Sidagro Development: An Experience Report on the Partnership between the Federal University of Lavras and the Agricultural Institute of Minas Gerais

Lucas Alvarenga Lopes
Universidade Federal de Lavras
Lavras, MG, Brasil
lucas.lopes7@estudante.ufla.br

Rafael Serapilha Durelli
Universidade Federal de Lavras
Lavras, MG, Brasil
rafael.durelli@ufla.br

Ricardo Terra
Universidade Federal de Lavras
Lavras, MG, Brasil
terra@ufla.br

ABSTRACT

This article presents an experience report on the development of Sidagro Web, a governmental system developed through a partnership between the Federal University of Lavras (UFLA) and the Agricultural Institute of Minas Gerais (IMA), with a primary focus on software quality assurance. The report details key architectural and technical decisions that guided the project throughout its lifecycle, providing the rationale for each choice and emphasizing their contributions to enhancing system quality. The discussion is organized into five main categories: (i) high-level architectural choices, including layered separation and foundational technologies; (ii) backend-specific decisions, such as exception handling, object mapping, auditing mechanisms, and code conventions; (iii) frontend considerations, including form validation and dynamic data handling; (iv) infrastructure-related strategies, covering environment setup, CI/CD pipelines, and DevOps tooling; and (v) documentation and version control practices, which are applied transversally across all project dimensions. We demonstrate that the systematic adoption of these practices significantly improved the system's robustness, maintainability, and long-term sustainability, offering valuable insights for similar large-scale software projects with stringent quality requirements.

KEYWORDS

Experience Report, Software Quality Assurance, Best Practices

1 Introdução

A adoção de boas práticas de engenharia de software é fundamental para garantir a qualidade, manutenibilidade e sustentabilidade de sistemas computacionais [4, 41]. Em contextos governamentais, onde requisitos de confiabilidade e longevidade são críticos, tais práticas tornam-se ainda mais relevantes [66]. No entanto, muitos desses sistemas enfrentam dificuldades relacionadas à ausência de diretrizes arquiteturais claras, práticas inconsistentes de desenvolvimento e deficiências em processos de documentação e versionamento [6, 7].

Este artigo, portanto, apresenta um relato de experiência sobre o desenvolvimento do Sidagro Web, um sistema governamental voltado à gestão agropecuária, fruto de uma parceria entre a Universidade Federal de Lavras (UFLA) e o Instituto Mineiro de Agropecuária (IMA). Embora as tecnologias e práticas adotadas sejam amplamente utilizadas no mercado, a originalidade deste trabalho reside na sua aplicação integrada e sistematizada em um contexto governamental, que impõe desafios específicos como longevidade do sistema, rastreabilidade rigorosa e conformidade com diretrizes de órgãos estaduais de TI. O objetivo central deste relato é demonstrar como decisões arquiteturais e técnicas sistematizadas contribuíram para alcançar atributos essenciais de qualidade — como manutenibilidade, confiabilidade e escalabilidade —, evidenciando de que forma cada escolha técnica fortaleceu esses pilares dentro de um ambiente institucional de alta complexidade.

Inicialmente, definem-se as camadas e tecnologias subjacentes a serem utilizadas. Essas escolhas favorecem a organização estrutural e reduzem o acoplamento entre componentes, promovendo modularidade e escalabilidade. Na sequência, detalham-se as diretrizes aplicadas ao *Backend*, incluindo tratamento de exceções, mecanismos de auditoria e padronização de código. Essas definições fortalecem a confiabilidade, ao permitir uma gestão controlada de falhas, e promovem a manutenibilidade, ao garantir um código mais consistente e compreensível. No *Frontend*, estabelecem-se práticas voltadas à validação de formulários e à integração com dados dinâmicos, contribuindo diretamente para a usabilidade, confiabilidade e acessibilidade, ao evitar inconsistências na entrada de dados, assegurar respostas adequadas da interface em diferentes condições de uso e promover a conformidade com diretrizes de acessibilidade adotadas em sistemas públicos.

Complementarmente, a infraestrutura do projeto é organizada com a definição de ambientes, *pipelines* de integração e entrega contínua (CI/CD) e a adoção de ferramentas *DevOps*. Essa estruturação favorece a portabilidade entre ambientes, assegura a confiabilidade nos processos automatizados e contribui para a eficiência na manutenção e evolução contínua do sistema. Por fim, mas não menos importante, abordam-se práticas de documentação e versionamento, tratadas de forma transversal, cujo papel é fundamental na preservação do conhecimento técnico, no suporte à manutenção evolutiva e na colaboração entre equipes. Tais ações reforçam a manutenibilidade e a evolutividade, aspectos fundamentais para projetos de longa duração e com requisitos de qualidade rigorosos.

O restante deste artigo está organizado como a seguir. A Seção 2 discute as decisões macroarquiteturais e tecnológicas. As Seções 3, 4, 5 e 6 abordam, respectivamente, as práticas no *Backend* e *Frontend*, estratégias de infraestrutura e práticas transversais. Por fim, a Seção 7 apresenta considerações finais e reflexões sobre os aprendizados obtidos.

2 Decisões de Alto Nível

Esta seção tem como objetivo apresentar a estrutura em camadas adotada no Sidagro Web, bem como as principais tecnologias empregadas em cada uma delas. Ao detalhar os componentes que compõem a aplicação, evidencia-se como a segmentação em diferentes camadas contribui para a manutenibilidade, escalabilidade e

portabilidade do sistema. Além de indicar as tecnologias e soluções escolhidas para cada função, justifica-se suas adoções com base nos requisitos e critérios de qualidade estabelecidos para o projeto, respondendo diretamente às demandas do IMA e seguindo orientações da Prodemge, órgão responsável pela governança de TI em Minas Gerais. Essas definições foram validadas pela equipe técnica do projeto, que as confirmou não apenas pela conformidade institucional, mas também pela aderência a atributos de qualidade considerados centrais para o domínio, especialmente a sustentabilidade do sistema a longo prazo.

2.1 Camadas da Aplicação

Durante o desenvolvimento do Sidagro Web, uma das primeiras decisões arquiteturais foi a separação da aplicação em duas camadas principais: *Backend* e *Frontend*. Essa escolha não foi motivada por conveniência ou por seguir tendências do mercado, mas sim como resposta ao conjunto de necessidades identificadas no início do projeto, entre elas a escalabilidade, a facilidade de manutenção e a portabilidade do sistema.

Dessa forma, ao centralizar as regras de negócio no *Backend*, garante-se que, além da aplicação web em desenvolvimento, outras aplicações futuras — como um aplicativo móvel do Sidagro — poderão integrar-se ao sistema reutilizando os mesmos contratos e funcionalidades [62]. Essa característica amplia a vida útil do sistema e reforça sua capacidade de adaptação a novos contextos tecnológicos, garantindo sua portabilidade [39], um dos critérios de qualidade estabelecidos no projeto.

2.2 Tecnologias

Após a definição da arquitetura em camadas para o Sidagro Web — dividindo a aplicação em *Frontend* e *Backend* —, o próximo passo consistiu na seleção criteriosa das tecnologias para sustentar cada camada. Essa escolha não se limitou a preferências por ferramentas populares, mas considerou critérios como robustez, alinhamento com princípios arquiteturais, suporte a longo prazo e aderência a atributos de qualidade de software.

Na camada de *Frontend*, adotou-se o *framework* Angular [2] (versão 19), uma solução consolidada para o desenvolvimento de aplicações complexas e de larga escala [16]. O Angular destaca-se por promover uma arquitetura modular com separação clara de responsabilidades, além de oferecer um ecossistema completo para gerenciamento de estados, rotas, formulários e comunicação com APIs REST [10]. Essas características foram decisivas para assegurar um desenvolvimento robusto, escalável e manutenível.

No *Backend*, optou-se pelo *framework* Spring Boot [54] (versão 3), reconhecido por acelerar o desenvolvimento de aplicações e proporcionar estruturação clara em camadas — controle, serviço, repositório e modelo —, favorecendo práticas organizadas e sustentáveis [63]. A maturidade do ecossistema Java, combinada à sua estabilidade e ampla adoção no mercado, representou um diferencial chave, especialmente para atender necessidades de confiabilidade, portabilidade e facilidade de manutenção.

Para a persistência de dados, escolheu-se o banco de dados relacional PostgreSQL [40], cuja natureza relacional atende diretamente à modelagem rigorosa e à integridade dos dados do sistema. Seu desempenho, aliado a recursos como transações ACID, suporte a extensões e conformidade com padrões SQL, o torna ideal para sistemas que demandam confiabilidade e consistência [25]. Ademais, o PostgreSQL é amplamente empregado em aplicações com georreferenciamento, graças à extensão PostGIS [34]. Como o Sidagro Web é uma aplicação governamental no setor agropecuário, é plausível a futura incorporação de dados espaciais — como áreas de produção, propriedades rurais ou rotas de fiscalização —, tornando-o ainda mais adequado.

Assim, o conjunto de tecnologias selecionado não só cumpre os requisitos funcionais do projeto, mas também reforça atributos essenciais de qualidade de software, como manutenibilidade, escalabilidade e portabilidade. Ao priorizar ferramentas maduras, com suporte comunitário amplo e alinhadas às boas práticas de desenvolvimento de software, o projeto estabelece uma base sólida para sua evolução contínua, mantendo elevados padrões de qualidade.

3 *Backend*

Esta seção apresenta e justifica as principais decisões de projeto relacionadas ao desenvolvimento do *Backend*, destacando como cada escolha técnica contribuiu para a qualidade do produto final. Nesse contexto, são detalhadas as padronizações estabelecidas para garantir consistência no desenvolvimento, promover uniformidade entre os desenvolvedores e assegurar a qualidade do código-fonte.

3.1 Formatação do Código

No que tange à formatação do código, a equipe definiu como referência o padrão de formatação Eclipse na versão 2.1 [13]. Essa padronização não visa restringir o uso da Eclipse como ambiente de desenvolvimento, mas sim, garantir que os desenvolvedores utilizem *plug-ins* compatíveis, disponíveis nas principais IDEs [15, 26], capazes de replicar as regras de formatação e indentação adotadas pelo Eclipse.

A adoção de um padrão único de formatação favorece a uniformidade do código-fonte, facilitando leitura, revisão e manutenção [30]. Essa prática também se alinha às recomendações de engenharia de software, que valorizam convenções consistentes para assegurar qualidade e produtividade nas equipes de desenvolvimento [14].

Além dos benefícios de legibilidade, a padronização reduz conflitos durante a integração. Configurações distintas de formatação automática introduzem alterações meramente cosméticas — como espaçamentos, quebras de linha e indentação — que o sistema de controle de versão interpreta como modificações substantivas. Tais diferenças geram conflitos de integração (i.e., conflitos de *merge*) desnecessários [61], exigindo que os desenvolvedores resolvam divergências sem impacto funcional, sobrecarregando o trabalho e dificultando as revisões, pois diferenças irrelevantes obscurecem mudanças de fato significativas. A resolução manual desses conflitos consome tempo e eleva o risco de erros, comprometendo a integridade do código-fonte.

Dessa forma, a padronização do estilo de formatação assegura legibilidade e coerência, independentemente da IDE utilizada, fortalecendo a consistência do projeto e a qualidade do produto final.

3.2 Transferência de Dados

Com o objetivo de padronizar a troca de informações entre as diferentes camadas da aplicação, definiu-se que os objetos de transferência de dados (DTOs) seriam implementados utilizando o recurso *record*, introduzido na linguagem Java a partir da versão 14 e estabilizado na versão 16 [36]. Os *records* permitem a criação de tipos imutáveis com sintaxe concisa, favorecendo a redução de repetição de código e a expressividade do código, além de promover maior segurança e previsibilidade na manipulação de dados.

A escolha por essa abordagem está alinhada às boas práticas de encapsulamento e imutabilidade em projetos orientados a serviços, especialmente em arquiteturas multicamadas, onde a transferência de dados entre controladores, serviços e repositórios deve ocorrer de maneira clara e segura [5].

Em situações que exigiram herança, optou-se pelo uso de classes tradicionais juntamente com a biblioteca Lombok [42], uma vez que o modelo *record* não oferece suporte a herança de classes, apenas de interfaces [37].

A biblioteca Lombok foi empregada para automatizar a geração de código repetitivo por meio de processamento de anotações em tempo de compilação. Essa abordagem fundamenta-se no padrão de inversão de dependência [35], no qual a biblioteca injeta automaticamente implementações declaradas através de anotações específicas na estrutura das classes. Na implementação dos DTOs, foram utilizadas quatro anotações fundamentais: *@Data*, responsável pela geração automática de métodos *get*, *set* e implementações padrão de *equals()*, *hashCode()* e *toString()*; *@SuperBuilder*, que possibilita a construção de padrões Builder compatíveis com hierarquias de herança; *@NoArgsConstructor* e *@AllArgsConstructor* que geram construtores sem parâmetros e com todos os parâmetros, respectivamente. A integração dessas anotações com a estrutura convencional de classes Java resultou em DTOs concisamente implementados, preservando a capacidade de herança, enquanto promove significativa redução de código repetitivo e melhoria na manutenibilidade do sistema.

Assim, a adoção combinada de *records* e classes tradicionais, em conjunto com o uso estratégico da biblioteca Lombok, mostrou-se eficaz para garantir clareza, segurança e flexibilidade na modelagem e transferência de dados. Essa abordagem contribuiu de forma significativa para a obtenção de um código-fonte mais enxuto e legível, ao mesmo tempo em que reforçou a robustez e a qualidade da arquitetura da aplicação.

3.3 Mapeamento

Com a adoção de objetos de transferência de dados (DTOs) como estratégia para desacoplamento entre as camadas da aplicação, tornou-se necessário estabelecer um mecanismo eficiente e seguro para a conversão entre entidades do modelo de domínio e seus respectivos DTOs, e vice-versa.

Para atender a esse requisito com foco na padronização e na qualidade do código, decidiu-se pela utilização da biblioteca MapStruct [29] que permite a geração automática de mapeamentos por meio de interfaces anotadas com *@Mapper*. A biblioteca realiza a implementação dos métodos de conversão com base em convenções de nomenclatura e tipos, reduzindo significativamente o código repetitivo e mitigando erros associados a mapeamentos manuais.

A utilização do MapStruct aumentou significativamente a confiabilidade no tratamento de dados, eliminando erros recorrentes em mapeamentos manuais e reduzindo retrabalho. Esses ganhos foram evidenciados na prática, já que desde a adoção do MapStruct não houve registro de problemas relacionados ao mapeamento no fluxo de desenvolvimento. Além de gerar automaticamente a maior parte dos campos — restando apenas casos excepcionais para implementação manual —, a biblioteca mantém a lógica de transformação declarativa e concentrada em pontos bem definidos do código, o que favorece a legibilidade, a produtividade e a manutenção do sistema. Essa abordagem reforça a separação de responsabilidades entre as camadas da aplicação e os princípios de arquitetura limpa [31], contribuindo para a robustez e a qualidade geral do projeto.

3.4 Tratamento de Exceções

O tratamento adequado de exceções é fundamental para garantir a robustez e a previsibilidade de sistemas de software, especialmente em arquiteturas orientadas a serviços REST. A padronização dessa prática contribui para a clareza na comunicação de falhas, facilita a manutenção do código e melhora a experiência dos consumidores das APIs.

No contexto do Sidagro Web, adotou-se um modelo simplificado e centralizado para o tratamento de exceções, com a implementação de um tratador global de exceções responsável por interceptar e tratar os erros de maneira unificada. Além disso, em vez de criar múltiplas classes específicas para cada tipo de erro, foram desenvolvidas estruturas genéricas capazes de abranger diferentes cenários de falha, o que reduziu significativamente a complexidade do código e favoreceu a reutilização e manutenção das rotinas de tratamento.

A centralização proporcionada pelo tratador global de exceções permitiu a definição de padrões claros para a manipulação de exceções, assegurando que as respostas às falhas fossem uniformes em toda a aplicação. Essa uniformidade mostrou-se especialmente vantajosa para a camada de *Frontend*, que passou a receber respostas padronizadas, facilitando o tratamento de erros e a exibição de mensagens adequadas aos usuários finais.

Com essa abordagem, a centralização e padronização do tratamento de exceções contribuíram para a qualidade do produto final (i) ao cobrir de forma completa todos os possíveis retornos no *Backend*, evitando a exposição de dados sensíveis, e (ii) ao fornecer ao usuário mensagens padronizadas e amigáveis. Além disso, essa estratégia simplificou a arquitetura interna do sistema e promoveu maior transparência e confiabilidade na comunicação entre as diferentes camadas da aplicação, reforçando a manutenibilidade e a robustez do sistema.

3.5 Auditoria

Sistemas governamentais que manipulam dados sensíveis e significativos, como o Sidagro Web, exigem mecanismos robustos de auditoria e rastreabilidade. A implementação de práticas que assegurem a integridade, a transparência e a responsabilidade sobre as alterações realizadas é essencial para manter elevados padrões de confiabilidade e conformidade, impactando diretamente a qualidade do produto final [47].

Nesse contexto, durante a fase inicial do desenvolvimento, a classe *ImaEntity* foi estruturada para centralizar os atributos essenciais à auditoria e os campos comuns a todos os modelos persistidos no sistema. Entre os atributos presentes na *ImaEntity*, destacam-se: *id*, responsável por identificar unicamente cada entidade no banco de dados; *created_date* e *last_modified_date* que armazenam, respectivamente, as datas de criação e de última modificação da entidade; e *version*.

O campo *version* é particularmente importante para o controle de concorrência otimista, garantindo que as alterações sejam persistidas apenas quando o valor da versão enviado na requisição corresponder ao valor presente no banco de dados [60]. Esse mecanismo previne inconsistências decorrentes de atualizações concorrentes, especialmente em ambientes multiusuário.

Considera-se a existência de uma entidade XPTO previamente registrada no banco de dados, conforme ilustrada na Listagem 1.

```
{
    "id": 1,
    "version": 0,
    "createdDate": "2025-04-04T14:49:08.118802",
    "lastModifiedDate": "2025-04-04T14:49:08.118802",
    "valor": "x",
    ...
}
```

Listagem 1: Atualização concorrente com controle otimista.

Supõe-se então que dois usuários tentem modificar a entidade simultaneamente, ambos enviando o campo *version* com valor 0. O primeiro a realizar a alteração terá sua requisição processada com sucesso, resultando no incremento do campo *version* para 1 no banco de dados. Caso o segundo usuário tente persistir sua alteração utilizando ainda a versão anterior, o mecanismo de controle de concorrência otimista do JPA identificará a divergência e rejeitará a atualização, disparando um erro de concorrência. Dessa forma, garante-se que modificações concorrentes não sobrescrevam alterações realizadas por outros usuários sem a devida sincronização, preservando a integridade e a consistência dos dados ao longo do ciclo de vida do sistema.

Além dos atributos mencionados, a estrutura foi preparada para incorporar futuramente os atributos *created_by* e *last_modified_by* que registrarão os usuários responsáveis pela criação e pela última modificação dos registros, quando o sistema passar a contar com autenticação de usuários. Adicionalmente, o sistema já foi preparado para integrar-se ao Hibernate Envers [23], uma ferramenta que permite a auditoria detalhada de todas as alterações realizadas nas entidades, proporcionando um histórico completo das modificações e reforçando a segurança e a rastreabilidade do sistema.

A classe *ImaEntity* foi definida como classe base para todos os modelos de dados do sistema, assegurando que cada entidade herde automaticamente os atributos comuns e de auditoria. Com essa estrutura, tornou-se possível a integração com o recurso da API de Persistência Java (JPA) *Auditing* [56] do Spring Boot, que automatiza o gerenciamento dos campos de auditoria [53], dispensando a necessidade de manipulação manual desses atributos durante as operações de persistência.

Em síntese, a adoção dessa abordagem centralizada proporcionou maior padronização, segurança e facilidade de manutenção. Essa abordagem contribuiu significativamente para a elevação da qualidade do sistema, ao garantir a conformidade com práticas recomendadas para sistemas que demandam alto grau de rastreabilidade e controle sobre as alterações realizadas nos dados.

3.6 Convenções de Código e Dados

A qualidade de software em sistemas de larga escala está intrinsecamente ligada à adoção de práticas rigorosas de modelagem de dados e ao estabelecimento de convenções de código claras e padronizadas [46]. No contexto do Sidagro Web, a definição criteriosa dessas diretrizes foi essencial para garantir não apenas a robustez do código-fonte, mas também para assegurar a legibilidade e a manutenibilidade do sistema ao longo de seu ciclo de vida.

Dentre as principais convenções adotadas para a modelagem das entidades persistentes nas fases iniciais do projeto, destaca-se, primeiramente, a definição explícita do tamanho das colunas por meio do parâmetro *length()* na anotação *@Column* da JPA, mesmo nos casos em que o valor padrão seria suficiente. Essa prática visa alinhar o modelo de dados às regras de negócio e promover a transparência das restrições impostas.

No que se refere à configuração dos relacionamentos entre entidades, optou-se pelo uso do valor padrão da JPA: *FetchType.LAZY*. Entretanto, devendo especificá-lo como parâmetro das anotações de relacionamento (*@OneToMany*, *@ManyToOne* e *@OneToOne*), visando mais uma vez a transparência das regras estabelecidas. Tal escolha busca otimizar o desempenho das consultas, evitando o carregamento desnecessário de dados e promovendo a escalabilidade do sistema, especialmente em operações de maior complexidade [65].

Outra diretriz relevante foi a determinação do uso consistente da referência *this* para o acesso a atributos de classe. Essa prática contribui para a clareza do código, facilitando a distinção entre atributos de instância, variáveis locais e parâmetros formais, além de promover a legibilidade e a uniformidade do código.

No tocante à persistência de enumerações (*Enums*), optou-se pelo armazenamento textual em detrimento do formato inteiro. Embora a representação por inteiros seja mais eficiente em termos de armazenamento, a persistência como texto favorece a compreensão e a transparência dos dados armazenados, facilitando processos de depuração, auditoria e análise manual do banco de dados.

Em relação à nomenclatura de tabelas e pacotes, estabeleceu-se o uso do singular (por exemplo, *Produtor* em vez de *Produtores*), de modo a refletir a semântica individual de cada entidade e simplificar a compreensão do modelo por novos desenvolvedores. Para coleções, convencionou-se o sufixo *List* (por exemplo, *produtorList*), evidenciando o tipo de dado manipulado e aprimorando a legibilidade do código.

A implementação rigorosa dessas convenções e práticas de modelagem de dados demonstrou-se determinante para a uniformidade, clareza e sustentabilidade do código-fonte, além de facilitar a manutenção e a evolução do sistema. Tais medidas revelaram-se essenciais para garantir a qualidade do produto final, sobretudo em projetos de larga escala e ambientes colaborativos.

3.7 Garantia da Aderência às Convenções

A definição de convenções claras e específicas representa apenas o ponto de partida para assegurar a qualidade do código-fonte em

sistemas de larga escala. É igualmente fundamental implementar mecanismos que garantam a adesão a essas diretrizes durante todo o processo de desenvolvimento. No contexto do Sidagro Web, foi adotada a ferramenta *CheckStyle* [8], disponível como *plug-in* para as principais IDEs.

A configuração do *CheckStyle* com um conjunto inicial de regras bem definidas possibilitou a automatização da verificação do código, promovendo aos desenvolvedores a identificação de eventuais violações às convenções estabelecidas. Essa abordagem contribui diretamente para a padronização e legibilidade, aspectos essenciais para a manutenção e evolução do sistema.

Adicionalmente, foi incorporada ao fluxo de versionamento uma etapa automatizada na *pipeline* de desenvolvimento, utilizando o *SonarQube* [52]. Essa ferramenta realiza uma análise adicional do código, verificando o cumprimento das regras definidas e identificando potenciais violações não detectadas nas etapas anteriores. Essa verificação complementar eleva a confiabilidade do processo e assegura aderência contínua aos padrões de desenvolvimento.

A integração dessas práticas ao fluxo de trabalho contribui para a consolidação de uma cultura de qualidade, essencial à sustentabilidade e à manutenibilidade do Sidagro Web.

3.8 Versionamento do Banco de Dados

Em projetos de larga escala desenvolvidos de forma incremental, a necessidade de alterações frequentes no esquema do banco de dados é inerente à evolução do sistema. A ausência de mecanismos sistemáticos para controlar tais modificações pode resultar em inconsistências entre ambientes, dificultar a replicação do banco de dados e comprometer a rastreabilidade das alterações realizadas ao longo do tempo.

No início do desenvolvimento, o *script* de construção do banco de dados foi gerado automaticamente pelo Hibernate [24] – *framework* responsável pelo mapeamento objeto-relacional (ORM), para acesso e persistência dos modelos no banco de dados –, o que resultou em um arquivo sem ordenação semântica das colunas e com restrições (*unique*, chave estrangeira, etc.) nomeadas por *hashes*. Esse formato dificultava a leitura, a compreensão e a manutenção do modelo de dados, especialmente em um contexto colaborativo e de longo prazo. Para solucionar esses problemas e aprimorar a clareza do esquema, o *script* gerado passou por uma revisão manual criteriosa, na qual as colunas foram reorganizadas para refletir uma ordem lógica e semântica, e as restrições foram renomeadas com termos legíveis e compreensíveis. Essa intervenção foi fundamental para facilitar futuras manutenções e contribuir para a qualidade e sustentabilidade do projeto ao longo de sua evolução.

Com o objetivo de mitigar riscos e promover maior controle sobre a evolução do banco de dados, foi adotada a biblioteca Flyway [44] para o gerenciamento de migrações no Sidagro Web. O Flyway permite que *scripts* de migração sejam versionados juntamente com o código-fonte da aplicação, garantindo que, a cada inicialização do sistema, todas as alterações estruturais pendentes — como criação ou modificação de tabelas, colunas e índices — sejam aplicadas de forma automática, ordenada e rastreável. Embora o Flyway não assegure a integridade dos dados em si, nem a reversibilidade completa de todas as operações, a ferramenta garante a consistência e a replicabilidade do esquema do banco de dados em diferentes ambientes, tais como desenvolvimento, homologação e produção.

No contexto do Sidagro Web, estabeleceu-se ainda uma convenção específica para o versionamento dos *scripts* de migração, utilizando a data e hora da submissão ao repositório como identificador, no formato AAAAMMDDHHMM (Ano, Mês, Dia, Hora e Minuto, respectivamente). Essa padronização foi adotada para evitar conflitos entre desenvolvedores na enumeração sequencial das migrações, uma vez que o *Flyway* considera não apenas o número da versão, mas também a ordem sequencial dos *scripts*. Dessa forma, evita-se que migrações inseridas fora de ordem gerem erros de execução ou sejam ignoradas pelo sistema de versionamento.

Em síntese, a adoção do *Flyway* e a padronização do processo de versionamento de migrações trouxeram benefícios significativos para a qualidade do software, promovendo padronização, rastreabilidade e auditoria detalhada das alterações no banco de dados. Além disso, a abordagem contribuiu para a redução de erros de configuração, simplificou o processo de integração de novos desenvolvedores e fortaleceu o ciclo de CI/CD, promovendo maior estabilidade e previsibilidade no ciclo de vida do sistema.

3.9 Testes Automatizados

A busca pela qualidade e confiabilidade no desenvolvimento do Sidagro Web fundamentou a adoção de uma abordagem estruturada e disciplinada para testes automatizados no *Backend*. Desde as etapas iniciais do desenvolvimento, foi estabelecido que todo desenvolvedor, ao implementar uma nova funcionalidade, deve também criar os testes de unidade e de integração correspondentes, assegurando que cada componente do sistema seja devidamente validado em diferentes níveis.

Os testes de unidade são responsáveis por verificar o comportamento isolado de métodos e classes, permitindo a detecção precoce de defeitos e facilitando a manutenção do código [59]. Adotou-se o JUnit [27], por ser um *framework* consolidado com integração direta a ferramentas de *build* e IDEs. Em conjunto, utilizou-se o Mockito [32], cuja principal vantagem é a facilidade na simulação de dependências, possibilitando maior agilidade no desenvolvimento. Já os testes de integração, que avaliam a interação entre diferentes módulos e a comunicação com recursos externos, como bancos de dados [28], foram apoiados pelo uso do Testcontainers [58].

O fluxo de desenvolvimento de testes no *Backend* segue o padrão AAA (*Arrange, Act, and Assert*) que organiza os testes em três etapas bem definidas: preparação do cenário (*Arrange*), execução da ação a ser testada (*Act*) e verificação dos resultados esperado (*Assert*) [64]. Esse padrão foi adotado por favorecer a legibilidade e a padronização dos testes, facilitando a colaboração entre membros da equipe e a detecção de falhas. Além disso, os testes são integrados às *pipelines* de CI/CD, permitindo que cada alteração submetida ao repositório seja automaticamente validada, promovendo ciclos de resposta rápidos e reduzindo o risco de regressões.

A adoção sistemática de testes automatizados trouxe benefícios concretos ao projeto, incluindo o aumento da cobertura de testes, detecção ágil de falhas, maior confiança nas entregas e facilidade de evolução do sistema. Ao equilibrar testes manuais e automatizados, a equipe garantiu um produto robusto, sustentável e alinhado aos padrões de qualidade estabelecidos para o Sidagro Web.

4 *Frontend*

Visando assegurar a consistência, manutenibilidade e qualidade da interface do usuário no Sidagro Web, foram estabelecidas diretrizes para orientar o desenvolvimento do *Frontend*. Essas orientações buscam promover uniformidade entre os desenvolvedores, resultando em uma experiência final coesa e alinhada aos padrões de excelência estabelecidos para o projeto. Nesta seção, são apresentados e justificados os principais critérios técnicos e decisões adotadas no *Frontend*.

4.1 Estilização de Componentes

A definição de um padrão consistente para os estilos dos componentes é fundamental para garantir a uniformidade visual e a experiência do usuário. Nesse contexto, foi estabelecida a utilização da biblioteca Material Design [18] como base para a criação dos componentes de interface.

A escolha pelo Material Design foi motivada por sua maturidade, ampla adoção em projetos de larga escala e manutenção contínua pela equipe do Google, o que assegura atualizações regulares e aderência a padrões consolidados de usabilidade e acessibilidade. Além disso, a biblioteca oferece um conjunto abrangente de componentes prontos e personalizáveis, facilitando a implementação de interfaces coesas e alinhadas às melhores práticas do mercado. Essa padronização também se mostrou vantajosa do ponto de vista da equipe de desenvolvimento, pois proporciona uma solução estável, bem documentada e de fácil assimilação, reduzindo o tempo de adaptação e o risco de inconsistências visuais entre diferentes partes do sistema.

Em síntese, a adoção do Material Design como referência para os estilos dos componentes contribuiu de maneira significativa para a qualidade visual, a previsibilidade e a manutenibilidade da interface.

4.2 Gerenciamento de Estilos

Após a definição do Material Design como base para a estilização de componentes, perdura a necessidade de assegurar a padronização e o gerenciamento eficiente dos estilos do sistema. Dessa forma, foi adotado o pré-processador SCSS [48] como solução para a organização e manutenção dos estilos da aplicação.

A escolha pelo SCSS se justifica por sua capacidade de incorporar ao desenvolvimento de estilos recursos avançados, como variáveis, aninhamento de seletores e modularização. Tais funcionalidades – ausentes no CSS tradicional – são essenciais para a construção de uma base de estilos escalável, reutilizável e de fácil manutenção, especialmente em projetos com múltiplos colaboradores. A possibilidade de segmentar estilos em módulos e reutilizar trechos de código contribui para a redução de redundâncias e para o aumento da previsibilidade no comportamento visual da aplicação.

Como resultado, a adoção do SCSS proporcionou um código de estilos mais legível, estruturado e sustentável, favorecendo a manutenção evolutiva e promovendo a consistência visual em todas as interfaces do sistema, elevando a qualidade do produto final.

4.3 Tratamento de Exceções

A padronização do tratamento de exceções no *Frontend* é fundamental para garantir a robustez, a previsibilidade e a qualidade da experiência do usuário em aplicações web. No contexto do Sidagro Web, foi adotada uma abordagem centralizada para o tratamento de erros provenientes da comunicação com a API REST, por meio da implementação de um interceptador de requisições HTTP (*Hypertext Transfer Protocol*), denominado *HttpErrorInterceptor*.

Essa decisão arquitetural permitiu concentrar em um único ponto a lógica de captura e tratamento de respostas de erro das requisições HTTP. O *HttpErrorInterceptor* intercepta todas as respostas de erro e, com base no código de status retornado (por exemplo, 400, 401, 404 e 500), aciona um serviço de notificações modularizado (*SnackbarService*) responsável por exibir mensagens padronizadas ao usuário. Esse serviço apresenta cartões informativos com design indicativo, promovendo clareza e consistência na resposta fornecida ao usuário final, independentemente da tela ou funcionalidade em que a falha ocorra.

A adoção dessa estratégia mostrou-se crucial para a robustez e a manutenibilidade do sistema, pois, ao centralizar o tratamento de erros, os componentes de interface puderam concentrar-se exclusivamente na lógica de apresentação e interação.

4.4 Validação de Formulários

A validação de formulários desempenha papel relevante na promoção da qualidade e da integridade dos dados em sistemas web [22]. No desenvolvimento do Sidagro Web, buscou-se incorporar mecanismos de validação diretamente na interface, por meio de validadores customizados aplicados aos formulários reativos do Angular. Embora o *framework* já disponibilize uma gama de validadores padrão, foram desenvolvidas regras específicas para atender às particularidades do domínio da aplicação – tais como *cpfValidator*, *cnpjValidator*, *pastDateValidator* e validadores condicionais (*requiredWhen* e *when*) – que ampliam a flexibilidade e a aderência às regras de negócio.

É importante ressaltar, entretanto, que a validação realizada no *Frontend* tem como principal objetivo aprimorar a experiência do usuário e evitar o envio de dados evidentemente inválidos ao *Backend*. Essa camada de validação atua como um facilitador, fornecendo a resposta imediata e orientando o preenchimento correto dos formulários, além de reduzir requisições desnecessárias e minimizar o risco de sobrecarga da API. No entanto, a responsabilidade final pela integridade e segurança dos dados permanece no *Backend*, que é projetado para validar rigorosamente todas as requisições recebidas, independentemente de sua origem. Tal abordagem é fundamental, pois o *Backend* pode ser acessado por diferentes interfaces, incluindo aplicações web, aplicativos móveis e integrações externas, e deve estar preparado para lidar com as mais variadas entradas.

Assim, a validação no *Frontend* deve ser compreendida como uma camada adicional de proteção e usabilidade, que complementa, mas não substitui, os mecanismos de validação implementados no *Backend*. Essa estratégia integrada contribui para a robustez, a confiabilidade e a qualidade geral do sistema, assegurando que apenas dados válidos e consistentes sejam efetivamente processados e armazenados.

4.5 Enumerações e Dados Dinâmicos

Para garantir a consistência e facilitar a manutenção do Sidagro Web, as listas dinâmicas – como estados, cidades e outros atributos

selecionáveis – foram centralizadas no *Backend*. Em vez de manter essas informações diretamente no *Frontend*, o *Backend* fornece esses dados por meio de rotas específicas, entregando tanto as descrições quanto os identificadores necessários para a interface. Após o primeiro carregamento, esses dados são armazenados em cache no *Frontend*, permitindo seu reaproveitamento e melhorando o desempenho.

Dessa forma, o *Frontend* atua exclusivamente como consumidor dessas informações, sem lógica embutida ou dependência de dados internos, o que reduz o risco de inconsistências e simplifica atualizações. Essa abordagem também permite que diferentes clientes reutilizem as mesmas informações, promovendo maior flexibilidade e portabilidade.

Em síntese, a centralização do fornecimento de enumerações e dados dinâmicos no *Backend* consolidou uma arquitetura mais robusta, previsível e alinhada aos princípios de qualidade do projeto, eliminando redundâncias e garantindo que o *Frontend* permaneça desacoplado das regras de negócio, focando exclusivamente na apresentação e interação com o usuário.

4.6 Acessibilidade

A acessibilidade constitui um aspecto essencial no desenvolvimento de sistemas de software, garantindo que pessoas com diferentes habilidades possam interagir de forma eficiente e satisfatória com a aplicação [1, 9]. No contexto do nosso projeto, inicialmente, a incorporação de práticas de acessibilidade ainda não havia sido implementada, refletindo uma fase inicial de desenvolvimento centrada principalmente em funcionalidades e desempenho técnico.

Posteriormente, medidas específicas de acessibilidade — e.g., avaliações heurísticas voltadas à usabilidade, compatibilidade com leitores de tela e adoção de padrões de design inclusivo — foram progressivamente integradas ao fluxo de desenvolvimento. Essas ações visaram não apenas atender a requisitos legais e normativos, mas também proporcionar uma experiência mais inclusiva e uniforme para todos os usuários, independentemente de suas capacidades físicas ou cognitivas. A implementação dessas práticas resultou em melhorias tangíveis na experiência do usuário e reforçou o compromisso da equipe com o desenvolvimento de software inclusivo e sustentável, em consonância com boas práticas de qualidade técnica e responsabilidade social.

5 Infraestrutura

Decisões relativas à infraestrutura e à definição de ambientes – alinhadas entre o IMA, a Prodemge e a equipe técnica do projeto – são fundamentais para sustentar a qualidade, a manutenção e a evolução contínua de sistemas de informação críticos, como o Sidagro Web. Esta seção tem o objetivo de apresentar a arquitetura adotada, em conformidade institucional, de modo a maximizar a confiabilidade das operações e mitigar riscos ao longo do ciclo de vida do software, ao mesmo tempo em que facilita a validação incremental das entregas e promove a melhoria contínua do produto.

5.1 Ambientes e Fluxo de CI/CD

A infraestrutura do Sidagro Web foi projetada para assegurar qualidade, rastreabilidade e estabilidade nas entregas, por meio da adoção de múltiplos ambientes independentes. Cada ambiente é implementado em uma máquina virtual dedicada, garantindo isolamento entre contextos e controle rigoroso de versões em todas as fases do ciclo de vida do sistema.

Essa estrutura compreende três ambientes principais:

- **Desenvolvimento:** destinado à integração contínua, testes preliminares e experimentação de novas funcionalidades pela equipe técnica.
- **Homologação:** voltado à validação funcional e não funcional pela equipe de qualidade e pelo cliente, assegurando conformidade com os requisitos antes da promoção para produção.
- **Produção:** dedicado ao funcionamento estável do sistema, acessível ao público e sujeito a padrões elevados de disponibilidade e segurança.

Tal configuração promove um fluxo de evolução do software mais seguro, prevenindo interferências entre etapas e minimizando riscos de impactos em produção oriundos de alterações ainda em desenvolvimento ou validação.

O processo de integração e entrega contínua (CI/CD) é orquestrado por uma *pipeline* baseada no GitLab. A cada *merge* nas principais *branches* do repositório (*develop*, *homolog* e *main*), a *pipeline* é acionada automaticamente, padronizando o ciclo de validação e publicação das versões, conforme ilustrado na Figura 1.

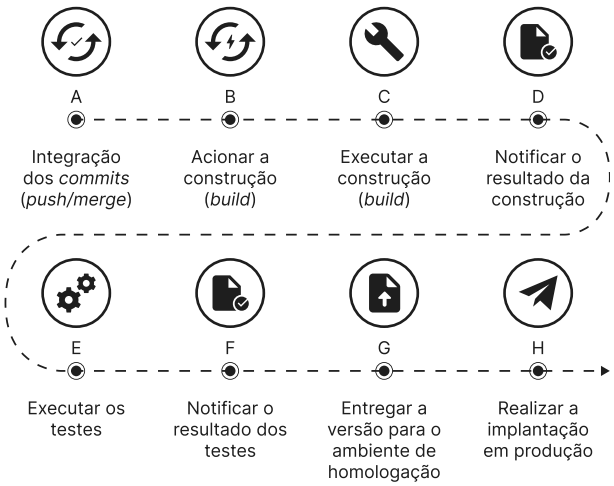

Figura 1: Fluxo de CI/CD representando as etapas automatizadas desde a integração de código até a entrega em produção.

Conforme ilustrado na Figura 1, a execução da *pipeline* inicia-se a partir de *commits* ou *merges* (A), acionando o processo de construção (*build*) da imagem da aplicação Java ou Angular por meio do respectivo *Dockerfile* (B e C). Ao final dessas etapas, o sistema reporta o status do processo, indicando sucesso ou falha (D). Em seguida, executam-se os testes automatizados (E), acompanhados de notificações sobre o resultado obtido (F). O avanço para as fases subsequentes ocorre apenas se todas as etapas anteriores forem concluídas com êxito, permitindo a publicação da nova versão no ambiente de homologação (G) e, posteriormente, em produção (H). Essa execução sequencial garante que qualquer falha interrompa

imediatamente o processo, minimizando o risco de implantação de versões instáveis.

Na entrega contínua, as imagens validadas são implantadas automaticamente nas máquinas virtuais de cada ambiente, assegurando rastreabilidade, padronização e agilidade em procedimentos de *rollback*, quando necessário. Esse ciclo otimiza o gerenciamento de versões e reforça o controle sobre o histórico das implantações.

A integração entre ambientes e o fluxo automatizado e robusto de CI/CD revelaram-se essenciais para promover qualidade, previsibilidade e transparência no ciclo de vida do Sidagro Web. Essa abordagem reduz a ocorrência de falhas, acelera o tempo de entrega de novas versões e facilita a identificação e correção rápida de problemas, conferindo maior confiança ao processo de desenvolvimento e implantação do sistema. Dessa forma, reforça o alinhamento do projeto com os critérios de qualidade estabelecidos, garantindo entregas consistentes, ágeis e seguras.

5.2 Infraestrutura Computacional

Um dos pilares para assegurar a robustez e a disponibilidade do Sidagro Web reside em sua infraestrutura computacional. A definição dos recursos foi guiada pela necessidade de fornecer ambientes com desempenho consistente, alta confiabilidade e capacidade de adaptação à evolução do sistema.

Cada ambiente da arquitetura — desenvolvimento, homologação e produção — é provisionado com máquinas virtuais dedicadas. Essas instâncias foram configuradas com 16 GB de memória RAM e *vCPUs* alocadas conforme o perfil e a criticidade de cada etapa, promovendo a execução eficiente de cargas de trabalho.

Essa abordagem modular garante não apenas a adequação da capacidade computacional, mas também contribui para a escalabilidade do sistema. A flexibilidade para ajustar recursos conforme as demandas específicas de cada ambiente permite respostas ágeis a desafios operacionais, além de mitigar riscos de indisponibilidades ou degradação de desempenho.

O planejamento criterioso da infraestrutura computacional revela-se fator determinante para o Sidagro Web, viabilizando operações seguras, confiáveis e eficientes, contribuindo para a qualidade geral do software.

5.3 Infraestrutura do Sistema

Essa seção apresenta a arquitetura da infraestrutura que sustenta o Sidagro Web, trazendo um panorama dos principais componentes e suas relações, conforme ilustrado na Figura 2.

A representação oferece uma visão geral da arquitetura de infraestrutura do Sidagro Web, incluindo contêineres, ferramentas de *DevOps* e os principais fluxos de comunicação entre os componentes do sistema. Os detalhes de cada elemento representado, bem como suas funções e interações, são aprofundados nas subseções seguintes, proporcionando uma compreensão completa da solução.

5.4 Integração entre *Frontend*, *Backend* e Serviços Intermediários

A arquitetura do Sidagro Web foi projetada com ênfase no desacoplamento, na escalabilidade e na segurança, particularmente na comunicação entre seus componentes principais: *Frontend*, *Backend*

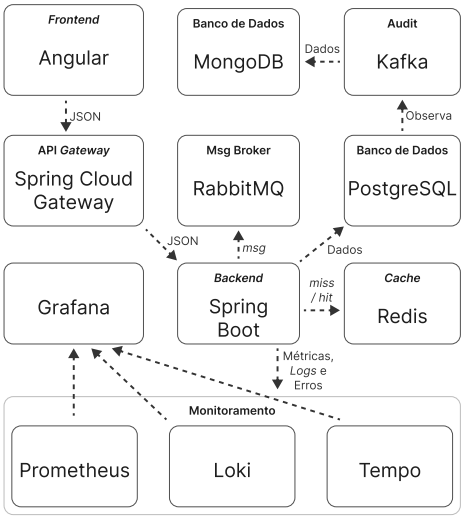

Figura 2: Arquitetura da infraestrutura do sistema, ilustrando as comunicações entre os diversos componentes.

e serviços intermediários. Tanto o *Frontend* (Angular) quanto o *Backend* (Spring Boot) são modularizados em contêineres, o que reforça a independência entre os serviços e facilita a gestão, a implantação e a evolução do sistema. Essa estrutura não apenas promove maior eficiência no fluxo de dados, mas também garante que o Sidagro Web se adapte facilmente a variações de demanda, ampliando sua escalabilidade.

O Angular atua como interface de interação com o usuário e se conecta ao *Backend* por meio do Spring Cloud Gateway [55]. Esse *gateway* serve como ponto de entrada unificado para todas as requisições destinadas à plataforma, roteando, autenticando e controlando o tráfego entre o *Frontend* e o *Backend*. Essa abordagem proporciona maior segurança e transparência nas integrações, além de permitir a implementação centralizada de políticas de controle de acesso e monitoramento.

No *Backend*, o Spring Boot estabelece comunicação eficiente com componentes de *middleware* que sustentam funcionalidades críticas do sistema. O Redis [45] é utilizado como cache, reduzindo a latência nas respostas e aprimorando a escalabilidade da aplicação. Essa integração ocorre de forma transparente, por meio de bibliotecas e conectores específicos, permitindo acesso rápido e gerenciamento eficiente de dados voláteis.

Outro elemento central é o RabbitMQ [38], empregado para mensageria e integração assíncrona entre os diversos microsserviços. A comunicação com o RabbitMQ se realiza por meio de filas e tópicos, garantindo que operações que demandam processamento desacoplado ou que não impactem o fluxo síncrono da aplicação sejam gerenciadas de forma resiliente e escalável. Esse padrão é essencial para preservar a integridade e a fluidez do processamento de tarefas em sistemas distribuídos.

Em suma, a utilização do Spring Cloud Gateway como camada de entrada, aliada à comunicação robusta do *Backend* com Redis e RabbitMQ, permite que o Sidagro Web opere de maneira coesa, eficiente e resiliente. Essa combinação tecnológica assegura uma experiência de usuário fluida, ao mesmo tempo em que facilita

a manutenção, a escalabilidade e o monitoramento contínuo do sistema.

5.5 Observabilidade do Sistema

A observabilidade do Sidagro Web foi estruturada para proporcionar visibilidade abrangente e em tempo real tanto da infraestrutura quanto das aplicações, promovendo monitoramento contínuo, análise detalhada e resposta proativa a incidentes. Para atingir esse objetivo, diferentes ferramentas atuam de forma integrada nos principais pilares da observabilidade: métricas, *logs* e rastreamentos.

O Prometheus [43] desempenha um papel fundamental na coleta e armazenamento de métricas detalhadas, como uso de CPU, consumo de memória e latência de serviços. Essa funcionalidade permite o acompanhamento contínuo da saúde do ambiente, a identificação de tendências e a antecipação de possíveis gargalos de desempenho. As métricas extraídas orientam a equipe técnica na análise preditiva de capacidade e no ajuste fino dos recursos de infraestrutura.

A coleta, agregação e indexação centralizada dos *logs* são realizadas por meio do Loki [19], que registra eventos de aplicações e componentes de infraestrutura de maneira estruturada. Esse mecanismo facilita buscas ágeis, a correlação de eventos relevantes e a auditoria de ocorrências, promovendo maior capacidade de diagnóstico para comportamentos inesperados ou falhas operacionais.

No contexto de sistemas distribuídos, o rastreamento do fluxo de requisições entre microsserviços é efetuado pelo Tempo [20]. Essa ferramenta de *tracing* permite observar a trajetória completa das chamadas, auxiliando na identificação de pontos de latência, dependências críticas e falhas em cadeias de execução complexas, o que é vital para a manutenção de aplicações modernas baseadas em microsserviços.

Para consolidar todas essas informações, o Grafana oferece painéis interativos e integrados, reunindo dados de métricas, *logs* e rastreamentos em uma interface única e intuitiva. Além da visualização holística do ambiente, o Grafana [21] possibilita a criação de alertas, o acompanhamento de indicadores cruciais de desempenho e a análise de tendências históricas, apoiando decisões de refinamento contínuo na operação do sistema.

Por exemplo, o Grafana foi integrado a um *bot* do Telegram [57], possibilitando o envio imediato de alertas à equipe sempre que uma falha crítica for identificada, como a indisponibilidade do servidor. Essa integração assegura que incidentes relevantes sejam comunicados em tempo real, promovendo respostas ágeis e, consequentemente, minimizando o tempo de indisponibilidade do sistema.

A integração desses componentes proporciona um alto grau de transparência operacional, fortalece a detecção proativa de problemas e agiliza o diagnóstico e a resposta a incidentes. Assim, a arquitetura de observabilidade do Sidagro Web sustenta práticas modernas alinhadas às necessidades de ambientes críticos, garantindo robustez e confiabilidade ao longo de todo o ciclo de vida do sistema.

5.6 Mecanismo de Auditoria

Com o objetivo de garantir auditoria eficiente e plena rastreabilidade das modificações ocorridas nos dados do sistema, foi implementado um mecanismo de *Change Data Capture* (CDC) [49]. Essa solução é essencial para atender às exigências regulatórias e possibilitar análises forenses detalhadas do histórico de alterações em ambientes críticos de produção. A Figura 3 representa de forma clara o funcionamento desse mecanismo, evidenciando como o Sidagro Web monitora e registra, de maneira contínua e confiável, todas as mudanças efetuadas sobre os seus dados.

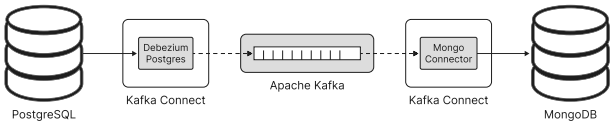

Figura 3: Comunicação entre os serviços que compõem o mecanismo de auditoria.

Como ilustrado na figura, a estratégia adotada envolve o uso do Debezium, uma ferramenta de monitoramento que atua diretamente sobre o *write-ahead log* (WAL) do banco de dados PostgreSQL. Por operar no nível dos *logs* de transação, o Debezium captura de forma eficiente e em tempo real todas as operações de criação, atualização e remoção de registros, sendo desnecessário monitorar consultas, o que garante um impacto mínimo no desempenho do banco de dados transacional [12].

As alterações detectadas são serializadas e publicadas como eventos em tópicos do Apache Kafka, permitindo processamento assíncrono e escalável dessas informações. Para assegurar a persistência e integridade do histórico, um serviço consumidor lê os eventos do Kafka e os armazena em uma coleção dedicada no banco de dados MongoDB. Essa arquitetura desacoplada garante que o registro de auditoria seja completo, imutável e disponível para consultas retroativas detalhadas sobre a evolução de qualquer registro do sistema.

Dessa forma, o mecanismo de auditoria implementado consolida uma trilha de evidências robusta e confiável, fortalecendo o compromisso do Sidagro Web com a transparência, a governança de dados e a aderência a padrões elevados de qualidade.

6 Práticas Transversais

Nesta seção, são apresentadas as práticas transversais que fundamentam a qualidade e a sustentabilidade do projeto Sidagro Web, abrangendo o versionamento de código e a sistematização da documentação técnica.

6.1 Versionamento de Código

A gestão eficiente do versionamento de código é essencial para garantir a qualidade em projetos de software, especialmente em contextos com dados sensíveis e requisitos elevados de segurança e controle. No desenvolvimento do Sidagro Web, tanto para o *Frontend* quanto para o *Backend*, optou-se pelo GitLab [17] em sua versão auto-hospedada nos servidores da UFLA.

A decisão por uma solução auto-hospedada foi motivada principalmente pela necessidade de manter o ambiente de versionamento sob controle institucional total, reduzindo riscos de exposição do código-fonte e de dados sensíveis. Essa abordagem assegura maior segurança, pois todo o ciclo de desenvolvimento — desde o armazenamento de repositórios até a execução de *pipelines* de Integração e

Entrega Contínuas (CI/CD) [51] — permanece restrito à infraestrutura da universidade. Ademais, contribui para a redução de custos operacionais, permitindo configurações personalizadas de repositórios, *pipelines* e *runners* sem cobranças adicionais, com todos os recursos gerenciados internamente.

Adicionalmente, adotaram-se práticas consolidadas de versionamento e organização de *branches* para promover rastreabilidade e clareza no ciclo de vida do software. O projeto segue o padrão de Versionamento Semântico (*SemVer*) [50], no qual as versões são identificadas pelo formato *Major.Minor.Patch* (x.y.z). Nesse esquema, o incremento do número principal (*Major*) indica mudanças incompatíveis na API; o secundário (*Minor*), adições de funcionalidades retro-compatíveis; e o terciário (*Patch*), correções de *bugs* e ajustes menores [3]. Essa padronização facilita a comunicação sobre o impacto das alterações e fornece à equipe de testes uma referência clara sobre a versão em análise, promovendo maior controle durante a validação e homologação.

Quanto à organização do fluxo de trabalho, adotou-se o modelo GitFlow. Nesse modelo, o desenvolvimento de novas funcionalidades ocorre em *branches* do tipo *feature*, enquanto correções de defeitos são tratadas em *bugfix*. Ambas convergem para a *develop*, que reúne as alterações em andamento e, ao final de cada ciclo, é integrada à *homolog*, ambiente destinado à validação pela equipe de testes. Após a aprovação, as mudanças consolidadas são integradas à *main*, representando a versão estável em produção. Esse fluxo estruturado promove isolamento adequado de funcionalidades, reduz o risco de regressões e fortalece a confiabilidade e a manutenibilidade do sistema, além de permitir que cada alteração seja facilmente rastreada e associada à respectiva tarefa.

Em síntese, a combinação do GitLab auto-hospedado, do *SemVer* e do GitFlow proporcionou ao Sidagro Web um ambiente de desenvolvimento seguro, rastreável, flexível e economicamente sustentável, alinhado às necessidades do projeto e aos princípios de qualidade de software. Essa estratégia fortaleceu o controle sobre o ciclo de vida do código-fonte e contribuiu para a confiabilidade e eficiência operacional do sistema.

6.2 Documentação

A documentação técnica exerce papel fundamental na garantia da qualidade e sustentabilidade de sistemas de larga escala como o Sidagro Web. Mais do que complementar a implementação e padronização do código-fonte, a documentação representa um recurso estratégico para a manutenção do produto, a transferência de conhecimento e o alinhamento da equipe em torno das melhores práticas de engenharia de software [11].

No contexto deste projeto, a documentação foi estruturada em artefatos práticos e acessíveis, abrangendo desde a arquitetura do sistema e convenções de código até fluxos de trabalho, guias técnicos para o consumo das APIs e instruções de configuração de ambientes. Esses documentos orientam o desenvolvimento, padronizam nomenclaturas, detalham processos de versionamento e fornecem suporte específico tanto para a integração de novos membros quanto para o esclarecimento de dúvidas recorrentes da equipe. A centralização desses materiais em uma plataforma dedicada, o *Notion* [33], assegura organização, controle de acesso e atualização contínua do repositório de conhecimento.

Em síntese, esse conjunto de medidas resultou em maior autonomia dos desenvolvedores, redução na curva de aprendizado e na dependência de interações para esclarecimento de dúvidas, além de garantir consistência, colaboração e alinhamento permanente com os padrões de qualidade estabelecidos no projeto. Dessa forma, a formalização e centralização da documentação configuram-se como pilares essenciais para a evolução sustentável do Sidagro Web.

7 Considerações Finais

Este artigo apresentou um relato técnico detalhado sobre o desenvolvimento do sistema Sidagro Web, conduzido por meio de uma parceria institucional entre a Universidade Federal de Lavras (UFLA) e o Instituto Mineiro de Agropecuária (IMA). As decisões arquiteturais e técnicas adotadas ao longo de todo o ciclo de vida do projeto foram guiadas por princípios de qualidade de software, com ênfase em requisitos não funcionais como modularidade, escalabilidade, confiabilidade e manutenibilidade.

Como principal contribuição, este trabalho configura-se como um guia prático de boas práticas para projetos com alta demanda por qualidade. As definições realizadas nos âmbitos de decisões de alto nível, *Backend*, *Frontend*, infraestrutura e práticas transversais foram sistematizadas de forma a promover organização, padronização e robustez – elementos essenciais em sistemas corporativos de larga escala. Tais diretrizes consolidam um conjunto de soluções técnicas e metodológicas que podem orientar futuras iniciativas institucionais de desenvolvimento de sistemas web, incorporando as lições aprendidas ao longo do projeto.

Além disso, o material produzido apresenta elevado potencial de reaproveitamento em contextos semelhantes, tanto no setor público quanto no privado, independentemente do regime de licenciamento adotado. Embora o sistema não seja de código aberto, por estar vinculado a uma autarquia estadual, suas definições e decisões técnicas preservam aplicabilidade geral, podendo subsidiar equipes de desenvolvimento em diferentes organizações. Ao registrar e justificar decisões críticas tomadas durante o processo, este relato estabelece uma base sólida para novas implementações, favorecendo ciclos iniciais mais ágeis e reduzindo riscos relacionados à qualidade e à sustentabilidade do sistema. Na experiência do projeto, tais práticas têm se mostrado estáveis e eficazes, não havendo até o momento necessidade de revisões significativas — o que evidencia sua maturidade e efetividade na prática.

Por fim, espera-se que este relato contribua para a consolidação de experiências que possam ser discutidas, adaptadas e implementadas por outras organizações, além de servir como insumo relevante para pesquisas futuras no campo da Qualidade de Software.

ACKNOWLEDGMENTS

Os autores agradecem à Universidade Federal de Lavras (UFLA), ao Instituto Mineiro de Agropecuária (IMA) e à Fundação de Desenvolvimento Científico e Cultural (FUNDECC) – Convênio nº 24/2024 pelo apoio para realização deste trabalho.

REFERENCES

[1] Amaia Aizpurua, Simon Harper, and Markel Vigo. 2016. Exploring the relationship between web accessibility and user experience. *International Journal of Human-Computer Studies* 91 (2016), 13–23.

[2] Angular Team. 2025. *Angular — Overview.* https://angular.dev/overview Acesso em: 5 jul. 2025.

[3] Baeldung. 2025. Semantic Versioning. Disponível em: <https://www.baeldung.com/cs/semantic-versioning>. Acesso em: 2 jul. 2025.

[4] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3 ed.). Addison-Wesley.

[5] Joshua Bloch. 2008. *Effective Java* (2 ed.). Addison-Wesley.

[6] Neil Brown, Robert Nord, Philippe Kruchten, and Ipek Ozkaya. 2010. Managing Technical Debt in Software-Reliant Systems. In *2nd Workshop on Future of Software Engineering Research (FoSER)*. 47–52.

[7] Daniel C. Carvalho and Guilherme H. Travassos. 2007. Análise da qualidade de artefatos de documentação de software: um estudo de caso em sistemas governamentais. In *XXI Simpósio Brasileiro de Engenharia de Software (SBES)*. 180–194.

[8] Checkstyle Community. 2025. *Checkstyle — Java Code Style Checker.* https://checkstyle.sourceforge.io Acesso em: 5 jul. 2025.

[9] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. 2001. Web content accessibility guidelines 1.0. *Interactions* 8, 4 (2001), 35–54.

[10] Jelica Cincovic, Sanja Delcev, and Drazen Draskovic. 2019. Architecture of Web Applications Based on Angular Framework: A Case Study. *Methodology* 7, 7 (2019), 254–259.

[11] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. 2003. Documenting software architectures: views and beyond. In *25th International Conference on Software Engineering (ICSE)*. 740–741.

[12] Luan Carvalho de Araújo Coelho. 2023. Uma arquitetura de código aberto para a aplicação do Strangler Pattern com Change Data Capture para setores com alto volume de dados. Trabalho de Conclusão de Curso (Graduação em Engenharia de Software), Universidade Federal do Ceará.

[13] Eclipse Foundation. 2025. *Eclipse IDE.* https://www.eclipse.org/ Acesso em: 5 jul. 2025.

[14] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

[15] GeeksforGeeks. 2024. 10 Best Java IDE For Developers in 2025. Disponível em: <https://www.geeksforgeeks.org/best-java-ide-for-developers>. Acesso em: 6 jun. 2025.

[16] G Geetha, Monisha Mittal, K Mohana Prasad, and J Godwin Ponsam. 2022. Interpretation and Analysis of Angular Framework. In *3rd International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)*. 1–6.

[17] GitLab Inc. 2025. *About GitLab.* https://about.gitlab.com/ Acesso em: 5 jul. 2025.

[18] Google. 2025. *Material Design 3.* https://m3.material.io Acesso em: 5 jul. 2025.

[19] Grafana Labs. 2025. *Grafana Loki — Log Aggregation System.* https://grafana.com/docs/loki/latest Acesso em: 5 jul. 2025.

[20] Grafana Labs. 2025. *Grafana Tempo — Distributed Tracing Backend.* https://grafana.com/docs/tempo/latest Acesso em: 5 jul. 2025.

[21] Grafana Labs. 2025. *Grafana — Observability Platform.* https://grafana.com/docs/grafana/latest Acesso em: 5 jul. 2025.

[22] Danny M Groenewegen and Eelco Visser. 2013. Integration of data validation and user interface concerns in a DSL for web applications. *Software & Systems Modeling* 12, 1 (2013), 35–52.

[23] Hibernate Team. 2025. *Hibernate Envers.* https://hibernate.org/orm/envers Acesso em: 5 jul. 2025.

[24] Hibernate Team. 2025. *Hibernate ORM.* https://hibernate.org Acesso em: 5 jul. 2025.

[25] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. 2012. A Survey and Comparison of Relational and Non-Relational Database. *International Journal of Engineering Research & Technology* 1, 6 (2012), 1–5.

[26] JetBrains Plugin Repository. 2025. *Adapter for Eclipse Code Formatter — IntelliJ Plugin.* https://plugins.jetbrains.com/plugin/6546-adapter-for-eclipse-code-formatter Acesso em: 5 jul. 2025.

[27] JUnit Team. 2025. JUnit. https://junit.org. Acesso em: 10 out. 2025.

[28] Hareton KN Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *6th International Conference on Software Maintenance*. 290–301.

[29] MapStruct Contributors. 2025. *MapStruct — Java Bean Mapping.* https://mapstruct.org/ Acesso em: 5 jul. 2025.

[30] Robert C. Martin. 2011. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall.

[31] Robert C. Martin. 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall.

[32] Mockito Team. 2025. Mockito Framework. https://site.mockito.org. Acesso em: 10 out. 2025.

[33] Notion Labs Inc. 2025. *Notion — Connected Workspace.* https://www.notion.com Acesso em: 5 jul. 2025.

[34] Regina Obe and Leo Hsu. 2021. *PostGIS in Action.* Simon and Schuster.

[35] Matthew O'Connell, Cameron Druyor, Kyle B Thompson, Kevin Jacobson, William K Anderson, Eric J Nielsen, Jan-Reneé Carlson, Michael A Park, William T Jones, Robert Biedron, et al. 2018. Application of the dependency inversion principle to multidisciplinary software development. In *48th Fluid Dynamics Conference (FDC)*. 3856.

[36] Oracle. 2021. JEP 395: Records. Disponível em: <https://openjdk.org/jeps/395>. Acesso em: 5 jun. 2025.

[37] Oracle. 2021. Record Classes. Disponível em: <https://docs.oracle.com/en/java/javase/17/language/records.html>. Acesso em: 12 jun. 2025.

[38] Pivotal Software. 2025. *RabbitMQ — Messaging Broker.* https://www.rabbitmq.com Acesso em: 5 jul. 2025.

[39] Peter C Poole and William M Waite. 1975. *Portability and Adaptability.* Springer Berlin Heidelberg, 183–277.

[40] PostgreSQL Global Development Group. 2025. *PostgreSQL — The World's Most Advanced Open Source Database.* https://www.postgresql.org Acesso em: 5 jul. 2025.

[41] Roger S. Pressman and Bruce R. Maxim. 2014. *Software Engineering: A Practitioner's Approach* (8 ed.). McGraw-Hill Education.

[42] Project Lombok. 2025. *Project Lombok.* https://projectlombok.org/ Acesso em: 5 jul. 2025.

[43] Prometheus Authors. 2025. *Prometheus — Monitoring System.* https://prometheus.io Acesso em: 5 jul. 2025.

[44] Redgate Software. 2025. *Flyway — Database Migrations Made Easy.* https://flywaydb.org/ Acesso em: 5 jul. 2025.

[45] Redis Inc. 2025. *Redis — In-Memory Data Store.* https://redis.io Acesso em: 5 jul. 2025.

[46] Elisabetta Ronchieri and Marco Canaparo. 2023. Assessing the impact of software quality models in healthcare software systems. *Health Systems* 12, 1 (2023), 85–97.

[47] Douglas Morgan Fullin Saldanha, Cleidson Nogueira Dias, and Siegrid Guillaumon. 2022. Transparency and accountability in digital public services: Learning from the Brazilian cases. *Government Information Quarterly* 39, 2 (2022), 101680.

[48] Sass Team. 2025. *Sass — Syntactically Awesome Style Sheets.* https://sass-lang.com/ Acesso em: 5 jul. 2025.

[49] Dhamotharan Seenivasan and Muthukumaran Vaithianathan. 2023. Real-Time Adaptation: Change Data Capture in Modern Computer Architecture. *ESP International Journal of Advancements in Computational Technology (ESP-IJACT)* 1, 2 (2023), 49–61.

[50] Semantic Versioning Initiative. 2025. *Semantic Versioning Specification (SemVer).* https://semver.org Acesso em: 5 jul. 2025.

[51] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.

[52] SonarSource. 2025. *SonarQube — Continuous Code Quality.* https://www.sonarsource.com/products/sonarqube Acesso em: 5 jul. 2025.

[53] Spring. 2025. Auditing. Disponível em: <https://docs.spring.io/spring-data/jpa/reference/auditing.html>. Acesso em: 16 jun. 2025.

[54] Spring Team. 2025. *Spring Boot Project.* https://spring.io/projects/spring-boot Acesso em: 5 jul. 2025.

[55] Spring Team. 2025. *Spring Cloud Gateway.* https://spring.io/projects/spring-cloud-gateway Acesso em: 5 jul. 2025.

[56] Spring Team. 2025. *Spring Data JPA — Auditing.* https://docs.spring.io/spring-data/jpa/reference/auditing.html Acesso em: 5 jul. 2025.

[57] Telegram Messenger LLP. 2025. *Telegram — Messaging App.* https://telegram.org Acesso em: 5 jul. 2025.

[58] Testcontainers Team. 2025. Testcontainers. https://www.testcontainers.org. Acesso em: 10 out. 2025.

[59] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 253–262.

[60] Catalin Tudose, Christian Bauer, and Gavin King. 2023. *Java Persistence with Spring Data and Hibernate.* Simon and Schuster, Chapter 11. Section 11.2: Controlling concurrent access.

[61] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2021. Challenges of resolving merge conflicts: A mining and survey study. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4964–4985.

[62] Ruben Verborgh and Michel Dumontier. 2018. A Web API ecosystem through feature-based reuse. *IEEE Internet Computing* 22, 3 (2018), 29–37.

[63] Craig Walls. 2015. *Spring Boot in Action.* Simon and Schuster.

[64] Chenhao Wei, Lu Xiao, Tingting Yu, Sunny Wong, and Abigail Clune. 2025. How Do Developers Structure Unit Test Cases? An Empirical Analysis of the AAA Pattern in Open Source Projects. *IEEE Transactions on Software Engineering* 51, 4 (2025), 1007–1038.

[65] Jonathan Wetherbee, Massimo Nardone, Chirag Rathod, and Raghu Kodali. 2018. *Entities and the Java Persistence API (JPA).* Apress, Berkeley, CA, 93–155.

[66] Maria A. Wimmer, Efthimios Tambouris, and Panos Panagiotopoulos. 2020. Digital Government: Research and Practice. *ACM Digital Government: Research and Practice* 1, 1 (2020).