM. *Decidim Documentation*. 2023. Acesso em: 10 de dezembro de 2023. Disponível em: <https://docs.decidim.org/en/v0.27/index.html>. Citado na página 33.

METADECIDIM. *General description and introduction to how Decidim works*. 2023. Acesso em: 10 de dezembro de 2023. Disponível em: <https://docs.decidim.org/en/v0.27/features/general-description>. Citado 2 vezes nas páginas 34 e 35.

METADECIDIM. *Participatory texts*. 2025. Acesso em: 19 de julho de 2025. Disponível em: <https://docs.decidim.org/en/v0.27/admin/components/proposals/participatory_texts.html>. Citado na página 35.

METADECIDIM. *View Models a.k.a. Cells*. 2025. Acesso em: 17 de julho de 2025. Disponível em: <https://docs.decidim.org/en/v0.30/develop/view_models_aka_cells>. Citado na página 47.

MODEL-VIEW-CONTROLLER Pattern. In: LEARN Objective-C for Java Developers. Berkeley, CA: Apress, 2009. p. 353–402. ISBN 978-1-4302-2370-2. Disponível em: <https://doi.org/10.1007/978-1-4302-2370-2_20>. Citado 3 vezes nas páginas 23, 29 e 30.

OLUWATOSIN, H. S. Client-server model. *IOSR Journal of Computer Engineering*, IOSR Journals, v. 16, n. 1, p. 67–71, 2014. Citado na página 27.

PARTICIPATIVO, B. *Sobre o Brasil Participativo*. 2023. Acesso em: 10 de dezembro de 2023. Disponível em: <https://brasilparticipativo.presidencia.gov.br/processes/brasilparticipativo/f/33>. Citado na página 36.

TANENBAUM, A. S.; KLINT, P.; BOHM, W. Guidelines for software portability. *Software: Practice and Experience*, Wiley Online Library, v. 8, n. 6, p. 681–698, 1978. Citado 2 vezes nas páginas 23 e 26.

THE RAILS FOUNDATION. *Action Controller Overview*. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: <https://guides.rubyonrails.org/action_controller_overview.html>. Citado na página 32.

THE RAILS FOUNDATION. *Action View Overview*. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: <https://guides.rubyonrails.org/action_view_overview.html>. Citado na página 32.

THE RAILS FOUNDATION. *Active Model Basics*. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: <https://guides.rubyonrails.org/active_model_basics.html>. Citado na página 31.

THE RAILS FOUNDATION. *Active Record Basics*. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: <https://guides.rubyonrails.org/active_record_basics.html>. Citado na página 31.

THE RAILS FOUNDATION. *Ruby on Rails API*. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: <https://api.rubyonrails.org/>. Citado 2 vezes nas páginas 23 e 31.

THE RAILS FOUNDATION. *Debugging Rails Applications*. 2025. Acesso em: 17 de julho de 2025. Disponível em: <https://guides.rubyonrails.org/debugging_rails_applications.html>. Citado na página 47.

WILLIAMS, L. G.; SMITH, C. U. Web application scalability: A model-based approach. In: *Int. CMG Conference*. [S.l.: s.n.], 2004. p. 215–226. Citado na página 33.

YU, S. Acid properties in distributed databases. *Advanced eBusiness Transactions for B2B-Collaborations*, p. 17, 2009. Citado na página 28.