UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

VINICIUS FERREIRA NOVACOSKI

REFATORAÇÃO E IMPLEMENTAÇÃO DE NOVAS FUNCIONALIDADES NO E-MUSEU

GUARAPUAVA

2026

VINICIUS FERREIRA NOVACOSKI

REFATORAÇÃO E IMPLEMENTAÇÃO DE NOVAS FUNCIONALIDADES NO E-MUSEU

Refactoring and Implementation of New Features in the E-Museu

Monografia de Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Sistemas para Internet do Curso de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Orientador: Dra. Sediane Carmem Lunardi Hernandes

Coorientador: Dr. Diego Marczal

GUARAPUAVA

2026

VINICIUS FERREIRA NOVACOSKI

REFATORAÇÃO E IMPLEMENTAÇÃO DE NOVAS FUNCIONALIDADES NO E-MUSEU

Monografia de Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Sistemas para Internet do Curso de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 30/06/2026

Sediane Carmem Lunardi Hernandes
Doutora (Orientadora)
Universidade Tecnológica Federal do Paraná, Campus Guarapuava

---

Andres Jessé Porfirio
Doutor (Membro da banca avaliadora)
Universidade Tecnológica Federal do Paraná, Campus Guarapuava

---

Diego Marczal
Doutor (Coorientador e membro da banca avaliadora)
Universidade Tecnológica Federal do Paraná, Campus Guarapuava

---

Marcelo Marcos Vichar Junior
Especialista (Membro da banca avaliadora)
Universidade Tecnológica Federal do Paraná, Campus Guarapuava

GUARAPUAVA

2026

Dedico este trabalho a todas as pessoas que buscam tornar a própria vida e o seu entorno um lugar melhor. Dedico à minha mãe Noeli, à vó Terezinha e à tia Soeli, com gratidão por tudo.

AGRADECIMENTOS

Início agradecendo a Deus pela existência. Agradeço imensamente a minha mãe Noeli da Aparecida Ferreira, minha avó Terezinha da Fonseca Ferreira e minha tia Soeli Terezinha da Fonseca Ferreira por tudo em minha vida, nunca poderei agradecer o suficiente.

Gratidão ao meu amigo Pedro Henrique Chemin Prado, que começou um semestre antes de mim, época em que eu não fazia ideia do que fazer da vida, me motivou a entrar no curso de Sistemas para Internet e por todo o suporte durante o curso, por me ter indicado falar com a professora Sediane para me orientar, quando eu não fazia ideia do que fazer, o que resultou nesse TCC, pelo apoio nos momentos difíceis no centro acadêmico, pela companhia e amizade.

Obrigado Nicolas Justino Veiga, que começou o curso junto comigo, pela ajuda, companhia e amizade, juntamente com sua mãe, Cristiane Justino, que sempre me deu carona, tornando minha formação muito mais acessível.

Obrigado Adnir Luiz de Andrade Junior, por ter nos instigado com a ideia do centro acadêmico quando ninguém nem pensava nisso, e por me motivar a me tornar presidente do CASIS, isso me deu a oportunidade de viver coisas que nunca imaginei, ampliando minha evolução em quase todas as áreas da minha vida.

Agradeço a todos os colegas membros da diretoria fundadora do CASIS, pelo apoio e companhia, principalmente ao William Wendling Veiga, Douglas Hendrick, Agatta Antonyelle, Flavio Borges de Lima, Pedro Henrique Chemin Prado e Maria Eduarda Guedes. Foi uma experiência impressionante, longe de ser fácil, porém absolutamente enriquecedora para mim e espero que também tenha sido enriquecedora para todos que foram impactados pelo centro acadêmico.

Muito obrigado ao Soon Sam Santos, sendo a primeira pessoa a confiar no meu trabalho e a me dar a primeira oportunidade de pôr em prática o que aprendi durante esses anos, muito obrigado por todo o apoio e companhia, é muito bom trabalhar com você. E também agradeço por ter patrocinado a viagem em que organizei para o evento Codecon Summit 26.

Eloi Mamcasz, agradeço pela gentileza com que sempre me tratou e muito obrigado por todo o apoio ao centro acadêmico, patrocinando duas viagens, sendo elas possíveis graças a você, também pela ajuda na nossa recepção aos calouros, e pelas oportunidades que chegaram até a mim e colegas pelo seu incentivo.

Minha gratidão à UTFPR Guarapuava como instituição e aos professores pelo apoio, na ajuda em fazer os eventos do centro acadêmico serem possíveis e todo ensino, agradeço especialmente ao professor Luciano Ogiboski por ter me ajudado nos momentos difíceis e sempre se mostrar interessado na evolução do curso. Também agradeço à professora Sediane Carmem Lunardi Hernandes, ao professor Diego Marczal pelo apoio e orientação durante o curso e também ao professor Roni Fabio Banaszewski pelo apoio.

Menciono também algumas pessoas, que ainda não foram mencionadas, que passaram pela minha vida acadêmica e guardo com carinho, sendo Mateus Sydor, Alessandro Dias, Va-

nessa Meira, Eder Murilo, Eduardo Kowaski e Luiz Felipe Ramos, prof Eleandro, prof Hermano, Marcio Koziel, Beatriz Amante, Milena Hamerski, Fernando Gonçalves, Jonas Venzel, João Gabriel Camilo, Ana Clara Novak, Gabriel Kaminski, Geovane Fedrecheski, Maurício Barfknecht, Duda Righi, Leandro Larson, Gabriel Kuiawa, Emanuel Oliveira, Miguel Geovanni, Augusto Netto, Thiago Sampaio e Níckolas Gomes, Lucas Fajardo, Guilherme Patricio, Maria Porcina, Alana Maiumy, João Victor Silva e João Arthur. E também agradecer a minha banca avaliadora de TCC, prof Andres Jessé Porfirio e prof Marcelo Vichar.

E agradeço a todo mundo que passou pelo meu caminho durante esses anos, realmente a todo mundo, mesmo que seu nome não tenha sido mencionado aqui, pois são muitas pessoas, eu lembro de você, de cada um de vocês que contribuiu de alguma forma para minha evolução e espero que igualmente eu tenha contribuido. Obrigado.

E finalizando, gostaria de dizer que temos muito mais poder do que imaginamos, nossa capacidade de transformar nossa realidade e ao nosso entorno é real, nós somos capazes de fazer a diferença, por isso não subestime seu poder, pois quem mais pode parar seu impacto é você mesmo. Com isso, não desistam daquilo que vale a pena ser buscado.

Obrigado.

RESUMO

Este trabalho apresenta a refatoração e a implementação de novas funcionalidades no E-Museu, sistema web da Universidade Tecnológica Federal do Paraná (UTFPR) para catalogação e exposição de itens da história da informática em formato de museu digital. O projeto foi motivado pelas limitações da versão anterior da aplicação, que dificultava a manutenção, carecia de padronização e de testes automatizados e não atendia demandas recentes de gestão do acervo e de implantação em ambientes institucionais. O desenvolvimento combinou levantamento e priorização de requisitos (MoSCoW), fluxo Gitflow no GitHub, ambientes Docker para desenvolvimento local, implantação com Coolify em homologação e produção, atualização tecnológica da aplicação (PHP 8.5 e Laravel 13), integração contínua e refatoração do código com ferramentas de qualidade e testes automatizados. Entre os resultados, o sistema passou a oferecer internacionalização de conteúdos, proteção contra abuso em formulários, armazenamento de mídia em objetos (MinIO), cadastro com múltiplas imagens por item e suporte à geração de etiquetas para peças físicas, permanecendo publicado nos domínios institucionais. Com isso, o trabalho consolidou uma base mais organizada, rastreável e sustentável para a continuidade do E-Museu como iniciativa de extensão e preservação da memória tecnológica.

Palavras-chave: museu; refatoração; funcionalidades; engenharia de software; sistema web.

ABSTRACT

This monograph presents the refactoring and implementation of new features in the E-Museu, a web system of the Federal University of Technology (UTFPR) for cataloging and exhibiting items from the history of computing in a digital museum format. The project was motivated by the limitations of the previous version of the application, which hindered maintenance, lacked code standardization and automated tests, and did not meet recent demands for collection management and deployment in institutional environments. Development combined requirements gathering and prioritization (MoSCoW), a Gitflow workflow on GitHub, Docker environments for local development, deployment with Coolify in staging and production, technological updates to the application (PHP 8.5 and Laravel 13), continuous integration, and code refactoring with quality tools and automated tests. Among the results, the system now offers content internationalization, abuse protection on forms, object storage for media (MinIO), registration with multiple images per item, and support for generating labels for physical items, while remaining published on institutional domains. Thus, the work consolidated a more organized, traceable, and sustainable foundation for the continuity of the E-Museu as an extension initiative and for the preservation of technological memory.

Keywords: museum; refactoring; functionalities; software engineering; web development.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

No campo da computação, a evolução do hardware acompanha décadas de avanço: de válvulas a vácuo a circuitos de escala muito reduzida, até os computadores atuais (MCCARTNEY, 1999). Essa trajetória levou à Internet e a novas áreas de estudo, com busca recorrente por mais eficiência, desempenho e confiabilidade nos sistemas. No desenvolvimento de software, vale o mesmo princípio: muitos sistemas precisam de atualização contínua, refatoração do código e novas funcionalidades para não se tornarem obsoletos. O desenvolvimento de software ocupa lugar central nesse cenário, por sustentar boa parte das soluções digitais usadas na educação, na cultura e na economia.

Nesse contexto de constante evolução tecnológica e de aprimoramento de sistemas, iniciativas educacionais e culturais que envolvem tecnologia digital demandam soluções adaptáveis e capazes de integrar novas funcionalidades de forma contínua. Reconhecendo essa necessidade de adaptação e aprimoramento, as universidades passaram a buscar maneiras de organizar e preservar informações de forma mais estruturada. A Universidade Tecnológica Federal do Paraná (UTFPR) e a Universidade Estadual do Centro-Oeste (UNICENTRO), a partir dos projetos de extensão "Tecno-Lixo: Oficina do Aprender"(RIBEIRO *et al.*, 2024) e "E-Lixo"(Ré; THOMEN, 2021), respectivamente, que trabalham com o reaproveitamento de peças de computadores descartadas, acabaram armazenando diversos itens de informática em suas dependências para a criação de um museu, preservando, assim, um pouco da história da computação. Gonzaga (2024) implementou um museu virtual de informática para o cadastro e a gerência desses itens, chamado E-Museu.

O E-Museu trata-se de um sistema web desenvolvido como espaço de catalogação, gerenciamento e exposição de itens tecnológicos no formato de museu digital. Esse sistema está em uso e encontra-se disponível na Web em https://e-museu.gp.utfpr.edu.br. Contudo, foi necessária uma refatoração e aprimoramento, além da adição de novas funcionalidades, motivando o presente trabalho. Além disso, esse tipo de evolução enfrenta obstáculos, como atualizar tecnologias sem atrapalhar o que já funciona, refatorar sem mudar o comportamento visível do website ao usuário e publicar o sistema em produção com confiabilidade. Assim, este trabalho refatorou e ampliou o E-Museu com padrões de desenvolvimento de software, testes automatizados e ambientes padronizados, buscando tornar o sistema mais robusto, seguro e de fácil manutenção. Dessa forma, o E-Museu ilustra essa lógica de evolução contínua e mostra como melhorias técnicas podem ampliar a experiência de quem consulta o acervo e de quem o administra.

1.1 Objetivos

1.1.1 Objetivo geral

Refatorar e implementar novas funcionalidades no E-Museu, aplicando boas práticas de programação, padrões de projeto de software e testes automatizados, com o intuito de melhorar a manutenibilidade e a qualidade do código.

1.1.2 Objetivos Específicos

- Estabelecer uma estrutura de versionamento e desenvolvimento de código com padrões bem definidos, utilizando GitHub, de modo a garantir organização, rastreabilidade e colaboração eficiente.
- Configurar ambientes estáveis de desenvolvimento, teste e produção, assegurando consistência, reprodutibilidade e confiabilidade do sistema.
- Refatorar o código existente, aprimorando legibilidade, modularidade, manutenibilidade e desempenho.
- Implementar testes automatizados a fim de assegurar a qualidade, confiabilidade e robustez do código, contemplando testes de fluxo e de integração.
- Desenvolver e aprimorar funcionalidades do E-Museu, incluindo múltiplas imagens por item, a internacionalização do sistema, a proteção de formulários (por exemplo, Turnstile) e a geração de etiquetas com código e nome para as peças físicas.
- Aplicar padrões de projeto e boas práticas de desenvolvimento, orientando tanto a refatoração quanto a implementação de novas funcionalidades de forma estruturada e eficiente.

1.2 Justificativa

Na versão analisada no início deste trabalho, o E-Museu não separava de forma clara os ambientes de desenvolvimento e produção na operação diária. Cada execução da aplicação exigia configurações manuais (instalação de dependências *PHP*, preparação do banco de dados e do servidor web). O código seguia uma versão simplificada do padrão *Model, View, Controller* (MVC), com lógica concentrada em dezoito controladores, sem divisão consistente entre *services*, *actions* e controladores, o que dificultava a manutenção. Faltavam também mecanismos automatizados de qualidade e de validação do comportamento da aplicação, como

linters e testes automatizados. As funcionalidades existentes precisavam de evolução, o que reforçou a necessidade de refatoração e de novos recursos (IEEE Computer Society, 2025).

Nesse contexto, a refatoração assume papel central, pois melhora a estrutura interna do código sem alterar o comportamento visível ao usuário, trazendo mais clareza, modularidade e facilidade de manutenção. Ao reduzir a complexidade e as duplicações, a refatoração facilita correções e novas funcionalidades com menor risco de regressão (FOWLER, 2004).

2 REFERENCIAL TEÓRICO

Este capítulo apresenta os conceitos de refatoração, arquitetura e ferramentas adotados no E-Museu.

2.1 O E-Museu

O E-Museu é uma iniciativa digital dedicada à preservação da história da computação, voltada à consulta pública e à gestão do acervo de informática. Seu propósito é registrar e disponibilizar informações sobre a evolução da informática, dos equipamentos clássicos aos marcos do desenvolvimento de hardware, de modo que o conhecimento histórico permaneça acessível mesmo quando os artefatos físicos se deterioram.

O sistema web do E-Museu oferece um catálogo público com fichas, imagens e textos, além de um painel administrativo para que curadores registrem, validem e organizem as peças. Entre os conteúdos, reúne dados técnicos sobre processadores, placas-mãe, memórias, unidades de armazenamento e periféricos, com descrição e histórico de uso. O E-Museu adota um modelo colaborativo: a comunidade pode contribuir com peças, documentos e relatos, que são submetidos à verificação antes de serem integrados à coleção. Voltado para fins educacionais e culturais, operando como uma iniciativa sem fins lucrativos com foco na divulgação da memória tecnológica.

2.2 Refatoração de software

De acordo com Fowler (2004), refatorar significa modificar a organização do software de forma sistemática, visando eliminar a complexidade desnecessária, reduzir duplicações, aprimorar o design e facilitar a compreensão do código. Refatoração não é uma etapa isolada, mas um processo contínuo que acompanha o desenvolvimento, permitindo melhorias incrementais à medida que o sistema cresce.

Além disso, práticas de refatoração auxiliam na prevenção da chamada "dívida técnica", evitando que o código se torne difícil de compreender, testar ou modificar. Essa preocupação é especialmente relevante em sistemas acadêmicos e colaborativos, nos quais novas funcionalidades podem surgir a partir de pesquisas, contribuições externas ou mudanças institucionais.

Fowler (2004) também enfatiza o papel dos testes automatizados no processo de refatoração, de modo que o comportamento do software permaneça correto após cada melhoria estrutural. Dessa forma, refatorar não apenas aprimora o design do código, mas fortalece a confiabilidade do sistema.

Desta forma, a refatoração é um processo fundamental no ciclo de desenvolvimento de sistemas, pois busca melhorar a estrutura interna do código sem alterar seu comportamento

externo. No E-Museu, essa prática torna-se importante para garantir qualidade, legibilidade, extensibilidade e facilidade de manutenção, especialmente diante da evolução constante dos requisitos técnicos e funcionais.

No contexto deste trabalho, a refatoração contempla a reorganização e a melhoria estrutural dos controladores da versão legada do E-Museu, visando padronizar responsabilidades, reduzir acoplamentos indevidos e tornar a lógica de controle mais clara, testável e alinhada às boas práticas de desenvolvimento de software.

2.3 Boas práticas de desenvolvimento

Esta seção apresenta as principais boas práticas consideradas no desenvolvimento do E-Museu e como cada abordagem contribui para a organização, a qualidade e a manutenção do sistema.

2.3.1 *Model, View, Controller* (MVC)

O padrão *Model, View, Controller* (MVC) estabelece a separação do software em três camadas principais: *Model*, *View* e *Controller* (DEACON, 2009). Essa divisão organiza a aplicação de modo que a lógica de negócios, a interface e o controle de requisições não se misturem, reduzindo o acoplamento e facilitando a manutenção. No contexto do E-Museu, o MVC estrutura o fluxo de exibição do acervo, a manipulação dos dados e o controle das requisições, tornando o sistema mais claro e modular para quem der continuidade ao código.

2.3.2 *Clean Code*

O princípio de *Clean Code* orienta a escrita de código simples, legível, coeso e de fácil entendimento, priorizando a qualidade sobre a complexidade desnecessária (MARTIN, 2009). Sua aplicação incentiva boas práticas, como nomes descritivos, funções curtas, ausência de código duplicado, clareza semântica e utilização consistente de padrões arquiteturais. Para o E-Museu, adotar *Clean Code* contribui para a evolução do sistema, reduz custos de manutenção, facilita correções e permite que novos desenvolvedores entendam o projeto com menos esforço.

2.3.3 *Domain-Driven Design* (DDD)

O *Domain-Driven Design* (DDD) propõe que o desenvolvimento de software seja guiado pelo domínio do negócio, priorizando o entendimento profundo do problema antes da solução tecnológica (EVANS, 2004). Entre seus conceitos centrais destacam-se a *linguagem ubíqua* (vocabulário compartilhado entre desenvolvedores e especialistas do museu), os *contextos de-*

limitados (fronteiras em que um modelo vale de forma coerente) e a organização do código em camadas que refletem regras de negócio e não apenas detalhes de infraestrutura.

No E-Museu, adotou-se uma aplicação formal dessa abordagem: o repositório foi reorganizado em pastas e *namespaces* PHP alinhados a contextos de negócio (por exemplo, catálogo do acervo, taxonomia de etiquetas, colaboradores externos e identidade administrativa), com serviços, ações e requisições HTTP agrupados por domínio. Não se buscou implementar todos os padrões táticos do DDD (como agregados explícitos ou eventos de domínio em toda a base), mas sim dar nome e delimitar as partes do sistema que já existiam no modelo de dados e nos fluxos de curadoria e de contribuição pública. A lista dos domínios adotados e a correspondência com modelos, controladores e serviços são detalhadas na Seção 4.4.1.

3 MATERIAIS E MÉTODOS

Neste capítulo são descritos os materiais e os métodos empregados no planejamento e no desenvolvimento do E-Museu. O conteúdo está estruturado em duas partes: a primeira apresenta as tecnologias e ferramentas utilizadas; a segunda descreve os procedimentos metodológicos. A execução detalhada dessas etapas é documentada no Capítulo 4.

3.1 Materiais

Na presente seção são apresentadas as principais tecnologias empregadas na refatoração e na implementação de novas funcionalidades do E-Museu, organizadas conforme a camada em que atuam no sistema (servidor, dados, infraestrutura, interface, internacionalização, comunicação por *e-mail*, segurança, qualidade de software, ambiente de edição de código, APIs de modelos de linguagem e registro do trabalho).

3.1.1 *PHP*

Hypertext Preprocessor (*PHP*) é uma linguagem de programação desenvolvida e mantida por uma ampla comunidade de desenvolvedores *open source*, amplamente utilizada no desenvolvimento de aplicações, especialmente no lado do servidor, sendo capaz de gerar conteúdo dinâmico para a *World Wide Web* (The PHP Group, 2025). No E-Museu, o *PHP* permanece como linguagem base do *back-end*. A implementação anterior do sistema, herdada da primeira versão do projeto, empregava a série 8.1. No início deste trabalho, o interpretador foi atualizado para a série 8.3 nas imagens Docker; posteriormente, em consonância com as atualizações do *framework* descritas na subseção seguinte, adotou-se a série 8.5, que constitui a versão atual do repositório. As bibliotecas do *back-end* são instaladas e versionadas pelo Composer (Composer Contributors, 2026), gestor de dependências padrão do ecossistema PHP, que declara no repositório o *framework* Laravel e os pacotes usados na refatoração.

3.1.2 *Laravel*

O *Laravel* é um *framework* desenvolvido em *PHP* que facilita a construção de aplicações web, oferecendo recursos para roteamento, autenticação, persistência de dados, filas, testes e organização do código (Laravel Team, 2025). A versão anterior do E-Museu utilizava o Laravel 10. No início deste trabalho, o projeto foi integralmente atualizado para o Laravel 12, o que exigiu revisão de dependências, de estrutura de diretórios e de convenções do *framework*. Posteriormente, no meio do desenvolvimento, foi lançado o Laravel 13; diante da necessidade de manter o sistema alinhado a versões suportadas e seguras, realizou-se uma segunda atuali-

zação para essa série, com novo ciclo de ajustes e de validação por testes automatizados. A versão atual do repositório reflete, portanto, o Laravel 13 como linha de base do *back-end*.

Entre os componentes do ecossistema Laravel empregados no projeto destacam-se: o *Eloquent ORM*, para mapeamento objeto-relacional e acesso ao banco de dados; o motor de *templates* Blade, para a camada de apresentação (Seção 3.1.9); o pacote *Laravel UI*, para fluxos de autenticação e *scaffolding* inicial; e o *Laravel Sanctum*, para autenticação de API e de sessões em aplicações de página única quando necessário. Para geração de códigos bidimensionais associados à catalogação física das peças, utiliza-se a biblioteca *endroid/qr-code*, integrada à camada de aplicação.

3.1.3 *Docker e Docker Compose*

O *Docker* é uma plataforma que permite criar e executar aplicações em ambientes independentes, garantindo isolamento, portabilidade e consistência entre diferentes ambientes de desenvolvimento e produção (Docker, Inc., 2025b). O *Docker Compose* complementa o *Docker*, permitindo coordenar múltiplos serviços por meio de arquivos declarativos (Docker, Inc., 2025b; Docker, Inc., 2025a). No E-Museu, essas ferramentas estruturam o ambiente local (aplicação *PHP*, *Nginx*, *MySQL*, *Redis*, serviço *Node* para o *Vite* e *phpMyAdmin*), as imagens multiestágio de *staging* e produção (`app` e `web` com volume compartilhado em `storage/`) e, quando necessário, o serviço *MinIO* declarado junto à aplicação. O *phpMyAdmin* (The phpMyAdmin Team, 2026) integra apenas o Compose local como interface web para consulta e manutenção auxiliar do banco de dados durante o desenvolvimento, sem participar de *staging* ou produção. Um script `run` encapsula comandos frequentes (`setup`, testes, análise estática), reduzindo erros operacionais na rotina de desenvolvimento.

3.1.4 *Coolify*

O *CapRover* é uma plataforma de *deploy* e administração de aplicações em servidores próprios, com automação por meio de containers e de *webhooks* (CapRover Team, 2025). Durante o planejamento da infraestrutura do E-Museu, o *CapRover* foi avaliado como alternativa para hospedar o sistema na UTFPR, por oferecer funcionalidades semelhantes às de outras soluções de *Platform as a Service* (*PaaS*) auto-hospedadas.

Optou-se, contudo, pelo *Coolify*, plataforma de *deploy* e gerenciamento de aplicações com suporte a containers Docker, variáveis de ambiente e *deploy* automatizado por meio de *webhooks* (CoolLabs, 2025). A escolha deveu-se à melhor adequação ao fluxo do projeto, em especial pelo suporte ao Docker Compose para declarar e reproduzir a composição dos serviços e pela automação de implantação disparada por *webhooks* a cada mudança nas ramificações

`main` e `develop`, enquanto o *CapRover* exigiria maior configuração manual de *Dockerfiles* e de *webhooks*.

No E-Museu, o *Coolify* na versão `v4.0.0-beta.470` organiza os serviços em Docker e dispara o *deploy* por *webhooks*. Antes da implantação na UTFPR, o mesmo fluxo foi testado em uma VPS na DigitalOcean, simulando produção, para validar *webhooks*, variáveis de ambiente e os arquivos `docker-compose.staging.yml` e `docker-compose.production.yml` sem ir ao campus. Na UTFPR, o *Coolify* ficou em https://coolify.e-museu.gp.utfpr.edu.br/, com a ramificação `develop` ligada ao *staging* e a `main` à produção. A evolução dessa automação está na Seção 4.8. Em ambos os casos, os serviços publicados incluem a aplicação (*back-end* em *PHP* 8.5 e Laravel 13, conforme acima), *MySQL* (`mysql:8.0-debian`), *Redis* (`redis:7.2`) e *MinIO* (`minio/minio:RELEASE.2025-09-07T16-13-09Z`); o *MinIO* foi declarado em um projeto Docker Compose vazio no *Coolify* quando saiu do catálogo de serviços da plataforma (detalhes na subseção sobre *MinIO*).

3.1.5 *MySQL*

O *MySQL* é um sistema de gerenciamento de banco de dados relacional, amplamente utilizado no desenvolvimento de aplicações web, que permite o armazenamento, consulta e manipulação de dados de forma eficiente e segura (Oracle Corporation, 2025). No E-Museu, o *MySQL* foi mantido desde a primeira versão do sistema para persistir entidades como itens do acervo, traduções, categorias, colaboradores e localizações. No ambiente local, o serviço integra o Docker Compose (Seção 3.1.3) com imagem `mysql:8.0-debian`; em *staging* e produção, a mesma imagem é provisionada e administrada pelo *Coolify* (Seção 3.1.4).

3.1.6 *Redis*

O *Redis* é um armazenamento de estruturas de dados em memória RAM, utilizado como *cache*, fila ou repositório de sessões em aplicações web de médio e grande porte (Redis Ltd., 2026). No E-Museu, o *Redis* permite que o Laravel utilize *cache* e sessões sem depender apenas do sistema de arquivos ou do banco relacional para esses fins. Nos ambientes de *staging* e de produção, hospedados na UTFPR, o *Coolify* (Seção 3.1.4) provisiona a imagem `redis:7.2`, de modo que a aplicação mantenha o mesmo papel para *cache* e sessões nos domínios publicados do sistema.

3.1.7 *Nginx*

O *Nginx* é um servidor web e *proxy* reverso de alto desempenho, frequentemente empregado para servir arquivos estáticos, encaminhar requisições ao interpretador *PHP* e expor a aplicação na porta HTTP (F5, Inc., 2026). No E-Museu, o tráfego público chega ao container `web`, no qual o *Nginx* na versão 1.29.3 (imagem Docker `nginx:mainline-alpine3.22-perl`) encaminha as requisições dinâmicas ao container da aplicação Laravel, em conformidade com a arquitetura de containers distintos para servidor web e aplicação descrita na Seção 3.1.3.

3.1.8 *MinIO* e armazenamento compatível com S3

O *MinIO* é um servidor de armazenamento de objetos compatível com a API S3 da Amazon Web Services, adequado a arquivos de mídia e a *uploads* em larga escala (MinIO, Inc., 2026). No E-Museu, o disco `public` do Laravel foi configurado para utilizar o driver S3 por meio do pacote *league/flysystem-aws-s3-v3*, de modo que fotos e demais arquivos do acervo possam ser persistidos em armazenamento de objetos nos ambientes de *staging* e de produção.

Durante o desenvolvimento, o *MinIO* deixou de contar com suporte adequado entre os serviços oferecidos pelo *Coolify* na configuração então disponível na UTFPR (Seção 3.1.4), e não foi identificada alternativa integrada à plataforma que reunisse, de forma equivalente, compatibilidade com a API S3, controle institucional sobre os dados e facilidade de operação já prevista para o projeto. Diante desse cenário, foi necessário aprimorar a documentação de implantação e passar a provisionar o *MinIO* em projeto Docker Compose no *Coolify*, com imagem `minio/minio:RELEASE.2025-09-07T16-13-09Z`, em *staging* e em produção. Em desenvolvimento local, o armazenamento pode permanecer em disco local, preservando simplicidade na máquina do desenvolvedor sem alterar a abstração de arquivos do *framework*.

3.1.9 HTML e Blade

O HTML (*HyperText Markup Language*) é a linguagem de marcação que estrutura o conteúdo exibido no navegador: títulos, formulários, listas, links e demais elementos semânticos da página (World Wide Web Consortium (W3C), 2026b). No E-Museu, o HTML não é editado como arquivo estático solto na raiz do servidor; ele é gerado no servidor a cada requisição, a partir de *views* Laravel que o *framework* renderiza e envia ao cliente junto com os *assets* de estilo e de script citados nas subseções seguintes.

O Blade é o motor de *templates* do Laravel (Laravel Team, 2025): arquivos com extensão `.blade.php` em `resources/views/`, com diretivas como `@extends`, `@section` e `@include` para montar *layouts*, repetir trechos comuns (cabeçalho, rodapé, menus) e injetar

dados vindos dos controladores sem misturar lógica de negócio com marcação. No repositório, as *views* se dividem entre a interface pública do museu virtual e o painel administrativo de curadoria, espelhando os fluxos de navegação do sistema. A versão anterior do projeto já se apoiava nesse modelo; neste trabalho, as telas foram mantidas e ajustadas à medida que o *front-end* passou a Vite e SCSS (Seções 3.1.10 e 3.1.11).

3.1.10 CSS e SCSS

O CSS (*Cascading Style Sheets*) é a linguagem padrão da web para definir a apresentação visual das páginas estruturadas em HTML (Seção 3.1.9): cores, tipografia, espaçamento, *layout* e adaptação a diferentes tamanhos de tela (World Wide Web Consortium (W3C), 2026a). Na versão anterior do E-Museu, os estilos já se apoiavam em CSS servido junto às *views*; permanecem válidos os seletores e as regras de apresentação, mas a organização dos arquivos foi revista neste trabalho.

O SCSS (*Sassy CSS*) é uma sintaxe do pré-processador Sass que acrescenta variáveis, aninhamento de seletores e importação modular em relação ao CSS puro (Sass Contributors, 2026). No repositório atual, as folhas de estilo ficam em `resources/sass/`, com entradas separadas para o site público e para o painel administrativo (por exemplo `app.scss` e `admin.scss`), o que permite reutilizar tokens de cor e de espaçamento alinhados ao *Bootstrap* 5 (subseção seguinte) sem repetir blocos longos de declarações. O Sass é compilado para CSS no momento do *build* dos *assets*, pelo *Vite* descrito abaixo; em desenvolvimento local, a alteração de um arquivo SCSS pode ser refletida na interface com recarregamento rápido quando o servidor do Vite está ativo.

3.1.11 *Vite* e tecnologias da interface web

O *Vite* é uma ferramenta de compilação e servidor de desenvolvimento para *front-end*, que integra-se ao Laravel por meio do plugin oficial *laravel-vite-plugin*, permitindo empacotar JavaScript e compilar as folhas SCSS da Seção 3.1.10 com recarregamento rápido durante o desenvolvimento (Vite Contributors, 2026; OpenJS Foundation, 2026). O *runtime* JavaScript é o *Node.js*, executado no container dedicado do Docker Compose (Seção 3.1.3) com a imagem `node:25.4.0-alpine3.22`, alinhada à versão empregada na compilação dos *assets* e na pipeline de integração contínua. No E-Museu, o *front-end* combina ainda *Bootstrap* 5 para componentes e *layout* responsivo (Bootstrap Team, 2026), *jQuery* para interações legadas compatíveis com a base existente, *Axios* para requisições HTTP assíncronas e *Bootstrap Icons* para iconografia. O suporte a múltiplos idiomas na interface do cliente emprega o *i18next*, detalhado na subseção seguinte.

3.1.12 Internacionalização

A internacionalização do E-Museu articula servidor e cliente. No Laravel, os textos do acervo (itens, categorias, etiquetas e *extras*) passaram a ser armazenados em tabelas de tradução associadas a cada entidade e a um registro de idioma, em substituição a colunas monolíngues da versão anterior; mensagens de interface, validação e notificações utilizam arquivos de idioma em português do Brasil e em inglês, com apoio do pacote Laravel Lang (Laravel Lang Contributors, 2026). Na camada do cliente, o *i18next* traduz textos dinâmicos em JavaScript e permite alternar o idioma exibido sem recarregar toda a página quando o fluxo da interface exige isso. As duas partes atuam em conjunto: o servidor persiste e valida conteúdos por idioma; o cliente exibe rótulos e interações coerentes com a escolha do usuário.

3.1.13 Envio de *e-mail* (Laravel Mail e SMTP)

O componente Laravel Mail (Laravel Team, 2025) concentra o envio de mensagens transacionais da aplicação. No E-Museu, ele é empregado no fluxo de verificação de *e-mail* de colaboradores do catálogo público. Para *staging* e produção, a UTFPR disponibilizou caixa institucional dedicada ao projeto, `e-museu-gp@utfpr.edu.br`, utilizada como remetente das mensagens e autenticada no servidor SMTP da universidade (`smtp.utfpr.edu.br`). No ambiente local e na suíte de testes automatizados utilizam-se transportes que não disparam envio real, de modo a preservar a reprodutibilidade dos testes sem depender de caixas postais externas.

3.1.14 Cloudflare Turnstile e proteção *antibot*

O Turnstile é um mecanismo da Cloudflare para verificar interação humana em formulários web, alternativa a *CAPTCHAs* tradicionais (Cloudflare, Inc., 2026). Nos objetivos do trabalho constava a incorporação de proteção no estilo reCAPTCHA; no E-Museu essa função ficou a cargo do Turnstile, com validação no *back-end* Laravel antes de aceitar o envio. O desafio aparece no *login* administrativo, no pedido de código de verificação por e-mail no catálogo de colaboradores e em cadastros de itens em que a proteção é necessária; nos testes automatizados o mecanismo pode permanecer desligado para não depender do serviço externo. Em *staging* e produção o Turnstile está ativo nos fluxos citados.

3.1.15 *Git*, *GitHub* e *GitHub Actions*

O *Git* é a ferramenta de controle de versões utilizada para registrar alterações, organizar ramificações e compartilhar o código-fonte (Git SCM, 2025). O *GitHub* concentra o repositó-

rio do projeto, as *issues*, os *pull requests*, os *templates* de contribuição e a documentação de fluxo de trabalho (GITHUB, 2025). Complementarmente, o *GitHub Actions* automatiza a integração contínua: em cada *pull request* direcionada às ramificações `main` ou `develop`, são executados trabalhos de testes (*PHPUnit* no ambiente Docker descrito na Seção 3.1.3), de verificação de qualidade de código (*PHP CodeSniffer*, *Laravel Pint*, *PHPMD*, *PHPStan*, *phpcpd*, *ESLint* e *Prettier*) e de validação de implantação em ambiente de produção simulado (GitHub, Inc., 2026a).

3.1.16 Ferramentas de análise estática e de testes

Para assegurar qualidade e manutenibilidade do código refatorado, o projeto adota um conjunto de ferramentas executado localmente e na pipeline de *Continuous Integration/Continuous Delivery* (CI/CD), coordenado pelo script `run` e reproduzido no *GitHub Actions* (subseção anterior). No que se refere a testes automatizados, utiliza-se o *PHPUnit* para validar testes unitários e de funcionalidade (*feature*) da aplicação Laravel (Sebastian Bergmann, 2026). No *back-end*, complementam a revisão manual as ferramentas de análise estática e de qualidade: o *PHPStan* com extensão *Larastan* para verificação tipada (PHPStan Contributors, 2026); o *PHP CodeSniffer* (*PHPCS*), com o corretor *PHPCBF* para ajustes automáticos de estilo quando aplicável, para convenções de codificação; o *Laravel Pint* para padronização alinhada ao ecossistema Laravel (Laravel Team, 2026); o *PHP Mess Detector* (*PHPMD*) para complexidade e possíveis problemas de desenho (PHPMD Contributors, 2026); o *PHP Insights* para métricas agregadas de qualidade; o *PHP Copy/Paste Detector* (*phpcpd*) para detecção de duplicação; e o *Churn PHP* para identificar arquivos candidatos a refatoração com base em frequência de alteração e complexidade. No *front-end*, empregam-se *ESLint* e *Prettier* para *lint* e formatação de JavaScript, SCSS e JSON. Na integração contínua, cada *pull request* dispara o *PHPUnit* e as verificações de *PHPCS*, *Laravel Pint*, *PHPMD*, *PHPStan*, *phpcpd*, *ESLint* e *Prettier*; *PHP Insights*, *Churn PHP* e *PHPCBF* permanecem disponíveis no ambiente local do desenvolvedor. Essas ferramentas documentam, de forma reprodutível, os critérios de aceitação do código no E-Museu.

3.1.17 *Cursor*

O *Cursor* é um ambiente de desenvolvimento integrado com *Git*, depuração e assistentes de inteligência artificial baseados em modelos de linguagem (Anysphere, Inc., 2026). No E-Museu, o *Cursor* foi a única *IDE* utilizada, de forma intensiva e ao longo de quase todo o projeto: a ampla refatoração da base legada, as atualizações de *PHP* e Laravel, as novas funcionalidades priorizadas (Capítulo 4) e os ajustes de infraestrutura implicaram alterações em

grande parte do repositório e em dezenas de *pull requests*, volume de trabalho difícil de sustentar no mesmo prazo sem esse apoio.

3.1.18 Groq, OpenRouter e GitHub Models

O painel administrativo do E-Museu consome APIs de *chat completion* compatíveis com o formato *OpenAI* para tradução assistida de conteúdos multilíngues (itens, categorias, etiquetas e campos associados). Foram integrados três provedores, o Groq (Groq, Inc., 2026), o OpenRouter (OpenRouter, Inc., 2026) e o GitHub Models (GitHub, Inc., 2026b), priorizando ofertas gratuitas ou com cotas sem custo para o projeto: quando um serviço atinge o limite de uso ou deixa de responder, a aplicação tenta o seguinte na ordem definida (*failover*). Há limite de requisições por administrador e, no modo automático, percorre-se essa cadeia até obter resposta válida; o administrador pode, opcionalmente, escolher um provedor específico na interface.

3.1.19 *Notion*

O *Notion* é uma ferramenta de anotações e de organização em páginas, usada para listas e registro de atividades (Notion Labs, Inc., 2026). No E-Museu, manteve-se uma página dedicada ao projeto e, ao longo do desenvolvimento, anotou-se diariamente o que já tinha sido feito, o que ainda precisava ser feito e o tempo gasto em cada dia; com isso foi possível registrar o tempo total e o período de trabalho indicados na Subseção 5.4.2.

3.2 Métodos

Nesta seção são apresentados, em ordem cronológica, os procedimentos adotados nas etapas de elaboração e de execução deste Trabalho de Conclusão de Curso. A aplicação concreta de cada etapa é relatada no Capítulo 4.

3.2.1 Levantamento de requisitos

O levantamento de requisitos define o que se pretende entregar em um projeto e exige atenção, pois imprecisões comprometem a qualidade do produto final. No E-Museu, a etapa foi essencial e contou com discussões com professores da orientação para alinhar necessidades reais às funcionalidades e refatorações implementadas neste trabalho.

3.2.2 Priorização *MoSCoW*

Must-have, *Should-have*, *Could-have* e *Won't-have* (*MoSCoW*) é uma técnica de priorização de requisitos em projetos, amplamente utilizada no contexto do desenvolvimento de software, que consiste na divisão dos requisitos em quatro ou cinco categorias distintas, dependendo da preferência (Agile Business Consortium, 2025):

- *Must-have* (deve ter): requisitos essenciais e obrigatórios para o funcionamento mínimo do sistema. Sem eles, o projeto é considerado inviável;
- *Should-have* (deveria ter): requisitos importantes, mas não críticos. Sua ausência pode ser contornada temporariamente, sem comprometer o funcionamento básico;
- *Could-have* (poderia ter): requisitos desejáveis que agregam valor; porém, a implementação pode ser adiada ou descartada se houver limitação de tempo ou recursos;
- *Won't-have* (não terá): requisitos que foram deliberadamente deixados de fora do escopo atual, mas que podem ser considerados em versões futuras do projeto;
- *Wish* (desejar): categoria adicional usada em algumas variações da técnica para representar ideias ou funcionalidades que não fazem parte do planejamento atual, mas refletem desejos ou possibilidades futuras.

Essa técnica ajuda a organizar o escopo e as expectativas de entrega, concentrando o esforço nos aspectos mais críticos. No projeto, o MoSCoW foi aplicado após o levantamento inicial para definir quais requisitos tinham prioridade.

3.2.3 Criar padrão de *Gitflow*

O *Gitflow* é um modelo de ramificação (*branching model*) que define uma estrutura organizada para o uso do *Git* durante o desenvolvimento de um projeto (Git Documentation Team, 2025; GitHub, Inc., 2025). Ele propõe um fluxo de trabalho padronizado, com regras claras sobre como e quando criar, mesclar e remover *branches*, facilitando o controle de versões, a colaboração entre desenvolvedores e a entrega contínua de novas funcionalidades. Nesse modelo, o desenvolvimento principal é dividido entre dois ramos estáveis:

- O `main`, que representa o código em produção;
- E o `develop`, que concentra o código em desenvolvimento.

A partir deles, são criados ramos auxiliares para diferentes propósitos, entre os quais ramos de trabalho vinculados a tarefas ou *issues*, ramos `release/` para preparar versões estáveis antes do lançamento e ramos `hotfix/` para correções urgentes, em geral a partir de `main`.

As versões publicadas no E-Museu seguem o versionamento semântico (SemVer): número *MAJOR.MINOR.PATCH* e, antes de uma release estável, identificadores de pré-lançamento `-alpha` (cortes iniciais) ou `-beta` (homologação) (Preston-Werner, Tom, 2026).

A Figura 1 ilustra a topologia desses ramos no modelo clássico, em diagrama elaborado com base no *Gitflow* tradicional (Git Documentation Team, 2025).

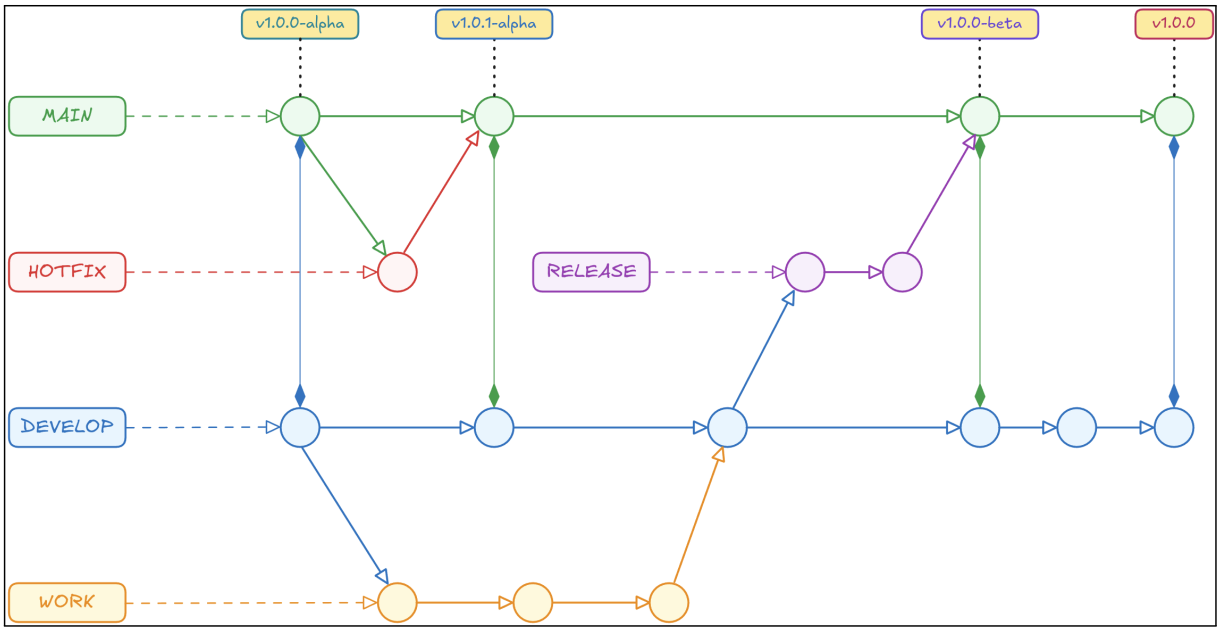


Figura 1 – Fluxo de *branches* no *Gitflow*.
Fonte: Adaptado de Git Documentation Team (2025). Elaboração própria no *Excalidraw*.

Dessa forma, o *Gitflow* estabelece um ciclo de vida bem definido para cada etapa do projeto, reduzindo conflitos de código e aumentando a rastreabilidade das mudanças. Durante o desenvolvimento do E-Museu, contudo, verificou-se que a aplicação rigorosa de convenções auxiliares (nomenclatura extensa, mensagens padronizadas, *templates* detalhados de *issues* e de *pull requests*) podia tornar-se excessivamente burocrática para o contexto de um projeto acadêmico com equipe reduzida. Por isso, elaborou-se e publicou, na versão 2.0.0 do repositório de documentação do fluxo (Novacoski, Vinicius Ferreira, 2026), uma adaptação que mantém a topologia do *Gitflow* (`main`, `develop`, *release* e *hotfix*), mas oferece dois perfis de adoção, ambos orientativos e ajustáveis conforme o projeto:

- *Perfil full:* ramificações no formato `@autor/tipo/número-da-issue/nome-curto` (por exemplo, `@vinifen/feat/123/user-dashboard`), mensagens de *commit* com emoji e tipo semântico (`feat`, `fix`, `chore`, entre outros) e *templates* completos de *issues* e *pull requests* no *GitHub*;
- *Perfil simple:* nomenclatura enxuta `tipo/número-da-issue/nome-curto` (por exemplo, `feat/123/user-dashboard`), mensagens no formato `tipo: descrição`, sem emoji, e *templates* reduzidos, preservando o mesmo fluxo de integração em `develop` e de promoção para `main`.

No E-Museu adotou-se o perfil *full*, por equilibrar rastreabilidade (vinculação explícita entre ramo, *commit* e *issue*) e disciplina de revisão em *pull requests*, sem abandonar os ramos permanentes nem a associação prevista entre o ramo de desenvolvimento (ambiente de *staging*) e o ramo principal (produção) na implantação institucional (Seção 3.2.5). A estrutura mais rica de *issues* e de *pull requests* do perfil *full* também favorece a documentação contínua do trabalho (descrição das mudanças, tempo dedicado, tipo de alteração e ligação a tarefas), o que se mostrou útil como registro ordenado do desenvolvimento na redação desta monografia. O perfil *simple* permanece documentado como alternativa mais direta para contextos em que a sobrecarga operacional do perfil *full* não se justifica.

3.2.4 Refatoração e desenvolvimento de novas funcionalidades

Com o levantamento de requisitos, a priorização *MoSCoW* e a definição do fluxo de *Gitflow* estabelecidos, seguiu-se a fase principal do projeto, voltada ao desenvolvimento efetivo do sistema. Essa etapa concentrou-se, em grande medida, no ambiente local containerizado (Seção 3.1.3), onde foram evoluídos o banco de dados, a topologia da aplicação (aplicação, servidor web, *MySQL*, *Redis* e demais componentes) e as funcionalidades previstas no escopo. Em paralelo, utilizou-se um *Coolify* simulado em VPS na DigitalOcean (Seção 3.1.4) para ensaiar *deploy* automatizado e a composição dos serviços em condições próximas às de *staging* e de produção, antes da replicação no servidor da UTFPR. O trabalho seguiu o fluxo de *Gitflow* definido anteriormente, com documentação e controle por meio do *GitHub* (*pull requests* e *issues*). Junto ao desenvolvimento foram executados testes e verificações automatizadas de qualidade, conforme as ferramentas descritas na seção de materiais, para garantir a estabilidade das funcionalidades novas e das já existentes.

3.2.5 Implantação do ambiente institucional na UTFPR

Embora, no planejamento inicial, a automação de *deploy* na UTFPR figurasse entre as primeiras necessidades do projeto, verificou-se durante a execução que seria mais prático consolidar previamente boa parte do sistema fora do campus, em especial o esquema do banco de dados, a composição dos serviços em Docker e as funcionalidades centrais, e somente então replicar essa arquitetura no servidor institucional. A ida presencial à UTFPR para aplicar o E-Museu concentrou-se, portanto, nas fases finais do trabalho, quando o repositório já oferecia base estável para publicação.

Nessa etapa tardia, replicou-se no campus a configuração já ensaiada na VPS com *Coolify* (Seção 3.1.4), instalando o *Coolify* na máquina física em que o sistema está hospedado e associando o ramo de desenvolvimento ao ambiente de testes (*staging*), em https://staging.

e-museu.gp.utfpr.edu.br/, e o ramo de produção ao ambiente publicado em https://e-museu.gp.utfpr.edu.br/.

Por política de segurança institucional, não há acesso remoto ao servidor da UTFPR no sentido de administração direta da máquina; em um primeiro momento, isso exigiu presença no campus para configurar o *Coolify* e os serviços associados. Posteriormente, o *Coolify* passou a ser acessível no subdomínio institucional https://coolify.e-museu.gp.utfpr.edu.br/, de modo que a gestão de projetos Docker, das configurações de implantação e dos *deploys* pudesse ser feita pela interface web sem deslocamento a cada ajuste rotineiro, mantendo *staging* e produção nos domínios citados acima. As tentativas anteriores de automação de *deploy* e a decisão de publicar esse subdomínio são detalhadas na Seção 4.8.

3.2.6 Experimentações e testes

Por fim, após a implantação institucional (Seção 3.2.5), foram realizadas etapas de validação no ambiente de *staging*, permitindo verificar as implementações em condições próximas às de produção. Somente após essa validação o sistema foi promovido ao domínio principal. Quando identificados erros ou inconsistências, o fluxo retornava à etapa de desenvolvimento local e correção antes de nova implantação no servidor, em conformidade com o ciclo de *pull requests*, testes automatizados e *deploy* descrito no Capítulo 4.

4 DESENVOLVIMENTO DO SISTEMA

Este capítulo apresenta o desenvolvimento realizado na refatoração e na implementação de novas funcionalidades no E-Museu. O texto organiza-se da seguinte forma: planejamento e versionamento; infraestrutura; refatoração; testes; funcionalidades; atualização tecnológica da aplicação; e implantação na UTFPR. Muitas vezes as atividades foram realizadas em paralelo.

Quanto ao versionamento, as *releases* com sufixo `-beta` (de `2.0.0-beta` a `5.0.0-beta`) marcaram marcos de homologação no fluxo Gitflow: código estável o suficiente para validar em *staging*, mas ainda sujeito a ajustes antes da produção institucional. O rótulo `1.0.0`, promovido depois dessa sequência, designa a primeira versão considerada estável para produção, sem o sufixo `-beta`. A numeração não “regride”: encerra-se o ciclo de ensaios com `-beta` e abre-se a linha estável em `1.0.0`.

4.1 Levantamento de requisitos e priorização

O levantamento de requisitos constituiu a primeira etapa do trabalho. Os requisitos foram identificados com base no TCC de Gonzaga (2024) e refinados posteriormente (observações da professora orientadora, do coorientador e da banca de TCC1). Assim, identificou-se no sistema E-Museu a necessidade de aprimoramento na estrutura do código e de testes automatizados, o que reforçou a demanda por refatoração e por novas funcionalidades. Com o refinamento dos requisitos, o trabalho ficou mais alinhado aos objetivos do TCC.

Desta forma, com os requisitos definidos e refinados, foi aplicada a técnica de priorização MoSCoW, com o objetivo de direcionar o desenvolvimento para as funcionalidades de maior impacto no sistema, otimizando o uso do tempo, do esforço e dos recursos disponíveis. O resultado dessa priorização é apresentado na Figura 2.

Priorização *MoSCoW* - Refatoração do Sistema E-Museu		
MoSCoW	Requisito	Descrição
MUST	Refatorar os *Controllers*, adicionar *Services* e *Actions*.	A refatoração dos *Controllers* deve remover toda a lógica de negócio e realocá-la em *Services* e *Actions*, garantindo que os atuais 18 *Controllers* atuem apenas como ponto de entrada das requisições. Essa reorganização deverá melhorar a manutenção, clareza e testabilidade do código, enquanto partes adicionais da lógica poderão ser fragmentadas em *Actions* específicas. Nenhuma nova funcionalidade será adicionada neste processo, focando exclusivamente na reestruturação interna.
	Implementar *deploy* automatizado.	Implementar um processo de *deploy* automatizado na máquina física da UTFPR usando o *Coolify*, garantindo a publicação contínua do site principal (a partir da *branch main*) e de um ambiente de *staging* (a partir da *branch develop*), permitindo atualizações rápidas e sem intervenção manual.
SHOULD	Implementar testes e qualidade de código.	Implementar testes automatizados e mecanismos de verificação para manter a qualidade e a consistência do código durante o desenvolvimento.
	Aumentar a capacidade de upload de fotos e vídeos.	Ampliar a funcionalidade atual para permitir o envio de múltiplas fotos por peça, garantindo uma apresentação mais completa e detalhada do conteúdo.
	Internacionalização do sistema.	Implementar a internacionalização no código utilizando i18n, permitindo que a aplicação ofereça suporte a múltiplos idiomas e facilitando a adaptação de textos e interfaces para diferentes públicos.
	Geração de etiquetas para peças físicas.	Criar um mecanismo para gerar etiquetas contendo o código e o nome de cada peça, permitindo sua impressão e colagem em itens físicos para facilitar identificação e organização.
COULD	Mecanismos de segurança.	Implementar ferramentas de proteção, como o *reCAPTCHA*, para fortalecer a segurança do sistema e evitar acessos ou ações automatizadas maliciosas.
WON´T	Reestilizar o *design* e implementar no E-Museu.	A reestilização do *design* do E-Museu foi cogitada como uma possível melhoria visual e de experiência do usuário; porém, após avaliar o escopo e as prioridades atuais, decidiu-se não realizar a alteração neste momento. Além disso, qualquer mudança exigiria uma pesquisa prévia para validar a necessidade de alterar o *design* atual, seguida de etapas de prototipação e posterior implementação no código, aumentando significativamente o esforço envolvido.

Figura 2 – Priorização *MoSCoW*.
Fonte: Autoria própria (Google Planilhas).

4.2 Organização do repositório e fluxo de desenvolvimento

Esta seção descreve como o código do E-Museu foi versionado no *GitHub* durante a atualização do sistema. O trabalho de implementação concentrou-se no repositório da aplicação (refatoração, infraestrutura, funcionalidades e implantação nas seções seguintes). Antes desse *fork*, foi iniciado um repositório à parte com a documentação reutilizável do fluxo (Novacoski, Vinicius Ferreira, 2026), produto extra em relação ao software do museu, retomado e ampliado durante a atualização do E-Museu. O modelo *Gitflow*, os perfis *full* e *simple* e a topologia de ramos estão no Capítulo 3; aqui, importam a ordem dos repositórios e o uso efetivo dessas convenções no E-Museu.

4.2.1 Repositórios no *GitHub* e histórico de entregas

O desenvolvimento no *GitHub* começou com um *fork* do projeto original descrito por Gonzaga (2024) para um repositório diferente (Novacoski, Vinicius Ferreira, 2025). Um *fork* é uma cópia ligada ao projeto original, mas com histórico e alterações independentes; assim,

foi possível refatorar o sistema e implementar novas funcionalidades sem alterar o repositório legado.

Quase todo o trabalho descrito neste capítulo começou nesse repositório (`vinifen/e-museu-2.1.0-beta`), com ramificações auxiliares integradas em `develop` (*staging*) e, quando estável, em `main` (produção). Desde o primeiro *pull request* após o *fork*, o fluxo seguiu o perfil *full* definido no Capítulo 3 (ramos, *commits*, *issues* e *pull requests* com rastreio entre tarefa, ramo e histórico), aplicado no dia-a-dia.

O histórico de *pull requests* reflete a ordem das frentes de trabalho: primeiro a infraestrutura (*Docker*, `run`, *Coolify* na VPS e versões *beta* até a `2.0.0-beta`); em seguida, ferramentas de qualidade e integração contínua; depois, reorganização por domínios, funcionalidades de catálogo e ampliação dos testes; por fim, implantação na UTFPR. Cada entrega foi revisada antes da promoção para `develop` e, quando aplicável, para `main`, com os trabalhos de testes e de análise estática no *GitHub Actions* descritos no Capítulo 3 como critério mínimo de aceitação.

Após a *release* `2.1.0-beta` no repositório (Seção 4.3.2), criou-se a organização do projeto no *GitHub* e o repositório institucional (e-museu-utfpr-gp, 2026), também por *fork* desse histórico e não do legado de Gonzaga (2024). A partir desse ponto, a refatoração, as funcionalidades e as integrações seguintes concentraram-se em `e-museu-utfpr-gp/e-museu`, enquanto o fluxo *Gitflow* permaneceu igual. Em síntese: *fork* do legado para o novo repositório; desenvolvimento com infraestrutura, qualidade e `2.1.0-beta`; e, em seguida, *fork* para a organização até o encerramento do ciclo.

4.2.2 Documentação do *Gitflow* como produto paralelo

O repositório dedicado ao fluxo (Novacoski, Vinicius Ferreira, 2026) foi iniciado antes do *fork* do legado e do desenvolvimento. Nele, foram definidos o modelo *Gitflow* adaptado e o perfil *full* que, em seguida, passaram a orientar o repositório da aplicação (wiki do projeto, *templates* no *GitHub* e rotina de ramos e *commits*). A versão 2.0.0 desse repositório consolidou, com base na experiência no E-Museu, os dois perfis orientativos descritos no Capítulo 3. No museu virtual, o perfil *full* foi mantido até o encerramento, devido à rastreabilidade entre ramo, *issue* e histórico de integração. A documentação externa pode evoluir conforme a necessidade em projetos futuros, independentemente das *releases* do E-Museu.

4.3 Infraestrutura inicial e ambientes de desenvolvimento

As Subseções 4.3.1 e 4.3.2 descrevem o trabalho realizado no repositório `vinifen/e-museu-2.1.0-beta`, até a *release* `2.1.0-beta` em `main`. A refa-

toração e as entregas seguintes do capítulo continuam no repositório da organização `e-museu-utfpr-gp` (Seção 4.2.1).

Com o repositório e o fluxo de integração definidos (Seção 4.2), iniciou-se a implementação no código do E-Museu. Em relação ao legado de Gonzaga (2024), faltava um ambiente reproduzível (instalação manual, sem *Docker* nem rotina única de comandos). A primeira entrega abrangeu essa base técnica e não alterou as regras de negócio do acervo, porque sem ambiente repetível qualquer refatoração posterior seria difícil de validar; as mudanças funcionais ficaram para as etapas seguintes. As ferramentas e a composição de serviços estão na Seção 3.1.3; nessa seção, descreve-se como essa infraestrutura foi introduzida e evoluída no repositório.

4.3.1 Infraestrutura inicial, *Coolify* e versões *beta*

Esta etapa marca o início efetivo do desenvolvimento no *fork*, após o fluxo de versionamento (Seção 4.2). O pacote principal de infraestrutura entrou em homologação após alinhamentos sobre requisitos (Seção 4.1), priorizando que qualquer pessoa do grupo pudesse subir o sistema com os mesmos comandos. O *Coolify* (Seção 3.1.4) foi utilizado para publicar o sistema no servidor da UTFPR, com *Docker* e *deploy* automatizado.

O pacote reuniu *Docker Compose* para desenvolvimento e produção, o script `./run`, compilação de *assets* com Vite e SCSS (Seção 3.1.11), ajustes pontuais no *back-end* (cabeçalhos no *HTTP* local, autocompletar do painel, *seeders* de desenvolvimento mais leves) e os ensaios de publicação com *Coolify*.

Ambientes em *Docker Compose* e script `run`

Para facilitar a execução do projeto foi definido dois perfis: o `local` (desenvolvimento) e o `production` (produção), cada um com `docker-compose` e *Dockerfile* próprios. O `APP_ENV` no `.env` indica qual perfil usar e o `./run` escolhe o Compose em todos os comandos (`setup`, `up`, `down`, `migrate`, `seed`). No `local`, a aplicação em PHP-FPM foi carregada, o *Nginx*, o *MySQL* 8 e o *phpMyAdmin* de apoio. No ambiente `production`, o *Dockerfile* foi configurado para *build* multiestágio (dependências PHP, *assets* compilados, imagens `app` e `web`). A prioridade desta etapa foi deixar o sistema executável de forma repetível no container. Os refinamentos do *front-end* e da suíte de testes foram deixados de lado nesta etapa, sendo implementados e executados em entregas posteriores.

Ensaio com *Coolify* e versionamento *SemVer*

Posteriormente à definição da base local, os arquivos foram adaptados ao *Coolify*, incluindo o perfil `staging` ligado à ramificação `develop`. O *Coolify* foi instalado em uma VPS na *DigitalOcean*. Repetiu-se o ciclo de ensaio do que depois seria implantado no servidor do E-Museu no Campus Guarapuava: promover `develop` para `main`, disparar *deploy* por *webhook*, corrigir o que falhasse e voltar a publicar. As versões seguiram o *SemVer* (Seção sobre *Gitflow*

no Capítulo 3), com sufixos `-alpha` ou `-beta` conforme a maturidade da entrega. Nessa etapa de ensaio no *Coolify* prevaleceram as marcações `-beta`.

A primeira *release* promovida para `main`, `1.0.1-beta`, serviu sobretudo para testar o *deploy* automático no *Coolify* com a infraestrutura já montada. Na prática do *Coolify*, surgiram falhas de separação entre homologação e produção. Logo, foram corrigidas com *hotfixes* que formalizaram o `docker-compose.staging.yml`, alinharam nomes de rede e serviços, além da criação de configurações do *Nginx* para o ambiente (`local`, `staging` e `production`). Em seguida, atualizou-se a pilha para *PHP* 8.5 e Laravel 12 (entre outras dependências de *front-end* e de testes), criou-se o arquivo `VERSION` no repositório e o comando de versão no `./run` passou a ser utilizado. Esse conjunto de atividades compôs a *release* `2.0.0-beta`, que marca, no versionamento do projeto, a infraestrutura estável para as etapas seguintes. A publicação na UTFPR é relatada na Seção 4.8.

4.3.2 Integração contínua, ferramentas de qualidade e *release* `2.1.0-beta`

Com a infraestrutura e a `2.0.0-beta` consolidadas, iniciou-se o atendimento às modificações da versão anterior (i.e., falta de mecanismos automatizados de análise estática e de validação do comportamento), em alinhamento com a justificativa do trabalho (Seção 1.2). Além disso, incluiu-se *Continuous Integration/Continuous Delivery* (CI/CD) e ferramentas de qualidade no repositório. A entrega concentrou-se em duas camadas: executar as mesmas verificações no ambiente local (script `./run`) e repetir o essencial no *GitHub Actions* antes de integrar em `develop` ou `main`. O conjunto de ferramentas está descrito no Capítulo 3.

Ferramentas locais e correção do legado

Posteriormente, incorporou-se ao ramo de homologação o pacote de qualidade (*issue* “implement tools to ensure quality”). A escolha por analisadores estáticos locais e na nuvem respondeu à ausência desses mecanismos no legado: sem eles, a refatoração por domínios correria o risco de reintroduzir erros de estilo, duplicação ou tipos inconsistentes. Foram adicionados ou formalizados analisadores para o *back-end* em *PHP* (*PHPStan* com *Larastan* em nível elevado, *PHPCS*/*PHPCBF* em PSR-12, *PHPMD*, *PHP Insights*, detecção de duplicação com *phpcpd*, *Churn* para candidatos a refatoração e *Laravel Pint*) e para o *front-end* em *JavaScript* (*ESLint* e *Prettier*, alinhados ao Vite e ao SCSS da infraestrutura inicial). Cada ferramenta possui configuração versionada no repositório e atalhos no script `./run`, de modo que o desenvolvedor pode reproduzir localmente o que, posteriormente, seria executado na nuvem.

Além da instalação de pacotes, o código herdado de Gonzaga (2024) foi ajustado até que as verificações passassem sem erros conhecidos, o que preparou a refatoração por domínios (Seção 4.4) sobre uma base já legível para as ferramentas. A suíte de testes do *PHPUnit*, nesse momento, ainda era enxuta e a ampliação sistemática dos casos de funcionalidade aconteceu posteriormente (Seção 4.5).

Pipeline no *GitHub Actions*

A primeira versão da integração contínua (*issue* de pipeline de *Continuous Integration/-Continuous Delivery* (CI/CD)) ocorreu depois da instalação de pacotes e das verificações de erros do código herdado por Gonzaga (2024). A *action* disparou em cada *pull request* para `main` ou `develop` e dividiu o trabalho em três *jobs* paralelos, todos apoiados no *Docker* e no `./run` já utilizados no dia-a-dia:

- *tests*: faz *upload* do ambiente com `./run setup -env -y`, executa `./run test` e encerra os *containers* ao final;
- *code-quality*: no mesmo ambiente local, executa *PHPCS*, *PHPMD*, *PHPStan*, *phpcpd*, *ESLint* e faz a verificação de formatação com *Prettier*;
- *test-production-deploy*: valida um *deploy* simulado com `docker-compose.production.test.yml`, incluindo *build* das imagens de produção, subida dos serviços, espera pelo *MySQL* e comandos típicos de publicação Laravel (`migrate -force`, *cache* de rotas e configuração).

Assim, o critério de aceitação deixou de depender somente da revisão manual, ou seja, as integrações que não passavam pelos testes, estilo ou pelo fluxo mínimo de produção em container eram barradas antes do *merge*. O *Laravel Pint* passou a fazer parte da verificação automática em etapa posterior ao cronograma, sendo que a padronização *PSR*-12 e o Laravel ficaram a cargo de *PHPCS* e das correções locais.

***Release* `2.1.0-beta` e transição para a organização**

Posteriormente a etapa de pipeline no *GitHub Actions*, realizou-se o `develop` para `main` na *release* `2.1.0-beta`, reunindo em `main` as ferramentas de qualidade e a pipeline de *Continuous Integration/Continuous Delivery* (CI/CD). No *SemVer* (Capítulo 3), a mudança de `2.0.0-beta` para `2.1.0-beta` sinalizou um incremento das funcionalidades de engenharia (automação e análise estática), sem alterar a pilha *PHP* 8.5 e Laravel 12 fixada na infraestrutura anterior.

Essa *release* encerra, no repositório do trabalho, o trabalho inicial que começou com o *Docker*, `run` e ensaios no *Coolify*. A infraestrutura, a qualidade local e a integração contínua passaram a fazer parte da linha `main` antes das grandes refatorações de domínio e das funcionalidades de acervo. Ainda, criou-se a organização `e-museu-utfpr-gp` e o repositório institucional por *fork* desse histórico (Seção 4.2.1). O desenvolvimento seguinte continuou no mesmo fluxo *Gitflow*. A refatoração do código no repositório da organização é relatada na Seção 4.4.

4.4 Refatoração da aplicação

Com a `2.1.0-beta` no repositório (Seção 4.3.2), o desenvolvimento migrou para o repositório da organização `e-museu-utfpr-gp/e-museu`, mantendo o mesmo fluxo *Gitflow*. O objetivo foi reduzir o acoplamento no legado de Gonzaga (2024) e alinhar o código às boas práticas citadas no referencial (MVC, *Clean Code* e organização por domínio, Seção 2.3.3). Essa seção trata disso.

4.4.1 Reorganização inicial e base técnica

A primeira etapa da refatoração preparou o repositório da organização para os ajustes de *deploy* e de *Continuous Integration/Continuous Delivery* (CI/CD), adoção de domínios no código, armazenamento de imagens com *MinIO*, *release* `3.0.0-beta` e alinhamento à estrutura do Laravel 12.

Ajustes de infraestrutura no repositório da organização

Antes de mover as pastas de domínio, corrigiu-se o que ainda falhava para o *Coolify* e para a *Continuous Integration/Continuous Delivery* (CI/CD) no novo repositório remoto. Foram movidas as redes Docker customizadas que o painel passava a gerenciar sozinho; a conexão ao *MySQL* em homologação e produção passou a usar URL configurável no ambiente, em vez de depender só de *host* fixo no Compose, o que facilitou o banco provisionado no *Coolify*. Testou-se *healthcheck* no *Coolify*, mas a verificação fazia a aplicação falhar no painel; um *hotfix* em `main` manteve a publicação sem *healthcheck*, por ser mais estável.

Em homologação, adicionou-se autenticação HTTP básica opcional para restringir o acesso público ao *staging* durante os ensaios. A pipeline do *GitHub Actions* foi atualizada para o novo repositório, de modo que os *jobs* de testes, de qualidade e de *deploy* simulado continuassem válidos após o *fork*. Esses passos são complementos da infraestrutura da Seção 4.3, não alteram as regras de negócio do acervo e antecedem a reorganização estrutural do código.

Organização por domínios (DDD)

Ainda, grande parte das classes permanecia em pastas genéricas de controladores, serviços e modelos, o que dificultava localizar as regras do acervo, da taxonomia ou dos contribuidores externos. A reorganização por domínios passou para homologação porque o referencial teórico (Seção 2.3.3) e a Tabela 1 exigiam agrupar responsabilidades por contexto de negócio, e não por tipo técnico do *framework*. Modelos, serviços, controladores, requisições de validação e, quando aplicável, *factories* e *seeders* passaram a agrupar-se em `Catalog`, `Taxonomy`, `Proprietary` (contribuidores do catálogo) e `Identity` (contas administrativas e travas de edição). Arquivos de configuração centralizam o vínculo entre *factory* e modelo e reduzem acoplamento nas travas de edição.

Nesta fase, o agrupamento de contribuidores ainda usava o vocabulário legado *Proprietary*; a padronização para *Collaborator*, `item_category` e `tag_category` ocorreria na Subseção 4.4.2. O acervo seguia monolíngue no banco, ainda não existiam a tabela `languages`, o modelo `Language` e nem as tabelas `*_translations` (somente posteriormente, Subseção 4.6.1). Após isso, antecipou-se a internacionalização da interface com arquivos `lang/` (Subseção 4.4.3); `Location` e o domínio Admin (IA) em momento posterior (Seção 4.6).

A Tabela 1 resume a estrutura consolidada do repositório ao final das etapas deste capítulo. Isso para o leitor localizar responsabilidades no código refatorado.

Tabela 1 – Domínios do E-Museu e principais responsabilidades no código.

Domínio	Responsabilidade e organização em `app/`
Catalog	Núcleo do acervo: itens, traduções, imagens, componentes, vínculos com etiquetas, categorias de item, *extras* e fluxos públicos de contribuição e de consulta. *Organização no código:* • `Models/Catalog`, `Services/Catalog` e `Actions/Catalog`; • `Http/Controllers/Catalog` (catálogo público); • `Http/Controllers/Admin/Catalog` (painel administrativo).
Taxonomy	Classificação transversal: etiquetas (*tags*) e categorias de etiqueta, com textos por idioma. *Organização no código:* • `Models/Taxonomy` e `Services/Taxonomy`; • `Http/Controllers/Admin/Taxonomy`; • `Http/Requests/Taxonomy`.
Collaborator	Pessoas que contribuem com itens ou *extras*: cadastro, papéis, bloqueio, verificação de e-mail e elegibilidade para o catálogo público. *Organização no código:* • `Models/Collaborator` e `Services/Collaborator`; • `Http/Controllers/Catalog/CollaboratorController` (contribuição pública); • `Http/Controllers/Admin/Collaborator`; • `Enums/Collaborator` e mensagens em `Mail/`.
Identity	Identidade administrativa: contas de curadoria, autenticação de *login* e travas de edição concorrente (*locks*) em telas sensíveis. *Organização no código:* • `Models/Identity` e `Services/Identity`; • `Http/Controllers/Admin/Identity`; • `Http/Middleware/Auth`; • `Console/Commands/Identity`.
Admin (IA)	Tradução assistida de conteúdos multilíngues no painel (itens, categorias, etiquetas), com provedores externos de *chat completion*. *Organização no código:* • `Actions/Admin/Ai`; • `Support/Admin/Ai` e `Client/Ai`; • `Http/Controllers/Admin/Ai`.

Fonte: Autoria própria, com base na estrutura do repositório `e-museu` (versão refatorada). Nota: em fevereiro de 2026 vigoravam sobretudo `Catalog`, `Taxonomy`, `Proprietary` e `Identity`, com acervo ainda monolíngue no banco; as demais linhas e `Language` em `app/Models/` refletem o estado após março e abril. `Location` também fica em `app/Models/` na raiz; `StorageProxyController` e `Support/` tratam infraestrutura transversal.

Dentro de cada domínio, manteve-se uma divisão de papéis coerente com o *framework*: controladores com responsabilidade HTTP reduzida, *Actions* para casos de uso compostos, `Services` para regras de aplicação e modelos Eloquent para persistência. As validações de

entrada ficaram em *Form Requests*, no mesmo agrupamento; *factories* e *seeders* seguiram a mesma lógica (detalhes no repositório institucional).

Armazenamento de imagens e MinIO

Em paralelo à reorganização por domínios, corrigiu-se o fluxo de imagens herdado do monólito: caminhos inconsistentes, ligação frágil entre disco e URL em *container* e exposição direta de arquivos no servidor web. Ajustou-se a camada de armazenamento do Laravel e o vínculo entre disco público e URLs servidas ao navegador. Em seguida, configurou-se o *MinIO* (Seção 3.1.4) para homologação e produção, evitando depender do sistema de ficheiros do servidor web para cada foto do acervo.

No *back-end*, o disco de armazenamento passou a depender de variáveis de ambiente: em desenvolvimento local permanece o disco local; em *staging* e produção, com credenciais no *Coolify*, usa o driver compatível com S3 apontando ao *MinIO*. Para não expor o serviço de objeto ao visitante, criou-se um *proxy* interno que atende pedidos de mídia pela URL da aplicação, lendo o arquivo pelo disco configurado (local ou S3).

***Release* `3.0.0-beta`**

A *release* `3.0.0-beta` promoveu `develop` para `main`. No SemVer, o salto de `2.1.0-beta` para `3.0.0-beta` marcou uma mudança estrutural relevante, como domínios no código, armazenamento preparado para objeto e ajustes de infraestrutura do repositório da organização descritos acima, ainda sem a segunda onda de refatoração de controladores e de *views*.

Alinhamento à estrutura do Laravel 12

Após a versão `3.0.0-beta`, concluiu-se em homologação a refatoração de base alinhada ao Laravel 12: remoção de artefatos supérfluos da transição anterior e concentração de *middleware*, rotas, comandos e exceções no bootstrap do *framework*. Os controladores administrativos deixaram de depender de uma hierarquia rígida de classe base para travas de edição; o comportamento de *lock* passou a um *trait* reutilizável e a configuração centralizada de rotas bloqueáveis. Além disso, as regras de validação duplicadas migraram para *Form Requests*; respostas 404 passaram a ser consistentes; corrigiram-se atribuições em massa inseguras em atualizações pontuais no painel.

Nesta mesma etapa, entrou o *Laravel Pint* na pipeline de qualidade (Seção 4.3.2), finalizando o que a primeira versão da *Continuous Integration/Continuous Delivery* (CI/CD) ainda não executava. Com isso, a reorganização inicial foi encerrada, com o repositório da organização tendo domínios, armazenamento por proxy, `3.0.0-beta` em `main` e código alinhado às convenções do Laravel 12 vigente na época.

4.4.2 Refatoração dos controladores e do modelo de dados

A segunda etapa concentrou-se em alinhar o vocabulário do banco e as rotas administrativas (Subseção 4.4.2), introduzir múltiplas imagens, retirar o código não utilizado do legado (Subseção 4.4.2) e, em seguida, transferir as regras dos controladores para os serviços, *Actions* e *Form Requests* (Subseção 4.4.2). Parte dessas entregas coincidiu no calendário com a preparação da interface (Subseção 4.4.3), mas a ordem lógica do texto segue os blocos abaixo.

Vocabulário do banco e papéis administrativos

Um pacote de migrações e de renomeações alinhou o banco e o código ao vocabulário dos domínios (Tabela 1), finalizando o que ainda estava sob nomenclatura legada de contribuidores, seções e contas genéricas. A tabela de contas do painel passou a refletir apenas curadoria (sem coluna que distinguisse “admin” de visitante): quem autentica no sistema é sempre equipe do museu. As travas de edição e as sessões no banco passaram a referenciar o administrador logado de forma explícita.

No catálogo, seções tornaram-se categorias de item; na taxonomia, categorias genéricas tornaram-se categorias de etiqueta. Contribuidores externos ganharam enum de papel (*internal* ou *external*), com regra de negócio para reservar e-mails institucionais a colaboradores internos. As rotas e validações do painel acompanharam os novos nomes. O estado consolidado do esquema é sintetizado nas Figuras do Capítulo 5 (Seção 5.2).

Esquema de imagens do acervo

Paralelamente, o esquema de múltiplas fotos por peça foi realizado, classificado no MoSCoW como funcionalidade, mas necessário para o armazenamento em objeto e para o painel já refatorados (Subseção 4.4.1). Criou-se tabela dedicada a imagens por item (capa e galeria), migrou-se o caminho monolítico anterior e removeu-se a coluna única de foto. O modelo garante, no máximo, uma capa por item.

No painel e na contribuição pública, a lógica de envio, remoção, promoção de capa e galeria passou a tratar listas de identificadores e arquivos em lote; a validação distingue a imagem de capa obrigatória na contribuição e os campos opcionais da galeria. No formulário público de envio de item, a interface separa três blocos: capa obrigatória (com selo *Capa*), área para arrastar ou selecionar várias imagens de galeria e lista de miniaturas já adicionadas, com remoção individual e distinção entre capa e galeria (Figura 3). A captura utiliza arquivos de exemplo apenas para ilustrar o fluxo.

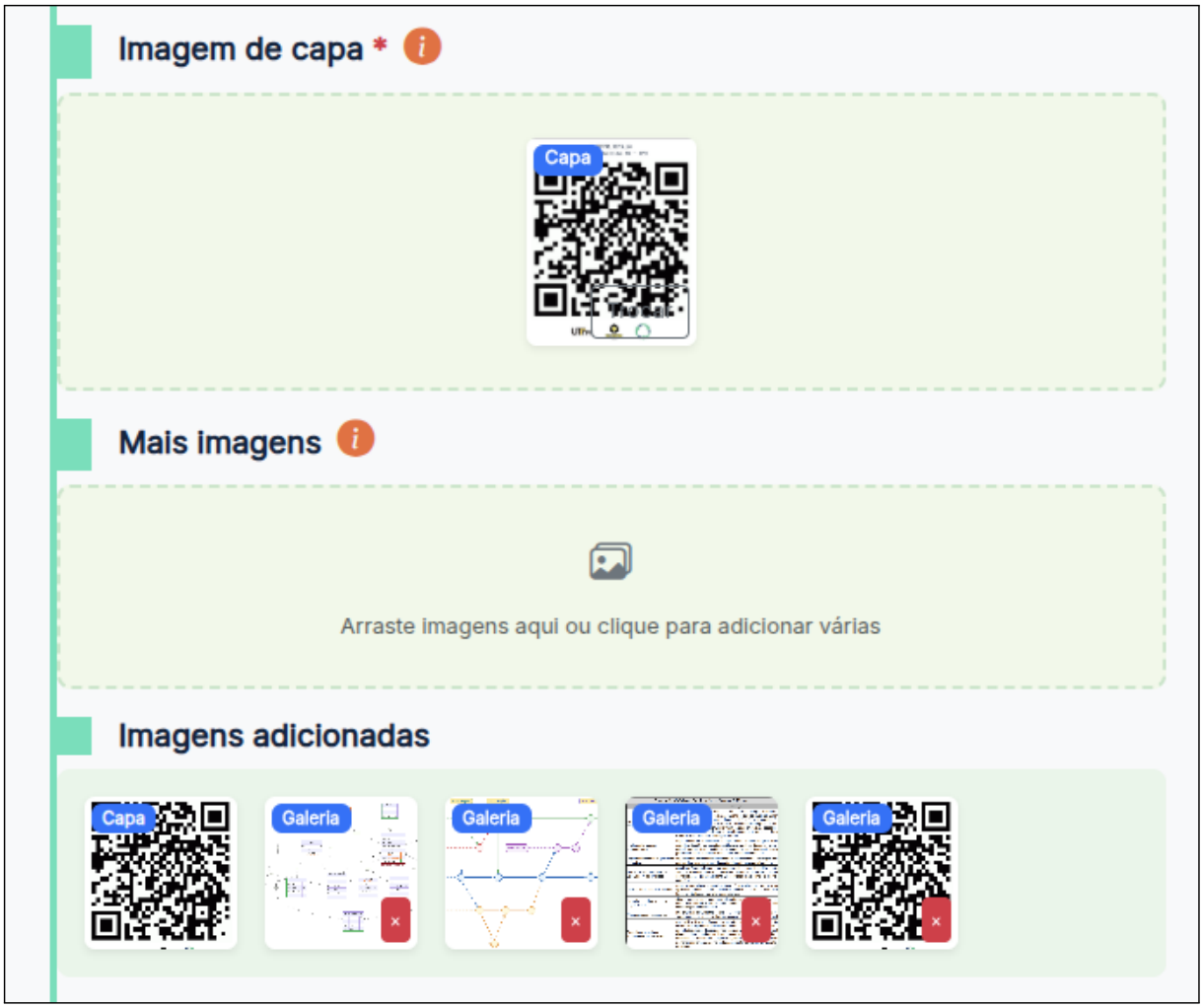


Figura 3 – Contribuição pública de item: capa, envio em lote e pré-visualização de imagens. Fonte: Autoria própria. Nota: miniaturas de demonstração, sem relação com o conteúdo real do item.

O mesmo contrato de dados e de serviços passou a ser consumido pelos controladores refatorados anteriormente e pelo JavaScript do Vite (Subseção 4.4.3); na ficha pública do item validado, capa e galeria aparecem com a resolução de mídia via proxy (Subseção 4.4.1).

Limpeza de legado

Nesta fase, removeu-se código que o *framework* Laravel traz por padrão, mas que o E-Museu refatorado não utilizava. Foram retirados os controladores e *views* de registro, de recuperação de senha e de verificação de e-mail voltados a um usuário genérico, permanecendo apenas o fluxo de *login* administrativo. Os *middlewares* que apenas repassavam a requisição ou duplicavam a validação foram eliminados. Assim, a checagem de item não validado no catálogo público ficou no controlador, com resposta HTTP 403 explícita. Retirou-se também a interface da assistente virtual legada (“Ada”), com *views*, scripts e textos associados. Isso foi feito para reduzir o ruído antes da reorganização das *views* Blade.

Camada de aplicação e controladores

Seguindo com a refatoração, três entregas sequenciais em homologação (refatoração de controladores em três partes) cobriram o legado em que consultas, transações e regras de negócio ficavam nos controladores. A meta foi restringir essas classes à camada HTTP (validar entrada, chamar serviço ou *Action*, devolver *view*, JSON ou redirecionamento) e concentrar o restante em serviços por domínio e em apoios reutilizáveis, porque controladores inchados dificultavam testes e repetiam lógica entre painel e catálogo público.

Depois disso, o painel de itens passou a serviços dedicados: índice com busca e ordenação, criação com código de identificação, atualização e ciclo de capa e galeria, sem consultas nem lógica de arquivo no controlador. A contribuição pública passou a ser executada por uma *Action* que, em transação, resolve o colaborador, grava o item, vincula etiquetas, *extras* e componentes e reutiliza o mesmo serviço de imagens do painel. A contribuição isolada de *extra* ganhou controlador e rota próprios, para não concentrar tudo em um único ponto. As validações do administrador migraram para *Form Requests* por domínio; busca e ordenação dos índices do painel saíram de *traits* genéricos e foram para consultas reutilizáveis. Complementaram a entrega a leitura pública de item validado, listagem por categoria em JSON e tratamento de datas opcionais nos itens.

Além disso, removeu-se um controlador auxiliar que misturava *endpoints* AJAX de etiquetas, autocompletar e checagem de contato sem pertencer a um domínio; as rotas foram redistribuídas aos controladores corretos, sempre via serviços. A listagem pública do acervo e os índices administrativos passaram a usar serviços e consultas padronizadas; travas de edição concentraram-se no domínio de identidade. *Actions* e serviços deixam de devolver redirecionamentos HTTP diretamente e respondem com *status* de domínio, que o controlador interpreta para mensagem e rota.

O `ItemController` do catálogo público resume bem o contraste entre legado e código refatorado. No sistema anterior, um único controlador acumulava consultas, validações repetidas, transações, métodos auxiliares e rotas AJAX no mesmo arquivo, com cerca de 398 linhas. Após a refatoração, o equivalente no domínio `Catalog` limita-se à camada HTTP: recebe serviços por injeção de dependência, chama um método do serviço ou da *Action* e devolve *view*, redirecionamento ou JSON. A contribuição pública, por exemplo, reduz-se a invocar a *Action* de armazenamento em vez de repetir dezenas de linhas de persistência no controlador.

A Figura 4 ilustra o início do `Catalog/ItemController.php` refatorado (à direita): métodos HTTP enxutos que delegam a `ItemService` e a `StoreItemContributionAction`, em contraste com o trecho inicial do legado (à esquerda), em que consultas e filtros do `index` permaneciam no controlador.

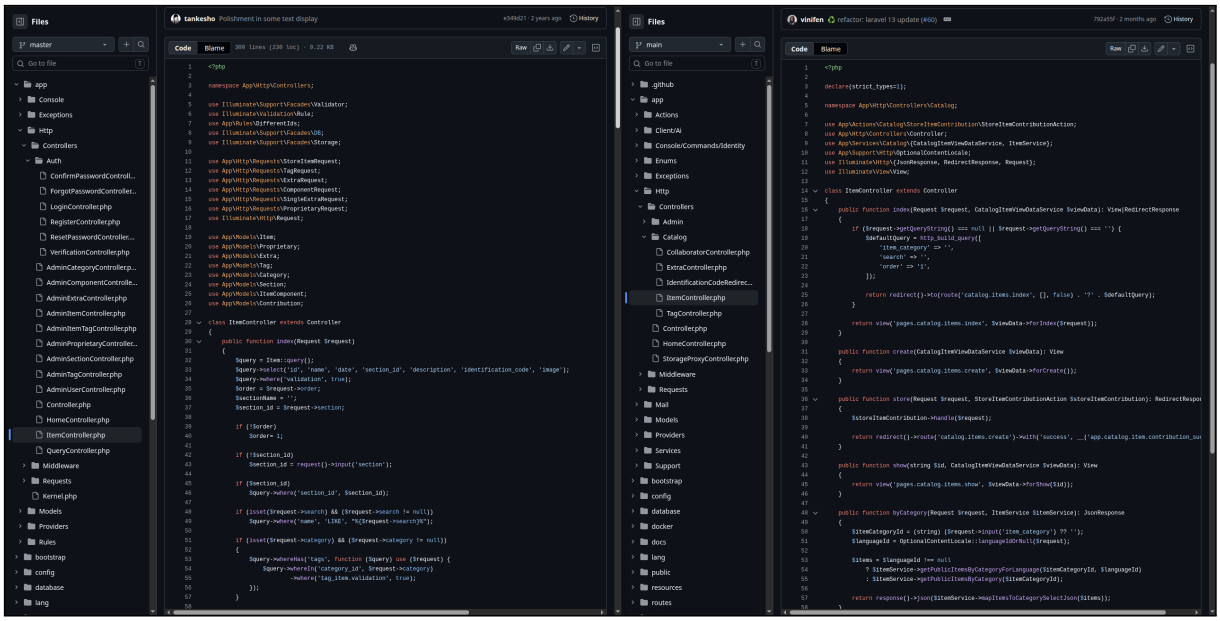


Figura 4 – `Catalog/ItemController.php` refatorado (início do arquivo; à direita), frente ao `ItemController.php` legado (à esquerda).

Fonte: https://github.com/e-museu-utfpr-gp/e-museu/blob/main/app/Http/Controllers/Catalog/ItemController.php e https://github.com/tankesho/e-museu/blob/master/app/Http/Controllers/ItemController.php (acesso em jun./2026).

Release `4.0.0-beta`

A *release* `4.0.0-beta` promoveu homologação para `main`. Após a `3.0.0-beta`, essa versão consolida em `main` o novo vocabulário do banco, o esquema de múltiplas imagens, a limpeza de legado e a camada de serviços nos controladores descritos acima. A reorganização ampla das *views* Blade, a componentização de listagens administrativas e a reestruturação dos módulos JavaScript no Vite (Subseção 4.4.3) permaneceram em homologação e só alcançaram `main` em entregas posteriores; o mesmo vale para a internacionalização completa do acervo e para outras funcionalidades.

4.4.3 Refatoração do *front-end* e das *views*

A terceira etapa abrangeu a interface: arquivos de idioma para PHP e JavaScript, reorganização das *views* Blade e das rotas nomeadas e, por fim, módulos JavaScript no Vite, retirando scripts inline que dependiam do novo esquema de imagens e dos *endpoints* JSON do painel. A internacionalização do conteúdo do acervo em tabelas de tradução e a tradução assistida por IA no painel (domínio Admin da Tabela 1) são relatadas na Seção 4.6.

Preparação de internacionalização da interface

Quanto a internacionalização, preparou-se em homologação um pacote amplo de internacionalização da interface, em paralelo às renomeações de banco da Subseção 4.4.2. O objetivo foi deixar de fixar textos em português diretamente nas *views* e no *JavaScript*, prepa-

rando o sistema para exibir rótulos e mensagens em português do Brasil e em inglês antes da reestruturação do acervo. O desenho geral da internacionalização no projeto (arquivos de idioma, Laravel Lang e *i18next* no cliente) está na Seção 3.1.12.

Os textos da interface (painel, catálogo público, validações do *framework* e mensagens de regras de negócio) passaram para pastas de idioma organizadas por área do sistema, em vez de strings fixas nas telas e nas validações. No navegador, avisos e confirmações concentraram-se em JSON consumidos pelo *i18next*, com módulo de inicialização e *build* do Vite ajustado nos *Dockerfiles* para incluir esses arquivos no *container*. O conteúdo multilíngue do acervo (tabelas de tradução, idioma do catálogo por sessão e tradução assistida no painel) ficou para inclusão posterior (Subseção 4.6.1), reutilizando a base preparada aqui.

Componentização das *views* e rotas

Após a *release* `4.0.0-beta` (Subseção 4.4.2), a reorganização estrutural das *views* Blade e das rotas nomeadas passou para homologação. O legado mantinha caminhos planos e repetia a mesma marcação *Bootstrap* em cada listagem do painel; a componentização visava reduzir duplicação e alinhar telas ao vocabulário dos domínios já refatorados. O papel do Blade, dos *layouts* e da separação entre site público e painel administrativo se encontra na Seção 3.1.9.

Adotou-se árvore com *layouts* e páginas por área, componentes reutilizáveis para listagens e trechos comuns do painel e *partials* nas páginas públicas maiores. As rotas passaram a agrupar catálogo e administração por domínio, com nomes alinhados aos controladores refatorados; classes de apoio concentraram busca, ordenação e rótulos dos índices administrativos, em vez de repetir configuração em cada controlador.

Além disso, um *hotfix* em `main` corrigiu a ficha pública do item, vínculos do banco com exclusão segura e busca de *extras* no painel, enquanto as telas componentizadas ainda não tinham entrado em homologação.

Reorganização dos recursos JavaScript

A reestruturação do JavaScript compilado pelo Vite entrou em homologação alinhada às *views* da Subseção 4.4.3 e aos serviços realizados. O legado servia scripts soltos na pasta pública e a infraestrutura inicial (Seção 4.3.1) já tinha adotado Vite (Seção 3.1.11); contudo, o código ainda se espalhava em componentes soltos e em blocos inline nas telas. Essa entrega organizou o cliente em torno de duas entradas (site público e painel), pastas compartilhadas e por página, carregamento sob demanda nos formulários mais pesados e retirada dos blocos inline, com *ESLint* alinhado aos novos módulos.

Com isso, encerrou-se a refatoração estrutural da interface: *views* componentizadas, rotas por domínio, dois *bundles* Vite e textos em arquivos de idioma (Subseções 4.4.3 e 4.4.3). As entregas posteriores (conteúdo multilíngue no banco, localização com QR Code, Turnstile, domínio Admin com IA e ampliação de testes) partem dessa base, sem reabrir o arranjo legado de scripts soltos.

4.5 Testes automatizados e qualidade de código

A ausência de testes automatizados na versão anterior foi um dos motivos da refatoração (Seção 1.2). As ferramentas de análise estática e o pipeline no *GitHub Actions* foram introduzidos logo após a infraestrutura inicial (Seção 4.3). O trabalho seguinte, em etapas posteriores do cronograma, concentrou-se em ampliar os testes de fluxo e de integração da aplicação refatorada.

Além disso, ampliou-se sistematicamente a cobertura funcional. Foram introduzidos casos de teste base compartilhados para cenários que exigem *MySQL* real, além de *fixtures* mínimas para *upload* de imagens sem depender de extensões gráficas no ambiente de *CI*. A consolidação ocorreu no *pull request* de testes abrangentes (#57/#58 no repositório do projeto), que removeu testes de exemplo, padronizou a injeção de dependências nos testes unitários de serviços e passou a exercitar, entre outros fluxos: autenticação e travas administrativas; CRUD e buscas do painel sobre itens, categorias, componentes, etiquetas, *extras* e colaboradores; contribuição pública ao catálogo e verificação de e-mail; resolução de traduções; *middleware* de idioma, autenticação, *staging* com autenticação HTTP básica e verificação *antibot*; proxy de armazenamento; e serviços de domínio (catálogo, colaboradores, identidade, taxonomia, localização). Ao final dessa etapa, o repositório contava com cerca de noventa arquivos de teste distribuídos entre testes unitários e de funcionalidade.

A validação de rotas públicas e administrativas apoiou-se nos testes de funcionalidade que simulam requisições HTTP, na execução da suíte na *CI* e nas verificações em *staging* antes da promoção à produção (Seção 4.8).

4.6 Funcionalidades implementadas

Com a refatoração estrutural concluída (Seção 4.4), o repositório da organização passou a concentrar as funcionalidades priorizadas no MoSCoW (Figura 2), sendo elas: internacionalização do acervo, verificação de colaboradores por e-mail, localização física com código legível e QR Code, proteção de formulários e tradução assistida no painel. O trabalho manteve o fluxo Gitflow. Cada frente foi validada em homologação e em *staging*; em dois marcos, promoveu-se o conjunto estável para `main` nas *releases* `5.0.0-beta` e `1.0.0`. Entre as releases, outras entregas aconteceram (limitação de taxa em rotas sensíveis, Redis para sessão e *cache*, Turnstile e ampliação de testes, enquanto as atividades da Seção 4.5) avançaram em paralelo. A atualização para Laravel 13 permanece na Seção 4.7, para não misturar o requisito de negócio com o *framework*.

4.6.1 Internacionalização e conteúdo do acervo

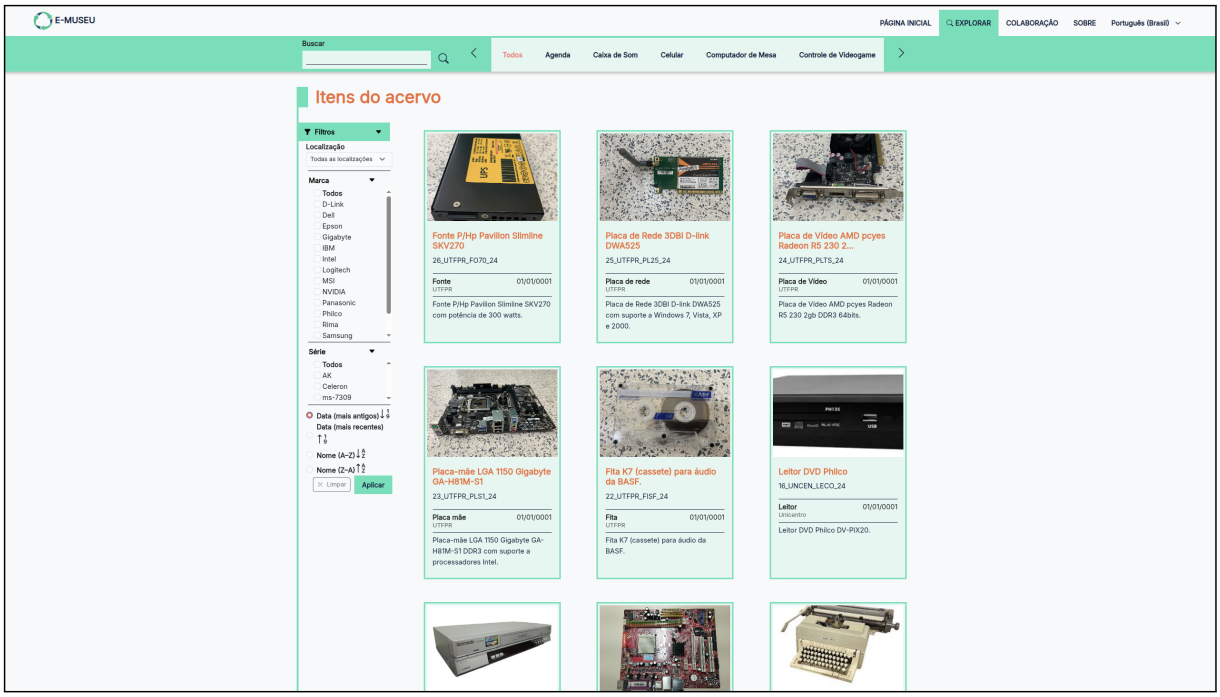
Conteúdo multilíngue no banco de dados

O suporte a múltiplos idiomas no acervo passou para homologação após a preparação da interface (Subseção 4.4.3). Até então, os textos de itens, categorias, etiquetas e *extras* ainda se encontravam em colunas monolíngues no banco de dados refatorado. Essa entrega alterou o esquema e o comportamento do catálogo e do painel porque o MoSCoW classificou a internacionalização do conteúdo como prioridade alta para o museu virtual.

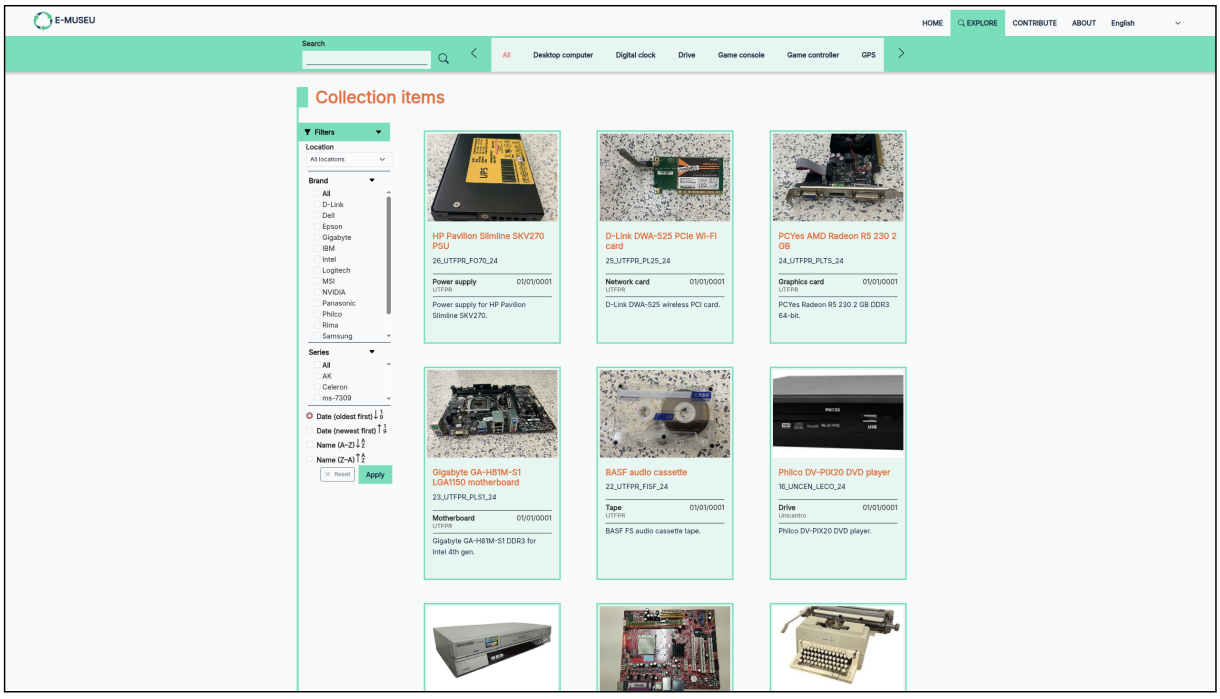
Inseriram-se tabela de idiomas suportados e tabelas de tradução para cada entidade com texto exibível ao visitante. No domínio, serviços e modelos passaram a gravar e resolver traduções com uma cadeia de *fallback* única, de modo que listagens, buscas e fichas públicas mostrem primeiro o idioma da sessão e, se faltar conteúdo, idiomas alternativos na ordem configurada. O visitante escolhe português do Brasil ou inglês por rota dedicada, persistido em sessão e aplicado por *middleware*; o painel ganhou seletor equivalente.

No cadastro e na edição administrativa, abas por idioma concentram nome, descrição, histórico e demais campos traduzíveis; a contribuição pública passou a criar linhas de tradução conforme o idioma preferido do formulário. Quando o sistema exibe texto de outro idioma que o pedido, um aviso discreto na ficha pública informa o *fallback*.

Na listagem pública *Explorar*, o efeito conjunto do esquema e da sessão de idioma aparece na prática: menus, filtros, categorias da barra superior e rótulos do painel lateral vêm dos arquivos de idioma e do *i18next* (Subseção 4.4.3); títulos, descrições curtas e nomes de categoria nos cartões são resolvidos a partir das tabelas de tradução conforme o *locale* escolhido. Os mesmos registros exibem ainda o código de identificação com sigla de localização (por exemplo, `UTFPR` ou `UNCEN`), alinhado ao domínio de localização integrado na mesma fase. A Figura 5 contrasta português do Brasil e inglês sobre o mesmo conjunto de itens em homologação.



(a) Português (Brasil)



(b) English

Figura 5 – Catálogo público *Explorar*: interface e textos do acervo conforme o idioma da sessão. Fonte: Autoria própria (ambiente de homologação).

O desenho geral (Laravel Lang, *i18next* e a separação entre interface e conteúdo) encontra-se na Seção 3.1.12. A passagem do acervo para o modelo multilíngue persistido é a base para a tradução assistida.

Tradução assistida no painel

A tradução assistida por IA no painel entrou em homologação após o acervo multilíngue da subseção anterior e as abas de tradução já utilizadas em itens, categorias, etiquetas e *extras*. A funcionalidade atende ao requisito de apoio à curadoria sem substituir revisão humana.

No formulário administrativo, a funcionalidade aparece nas abas *Universal*, português do Brasil e inglês. Assim, o curador escolhe o provedor em modo Auto ou fixa um serviço, aciona *Traduzir campos vazios* ou *Regenerar a partir das fontes* e visualiza o aviso de que a sugestão deve ser revisada antes de salvar, com menção ao limite de requisições (Figura 6).

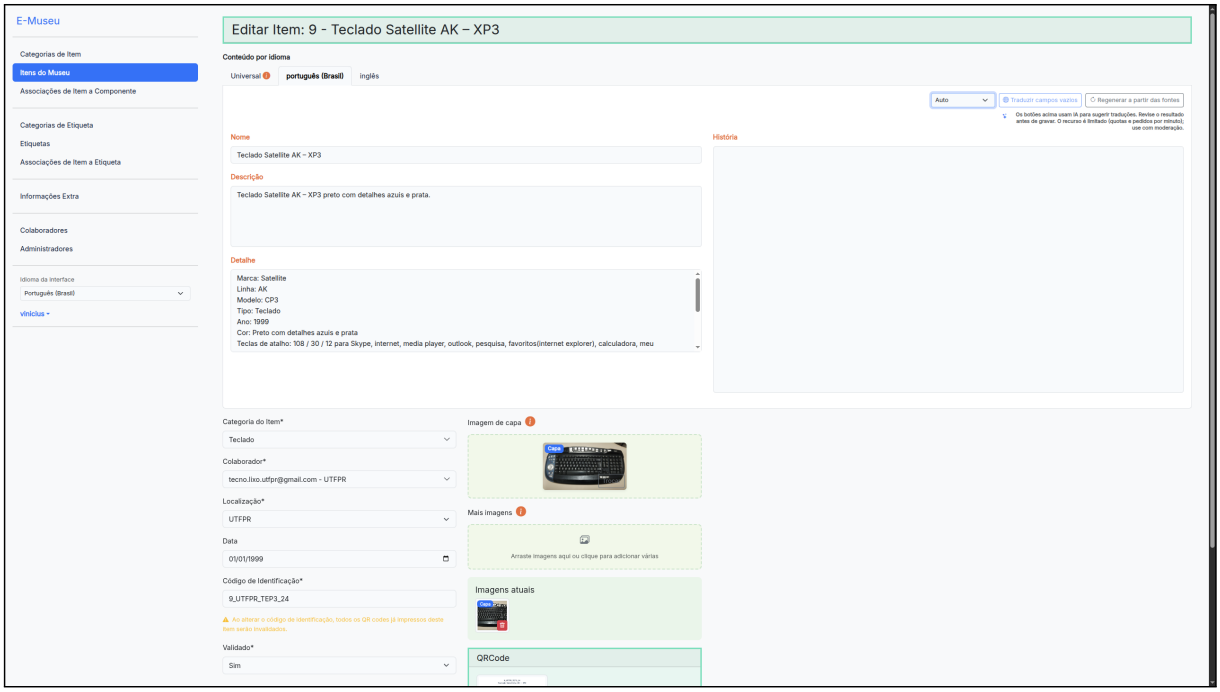


Figura 6 – Edição de item no painel: abas por idioma e ações de tradução assistida por IA. Fonte: Autoria própria (ambiente de homologação).

O *back-end* expõe rota administrativa de tradução, valida o recurso, o idioma alvo e o tamanho máximo do texto enviado e chama APIs de *chat completion* compatíveis com o formato OpenAI. Foram integrados três serviços com modelos gratuitos ou com cota sem custo para o projeto (Groq, OpenRouter e GitHub Models, Seção 3.1.18): a escolha de três provedores, e não apenas um, responde ao limite de uso típico das ofertas gratuitas; quando um serviço esgota a cota ou falha, a aplicação tenta o seguinte na ordem definida na configuração (*failover* automático no modo Auto). O administrador pode, na interface, fixar qual provedor (e, quando aplicável, qual modelo) deseja usar em vez de deixar o modo Auto percorrer a cadeia.

Os parâmetros de cada provedor (URL, chaves, modelos, *timeouts* e ordem da cadeia) ficam em configuração centralizada e variáveis de ambiente, sem classes dedicadas por fornecedor na segunda versão da entrega. O desenho permite incluir ou retirar serviços com pouca alteração de código se, no futuro, algum deixar de oferecer plano gratuito ou surgir outro substituto. As respostas válidas retornam JSON com traduções e metadados (provedor e modelo); falhas de configuração ou de cota retornam códigos HTTP adequados. O cliente administrativo

reutiliza o *jQuery* já presente no painel e exibe progresso durante a chamada. As chaves de API permanecem no servidor e há limite de requisições por administrador.

4.6.2 Contribuição pública, colaboradores e identificação física

Verificação de e-mail, Redis e mensagens transacionais

O fluxo de verificação de e-mail para colaboradores externos do catálogo público passou para homologação com o objetivo de exigir prova de posse do endereço antes de aceitar contribuições de itens ou *extras*, enviar confirmação quando o envio real estiver configurado e alinhar sessão, *cache* e limites de taxa a um armazenamento compartilhado em produção.

O colaborador solicita um código de seis dígitos por *e-mail*. O valor fica apenas na mensagem enviada, enquanto a sessão guarda o *hash* com validade de quinze minutos. As rotas JSON sob o prefixo do catálogo tratam o pedido, a confirmação, a limpeza de sessão de contribuição e a verificação de contato já cadastrado, com validação espelhando as regras do formulário. No formulário público de contribuição, o visitante informa e-mail e nome completo (quando o endereço ainda não está verificado), aciona o envio do código e confirma os seis dígitos antes de seguir o cadastro do item. A Figura 7 resume essa etapa no navegador, incluindo o desafio Turnstile quando o *antibot* está ativo (Subseção 4.6.3).

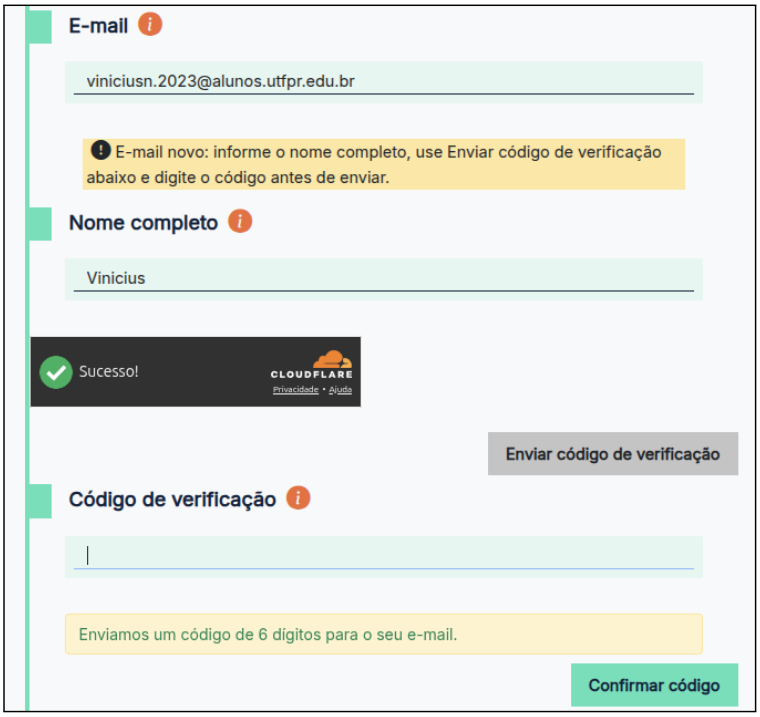


Figura 7 – Verificação de e-mail no formulário público de contribuição (e-mail, nome, envio e confirmação do código).

Fonte: Autoria própria (ambiente de homologação).

A mensagem transacional exibe o código em destaque e informa a validade de quinze minutos, conforme a Figura 8.

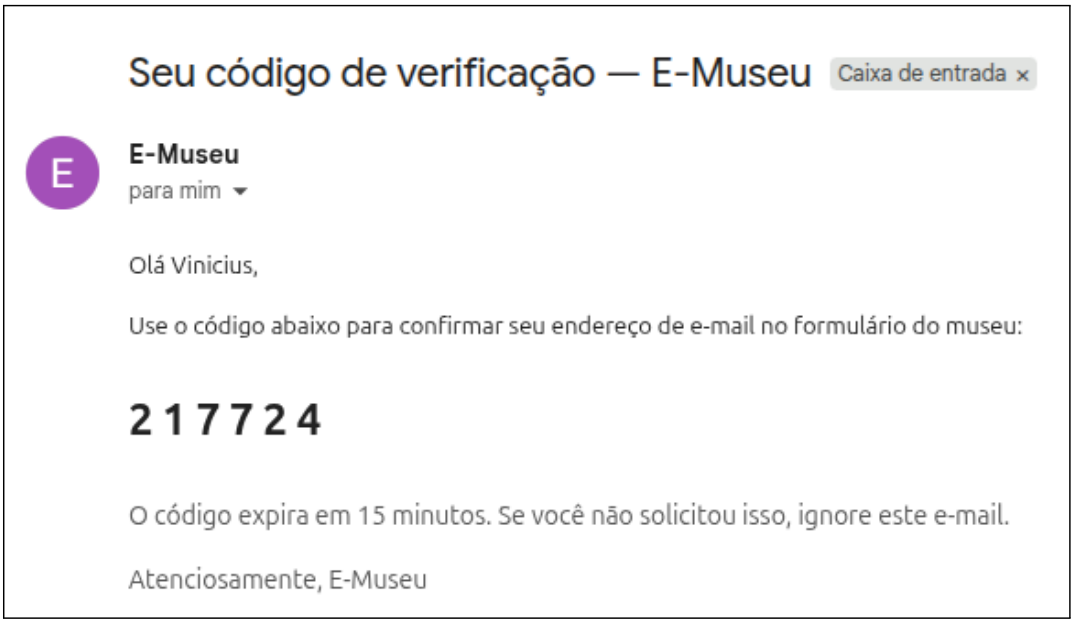


Figura 8 – E-mail transacional com código de verificação enviado ao colaborador. Fonte: Autoria própria (caixa de demonstração; código ilustrativo).

Se o servidor de correio não estiver configurado ou o envio falhar, a API devolve mensagem genérica em produção (sem expor detalhes internos) e registra o erro nos *logs*. Após a contribuição aceita, as ações de conclusão tentam enviar e-mail de “contribuição recebida” de forma *best-effort* e, em seguida, limpam a autenticação de sessão do fluxo público para não deixar o navegador em estado inconsistente.

No painel, os colaboradores ganharam campo opcional de confirmação manual de e-mail e indicadores na listagem. Para ambientes reais, sessão e *cache* passaram a usar Redis no exemplo de configuração e nos *compose* locais, de modo que códigos pendentes e contadores de *throttle* sobrevivam a reinícios e a mais de um processo PHP. Em *staging* e produção institucionais o Redis é externo ao *compose* da aplicação. Os testes automatizados continuam com driver em memória para não depender do serviço. O envio transacional por SMTP institucional está descrito no Capítulo 3.

Localização, código de identificação e QR Code

O domínio de localização e o código de identificação física dos itens entraram em homologação posteriormente. A demanda alinhava-se ao objetivo de etiquetas para as peças no museu. Cada item passou a referenciar um registro de localização (códigos normalizados, rótulos em português e em inglês), com padrão institucional `UTFPR` e *fallback* para indefinido quando aplicável.

O sistema gera e atualiza um código de identificação legível (identificador do item, código da localização, trecho do título e ano) e expõe rota pública que redireciona para a ficha do item quando o código é válido. No painel, ações permitem gerar, copiar, imprimir ou remover o QR Code cuja URL aponta para essa rota. A imagem do QR integra-se ao esquema de múltiplas imagens já adotado (Subseção 4.4.2) e a etiqueta impressa ou exportada reúne o código legível,

um trecho do nome do item e o símbolo QR que aponta para a rota pública. A Figura 9 mostra um caso real (teclado cadastrado no acervo com localização na `UTFPR`).

Figura 9 – Exemplo de etiqueta com código de identificação e QR Code gerados pelo painel. Fonte: Autoria própria.

Os filtros do catálogo público e os índices administrativos passaram a considerar a localização na busca e na ordenação. Os serviços de conclusão de contribuição e de e-mail de recebimento foram reorganizados nesta mesma entrega para concentrar as responsabilidades, sem alterar a regra de verificação de e-mail descrita acima.

4.6.3 Segurança e abuso em formulários

Esta subseção trata de camadas complementares, como a verificação humana nos envios sensíveis e a limitação da frequência de requisições. Ambas respondem a formulários públicos e ao *login* administrativo, em linha com a priorização MoSCoW e com o que se encontra no Capítulo 3 (Turnstile e envio de e-mail).

Cloudflare Turnstile

A proteção *antibot* com Cloudflare Turnstile entrou em homologação porque nos objetivos do trabalho constava a inclusão do mecanismo no estilo reCAPTCHA; na implementação adotou-se o Turnstile, conforme o Capítulo 3.

A configuração permite desativar o mecanismo em desenvolvimento e na *Continuous Integration/Continuous Delivery* (CI/CD), no qual o verificador permanece inativo. Com o driver ativo, *middleware* valida o token no *back-end* por chamada à API da Cloudflare. Se ocorrer falha de rede ou token inválido, é gerado erro de validação sem reexibir o segredo na sessão. O widget reutilizável aparece no *login* administrativo e no pedido AJAX de código de verificação por e-mail. O JavaScript do catálogo só dispara o envio do código após o desafio concluído e

reinicia o widget após cada tentativa. A Figura 10 ilustra o estado exibido ao usuário após a verificação bem-sucedida no navegador.

Figura 10 – Widget Cloudflare Turnstile após verificação concluída (*Sucesso!*).
Fonte: Autoria própria.

Em *staging* e produção, os exemplos de variáveis de ambiente para *Coolify* documentam o par de chaves necessário.

Limitação de taxa e proteção contra abuso de requisições

Limitadores nomeados passaram a ser registrados na inicialização da aplicação e o *middleware* `throttle` passou a cobrir rotas do catálogo, rotas JSON leves, painel, *proxy* de armazenamento e *login*. A intenção foi reduzir abuso de busca, de contribuição e de tentativas repetidas de autenticação sem depender só do captcha.

Com Redis como *cache* em ambientes reais (Subseção 4.6.2), os contadores dos limitadores personalizados passaram a ser compartilhados entre processos. Na verificação de e-mail, limites específicos cobrem o pedido de código, a confirmação, a limpeza de sessão e a consulta de contato; o *login* administrativo e o pedido de código combinam limitação de taxa com Turnstile (Subseção 4.6.3). Na atualização para Laravel 13 (Seção 4.7), acrescentou-se um limite adicional por IP na rota de armazenamento da contribuição pública de itens. Nos testes, o comportamento dos limitadores é verificado sem Redis obrigatório, em linha com a configuração de testes descrita na Seção 4.5.

4.6.4 Promoção das entregas para `main`

***Release* `5.0.0-beta`**

Homologação foi promovida para `main` na *release* `5.0.0-beta`. Esta versão consolida em `main`, após a `4.0.0-beta` de março, o acervo multilíngue, a verificação de e-mail com Redis, a localização com código e QR Code, os limitadores de taxa da primeira quinzena de abril e as melhorias correlatas de formulário e catálogo validadas na mesma semana. Ainda não alcançavam `main` o Turnstile, a tradução assistida por IA, a ampla suíte de testes nem a atualização para Laravel 13, que permaneceram em homologação até a promoção seguinte.

***Release* `1.0.0`**

Na *release* `1.0.0`, homologação foi promovida para `main`. O marco encerra o ciclo funcional descrito nesta seção: acervo e painel multilíngues, colaboradores verificados por e-

mail, identificação física com QR Code, Turnstile, tradução assistida no painel, limitação de taxa reforçada e demais entregas de abril já validadas (incluindo testes automatizados da Seção 4.5 e a atualização de *framework* da Seção 4.7). Como explicado no início deste capítulo, o rótulo `1.0.0` sinaliza a primeira linha estável para produção institucional, após a sequência de versões `-beta` de homologação.

4.6.5 Itens do escopo não implementados neste ciclo

A priorização MoSCoW (Figura 2) e o tempo disponível deixaram de fora deste ciclo requisitos classificados como *Could-have* ou *Won't-have*. Não foram implementados, entre outros, os relatórios de acesso; os logs de alterações por administradores; o backup automático em produção; a restilização ampla da interface, além de alguns ajustes; e fluxos herdados do legado já descontinuados na refatoração (registro público genérico, assistente "Ada"). Outras ideias do levantamento inicial que não entraram no MoSCoW como *Must* ou *Should* ficarão para trabalhos futuros (Seção 6.4).

Quanto ao cadastro de mídia, múltiplas fotos por item e o armazenamento em objeto foram realizados (Subseção 4.4.2), atendendo ao objetivo de ampliar o *upload* de imagens previsto no MoSCoW.

4.7 Atualização tecnológica da aplicação

Posteriormente, concluiu-se em homologação a atualização do E-Museu para Laravel 13. O trabalho aconteceu depois das funcionalidades centrais (Seção 4.6) e em paralelo à ampliação da suíte de testes (Seção 4.5), com validação na *Continuous Integration/Continuous Delivery* (CI/CD) antes da promoção à `1.0.0` (Subseção 4.6.4). Não se trata da mesma atualização de *framework* da infraestrutura inicial: a pilha já havia passado a *PHP* 8.5 e Laravel 12 (Seção 4.3.1); em seguida, o repositório da organização alinhou-se às convenções da versão 12 (Subseção 4.4.1). Esta seção descreve a segunda atualização, motivada por compatibilidade de dependências, mudanças do guia oficial de migração e endurecimento do código já multilíngue e exposto ao catálogo público.

4.7.1 Dependências e convenções do Laravel 13

O *composer.json* passou a exigir Laravel na série 13, com ajustes correlatos de dependências para que `composer install` no Docker e na pipeline reproduzisse o mesmo conjunto de versões. O projeto já apontava para *PHP* 8.5, dentro do mínimo exigido pelo *framework*.

Na configuração inicial do *framework*, o *middleware* de proteção contra falsificação de requisição no grupo `web` passou a usar a classe prevista pelo Laravel 13, com o mesmo alinhamento para requisições *stateful* (com sessão e *cookie* no navegador, como no painel administrativo). Adotou-se padrão mais restritivo de desserialização no *cache*. As alterações não mudam regras de negócio do museu; atualizam o esqueleto do *framework* para a versão vigente na data da entrega estável.

4.7.2 Análise estática e SQL dinâmico do acervo

Com o Laravel 13, os *stubs* usados pelo *PHPStan* via *Larastan* passaram a tratar o primeiro argumento de consultas SQL dinâmicas como *literal-string*. O catálogo multilíngue (Subseção 4.6.1) já montava consultas dinâmicas seguras (ordenação com funções do MySQL, subconsultas de tradução, *fallback* de *locale*), porém o analisador estático passou a sinalizar dezenas de erros de tipo onde a concatenação é feita no código da aplicação e não como literal fixa.

Para concentrar a exceção documentada, introduziu-se um apoio que delega nas APIs *raw* do Laravel e reúne os pontos em que o SQL é construído de forma controlada (serviços de catálogo e taxonomia, consultas de índices administrativos e públicos, modelos com relações traduzidas). O SQL gerado em tempo de execução permanece o mesmo; o ganho é manter o nível elevado de *PHPStan* definido na Seção 4.3.2 sem desligar a verificação nas áreas de listagem e busca do acervo.

4.7.3 Endurecimento da contribuição pública e do código

Na mesma entrega, reforçou-se o fluxo de contribuição pública de itens, já descrito nas Seções 4.6.2 e 4.6.2. A rota de armazenamento ganhou limitador por IP, além da limitação geral do catálogo (Subseção 4.6.3). A validação de componentes vinculados exige que o item referenciado exista e esteja validado no acervo, com checagem duplicada no serviço para evitar adulteração do formulário. Erros de estado inesperado na contribuição deixam de responder como falha genérica do servidor e passam a mensagens de validação traduzidas. A geração de QR Code após o cadastro passou a ocorrer depois do *commit* da transação, sem manter o banco aberto durante chamadas externas.

Padronizou-se tipagem estrita nos arquivos *PHP* principais do projeto. Pequenos ajustes no painel (tratamento de falha ao regenerar QR Code, documentação interna de travas de edição) e atualização dos documentos de projeto fecharam a etapa. Com a conclusão em homologação, a aplicação ficou na linha Laravel 13 validada pelos testes e pela *Continuous Integration/Continuous Delivery* (CI/CD) antes da *release* `1.0.0`.

Com a refatoração (Seção 4.4), os testes (Seção 4.5) e as funcionalidades (Seção 4.6), esta entrega fecha a fase de alterações diretas no E-Museu no repositório. A partir daí, o foco passou a publicar e manter o sistema na UTFPR com *Coolify*: *deploy*, variáveis de ambiente e serviços institucionais (Redis, SMTP, domínios), documentação de implementação no repositório e pacote de continuidade entregue à orientação, até homologação e produção reproduzirem o que já valia localmente e na VPS de ensaio (Seção 4.8).

4.8 Implantação, validação e entrega institucional

A publicação do E-Museu nos domínios da UTFPR concentrou-se quando o repositório institucional já reunia o modelo de dados, as funcionalidades centrais, os testes automatizados e a topologia Docker validada no ambiente local e na VPS de ensaio (Seção 4.3). Até então, a maior parte do código havia sido desenvolvida fora do campus. A pilha ensaiada com *Coolify* foi replicada no campus para corrigir variáveis e redes, iterar a automação de *deploy* e montar um arquivo offline para a continuidade do projeto. O registro diário dessas sessões na ferramenta é demonstrado na Seção 3.1.19.

4.8.1 Organização no *Coolify* e acesso ao painel

No servidor da UTFPR, o *Coolify* agrupa o E-Museu no projeto `e-museu` com dois ambientes lógicos, *staging* e *production*, cada um com recursos próprios e configurações independentes. Essa separação espelha o fluxo Gitflow: a ramificação `develop` alimenta homologação em https://staging.e-museu.gp.utfpr.edu.br/; a `main`, produção em https://e-museu.gp.utfpr.edu.br/. As imagens e versões dos serviços seguem o descrito na Seção 3.1.4.

No ambiente *production*, os recursos publicados dividem-se em aplicação principal (`main-production`), bancos e *cache* (`mysql-production`, `redis-production`) e armazenamento de objetos (`storage-minio-production`), todos sob o mesmo servidor gerenciado pelo *Coolify* (Figura 11). O ambiente de homologação repete a mesma composição com nomes e variáveis de *staging*, o que permitiu validar as migrações, o *upload* de imagens e a internacionalização antes da promoção à produção.

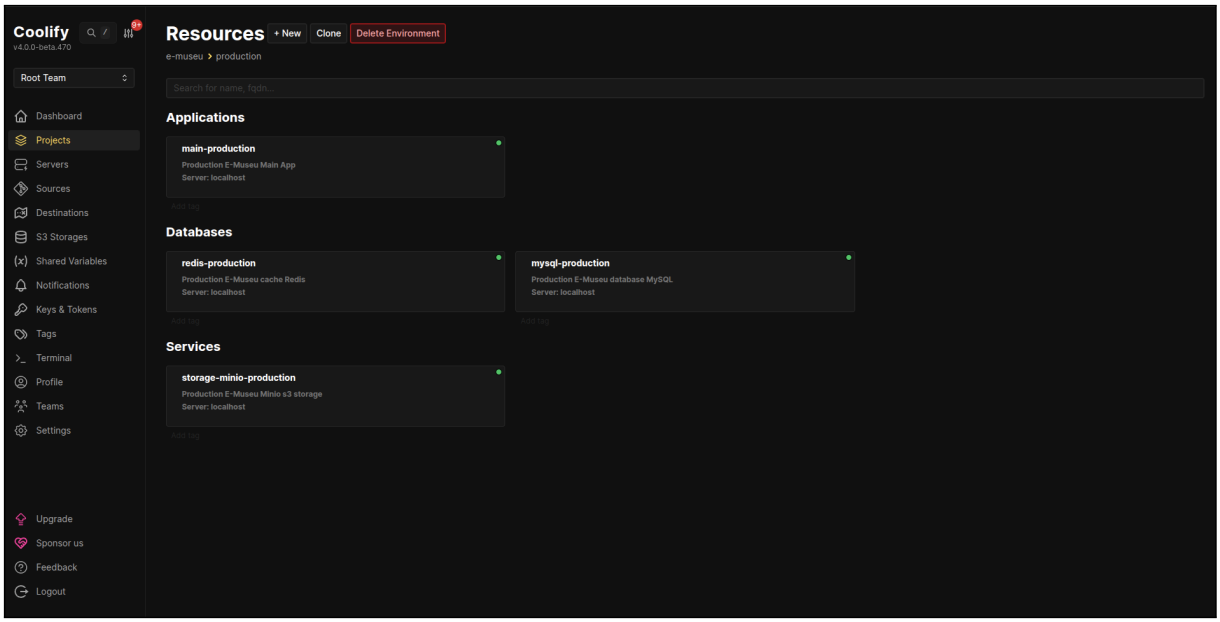


Figura 11 – Recursos do ambiente *production* no *Coolify* (aplicação, *MySQL*, *Redis* e *MinIO*). Fonte: Autoria própria (painel institucional, maio/2026). Nota: em *staging*, a estrutura é equivalente, com sufixo e domínio de homologação.

Inicialmente, o painel deveria ser usado apenas no campus, por receio de expor a administração da infraestrutura e por não haver, no começo, subdomínio dedicado ao *Coolify*. Isso exigiu a criação de serviços, o ajuste das variáveis de ambiente e o disparo de *deploys* manualmente.

4.8.2 Implantação presencial no campus

As primeiras sessões no servidor da UTFPR aconteceram quando a aplicação, o banco e as variáveis de ambiente foram alinhados ao que já executava na VPS. A máquina virtual da universidade tinha, na época, apenas um núcleo de CPU e 2 GB de RAM. Ao publicar a nova pilha com *Coolify* (aplicação Laravel, *Nginx*, *MySQL*, *Redis* e *MinIO* em *staging* e produção), os *deploys* passaram a falhar por falta de memória e de processamento. Assim, o servidor foi modificado para dois núcleos e 6 GB de RAM, de modo a sustentar os dois ambientes e permitir a continuidade da operação (Figura 26 - Capítulo 5). Na mesma fase, surgiram problemas típicos de rede Docker no *Coolify* (por exemplo, a opção *Connect To Predefined Network*, que impedia o banco de subir até ser desativada) e tentativas de *healthcheck* que, em um caso, passaram a responder com erro 404 e foram revertidas.

Posteriormente, *staging* e produção passaram a responder nos domínios institucionais com a pilha completa, mas os *webhooks* nativos do *Coolify* ainda não disparavam *deploy* de forma confiável após *commits* no *GitHub*. A validação manual cobriu login administrativo, catálogo público, envio de imagens, fluxo de contribuição com verificação por e-mail e, quando configurado, envio de mensagens transacionais. As falhas encontradas em homologação vol-

tavam ao desenvolvimento local, a *pull requests* e aos testes da Seção 4.5, antes de nova promoção em `main`.

4.8.3 Automação de *deploy*: tentativas e solução final

A automação de *deploy* na UTFPR passou por três alternativas até a configuração estável. O ponto de partida comum era o painel do *Coolify* inacessível fora do campus, o que dificultava usar de imediato os *webhooks*, que a própria plataforma oferece para ligar o repositório Git à homologação e à produção. Na primeira alternativa, implementou-se um script agendado no *Coolify* (`coolify-auto-deploy.sh`), executado por tarefa periódica no container da aplicação: o script consultava o repositório remoto, comparava o *hash* do último *commit* com um arquivo de estado e, havendo mudança, acionava a API de *deploy* do *Coolify* com token de autorização. A solução funcionou; porém, consumia processamento de forma contínua.

Na segunda alternativa: um *endpoint* na aplicação Laravel, disparado por *GitHub Actions* a cada alteração no repositório, com *pipeline* dedicada de implantação. O intuito era reduzir o custo do script periódico e alinhar o gatilho ao fluxo já usado no *GitHub*. A implementação começou na UTFPR e foi concluída; o *deploy* manual pelo novo *endpoint* já funcionava, mas a automação completa permaneceu instável por falhas de autenticação entre o *GitHub Actions* e o ambiente institucional (token que respondia em teste com `curl`, porém falhava na *action*).

Na solução definitiva, o *Coolify* foi exposto em subdomínio dedicado (https://coolify.e-museu.gp.utfpr.edu.br/), em lugar de manter o painel só acessível na rede local do servidor. Com o painel disponível pela web, deixou de ser necessário o *endpoint* na aplicação, criado justamente para contornar essa indisponibilidade. O repositório reverteu a rota de *deploy* e a *action* associada, habilitou a autenticação em dois fatores no painel e configurou os *webhooks* do *GitHub* para a URL que o *Coolify* disponibiliza. Assim, para cada *push*, o *Coolify* recebe o evento e dispara o *deploy* no ambiente correto (*staging* para `develop`, produção para `main`), sem lógica extra na aplicação Laravel.

Após revisão final, o fluxo por *webhooks*, descrito acima, passou a ser o padrão estável, dispensando script periódico e *endpoint* intermediário. Assim, o trabalho ficou concentrado em carga de dados, validação de funcionalidades ou ajustes que ainda exigem acesso local ao servidor, e não em cada integração rotineira em `develop`.

4.8.4 Documentação no repositório

Em paralelo, a documentação no *GitHub* foi ampliada. O `README` do projeto, os arquivos em `docs/deploy/`, o `.env.example` e a wiki do repositório (fluxo Gitflow no perfil *full*, Seção 4.2) passaram a descrever passo a passo a implantação com *Coolify*, a separação entre *staging* e produção e o papel de cada serviço. Parte desse material foi revisada a cada tentativa

de automação de *deploy*, de modo que o texto acompanhasse o que de fato estava em produção. As credenciais, tokens e exports completos do painel não foram incluídos no repositório público e ficaram no pacote de continuidade descrito na subseção seguinte.

4.8.5 Pacote de continuidade: backups e pen drive

Além do código versionado, montou-se um pacote de continuidade para recuperação e reconfiguração caso o servidor, o *Coolify* ou o acesso ao *GitHub* falhem no futuro. O trabalho avançou em conjunto com os ajustes no painel e culminou na aquisição de um pen drive dedicado ao E-Museu, com cópia adicional em arquivo protegido por senha em computador específico na UTFPR. O objetivo foi reunir, em um só lugar, evidências e artefatos práticos para a orientação e para a evolução do E-Museu.

O conteúdo foi organizado em três pastas de primeiro nível (`data`, `projects` e `txt`), conforme a Figura 12. A estrutura sugere um guia de consulta offline: dados do acervo, cópia do software e texto de apoio à infraestrutura.

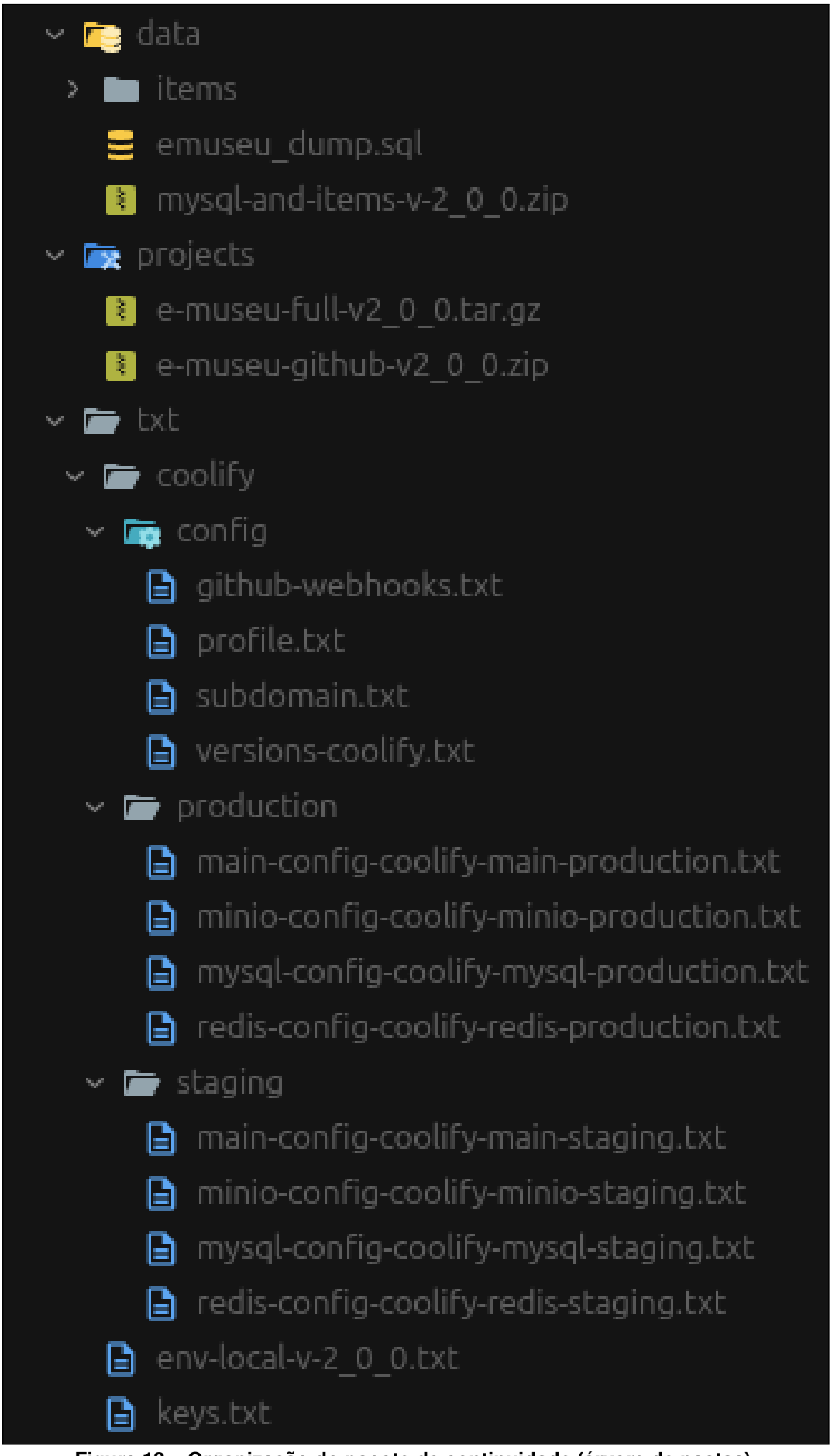


Figura 12 – Organização do pacote de continuidade (árvore de pastas).
Fonte: Autoria própria.

A pasta `data` concentra o estado do acervo na data da montagem do pacote. O arquivo `emuseu_dump.sql` permite restaurar ou inspecionar o banco relacional sem o servidor institucional. O diretório `items/` guarda cópias das imagens dos itens (subpastas por identificador do item). O arquivo `mysql-and-items-v-2_0_0.zip` reúne, em um único compactado, o dump e as imagens para quem preferir importar tudo de uma vez.

A pasta `projects` traz duas variantes do E-Museu na versão `2.0.0`. O arquivo `e-museu-github-v2_0_0.zip` corresponde ao que estava no repositório institucional no momento da entrega (código alinhado ao *GitHub*, sem credenciais sensíveis embutidas). O `e-museu-full-v2_0_0.tar.gz` acrescenta dependências (`vendor/`, `node_modules/` e demais pacotes externos) já baixados e o `.env` com as credenciais usadas naquele instante: serve para instalar em máquina sem rede na recuperação e, se algum pacote externo deixar de estar disponível ou mudar de forma incompatível, oferece uma cópia fixa do conjunto que funcionava com aquele código, incluindo variáveis de ambiente prontas para subir a aplicação.

A pasta `txt` documenta a infraestrutura e as integrações. Em `env-local-v-2_0_0.txt` há referência das variáveis de ambiente locais. O arquivo `keys.txt` indica quais contas e chaves o sistema usa (organização no *GitHub*, APIs de tradução, entre outras), para renovação de acessos sem depender da memória de quem mantiver o sistema no futuro. O subdiretório `coolify/` espelha o que foi configurado no painel. Em `config/`, há notas sobre perfil do servidor, subdomínio do *Coolify*, *webhooks* do *GitHub* e versões da plataforma. Em `staging/` e `production/`, cada serviço da Figura 11 (aplicação principal, *MySQL*, *Redis* e *MinIO*) ganhou um export textual com o bloco de configuração correspondente no *Coolify*. A ideia foi permitir copiar e colar dezenas de parâmetros já preenchidos (variáveis de ambiente, domínios, imagens *Docker*, redes e demais campos do painel) em vez de redigitar tudo à mão, recriando homologação e produção com os mesmos nomes de recurso usados na UTFPR.

Depois da entrega do TCC, o repositório `e-museu-utfpr-gp/e-museu` permanece a referência principal para código e novos *deploys*; o pacote complementa essa fonte com dados, configurações e cópias offline descritas acima. O esforço de organizá-lo, somado às horas de implantação e à documentação no *GitHub* (Subseção 4.8.4) faz parte da dimensão operacional do trabalho.

4.8.6 Validação final e entrega

Após cada entrega relevante em homologação, o sistema era validado em *staging* antes da promoção à produção em `main`. Com os *webhooks* estáveis, o encerramento consolidou a entrega nos domínios institucionais, o painel em subdomínio dedicado e o pacote de continuidade para a orientação. A síntese dos objetivos ligados a ambientes e operação aparece na Tabela 2 e na Seção 5.5 do Capítulo 5.

As etapas deste capítulo descreveram o processo de desenvolvimento (requisitos, versionamento, refatoração, testes, funcionalidades e implantação). A consolidação dos resultados em relação aos objetivos, à evolução do modelo de dados, aos indicadores quantitativos do trabalho e à operação em produção está no Capítulo 5, em especial na Tabela 2, na Seção 5.4 e na Seção 5.2.

5 RESULTADOS

Este capítulo apresenta o que foi alcançado após as etapas de desenvolvimento descritas no Capítulo 4. O foco não é repetir o cronograma de implementação, mas apresentar a evolução do modelo de dados e do produto em relação à versão legada, capturas da interface entregue e da raiz do repositório no *GitHub*, indicadores verificáveis do projeto (incluindo a atividade no repositório institucional, o tempo dedicado e os gastos diretos do ciclo) e a situação da entrega operacional.

5.1 Síntese em relação aos objetivos

Ao término do ciclo documentado nesta monografia, consolidaram-se os principais resultados do trabalho em relação ao objetivo geral (Seção 1.1.1) e aos objetivos específicos (Seção 1.1.2). A Tabela 2 resume, de forma sintética, o atendimento a cada meta e indica onde a evidência correspondente é detalhada no texto. Evoluções que extrapolam o escopo deste ciclo são discutidas nas limitações e nos trabalhos futuros (Seções 6.3 e 6.4).

Tabela 2 – Síntese dos objetivos específicos e dos resultados alcançados.

Objetivo específico	Principal evidência no trabalho	Situação
Versionamento e padrões no GitHub	Seções 4.2 e 4.1	Atendido
Ambientes de desenvolvimento, teste e produção	Seções 4.3, 4.8 e 5.5	Atendido
Refatoração do código existente	Seções 4.4 e 5.2	Atendido
Testes automatizados	Seção 4.5; Seção 5.4	Atendido
Novas funcionalidades do E-Museu	Seção 4.6	Atendido
Padrões de projeto e boas práticas	Seções 4.4 e 4.6; Capítulo 2	Atendido

Fonte: Autoria própria.

5.2 Evolução do modelo de dados

As Figuras 13 a 19 apresentam o esquema relacional da versão entregue em recortes legíveis, com destaque para alterações em relação ao legado descrito por Gonzaga (2024) e às migrações documentadas no Capítulo 4. A Figura 13 define a convenção visual; as figuras seguintes agrupam tabelas por domínio (catálogo em dois recortes, taxonomia, demais entidades e estruturas removidas); a última consolida a visão completa.

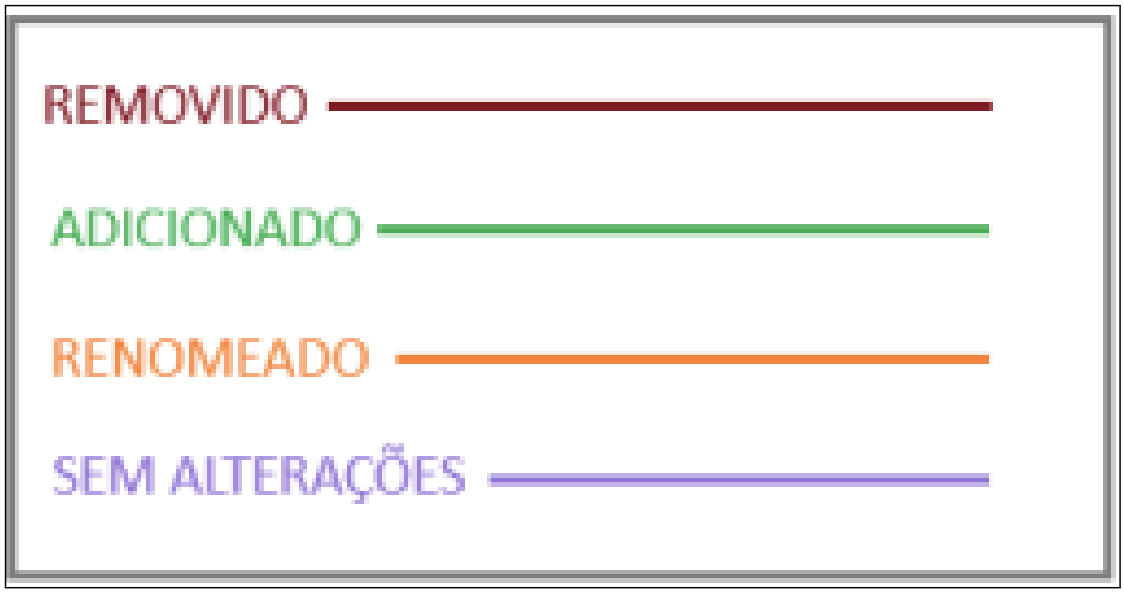


Figura 13 – Legenda do modelo relacional refatorado.

Fonte: Autoria própria.

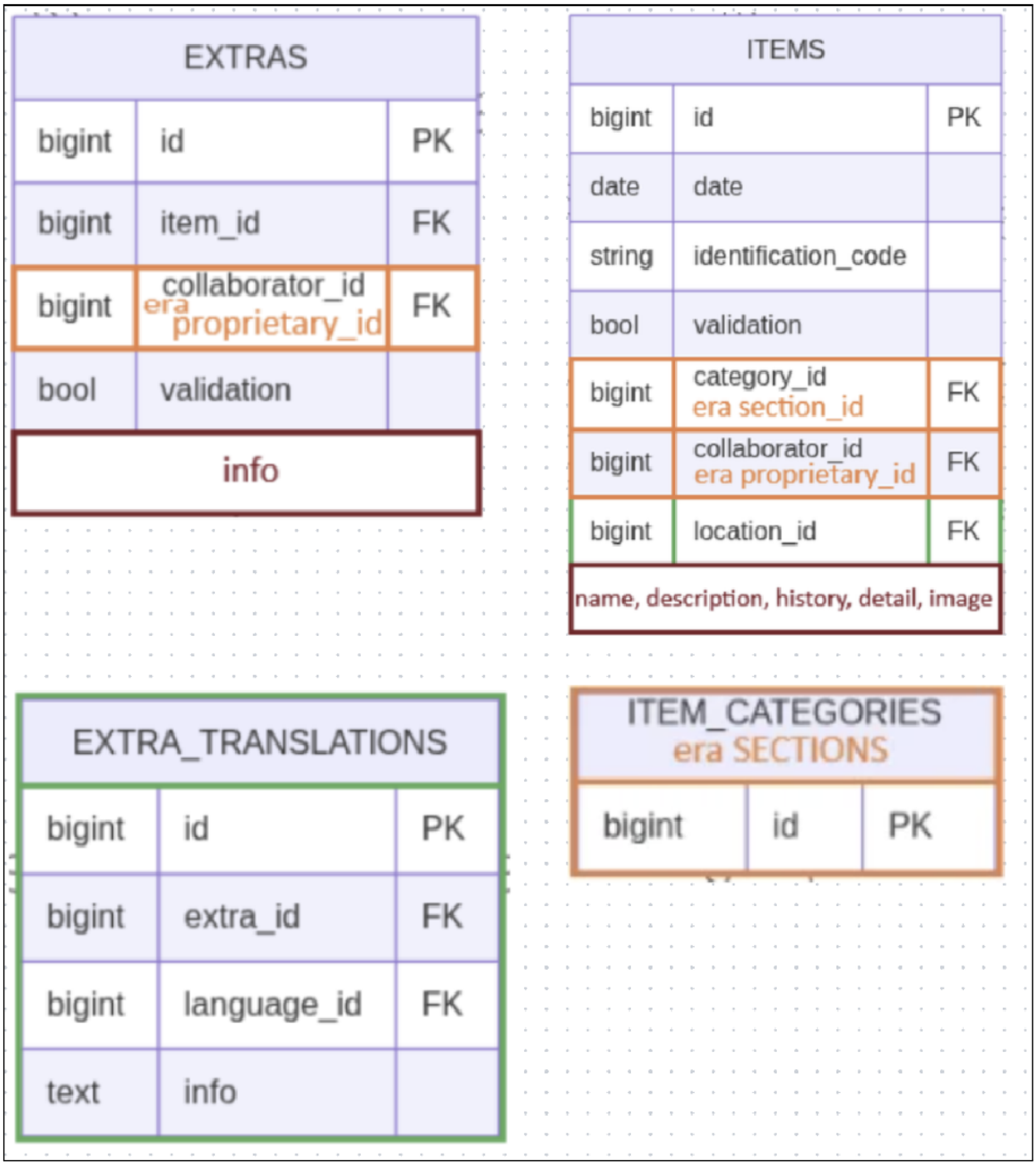


Figura 14 – Tabelas do domínio de catálogo (parte 1): `items`, `extras`, `item_categories` e traduções associadas.

Fonte: Autoria própria.

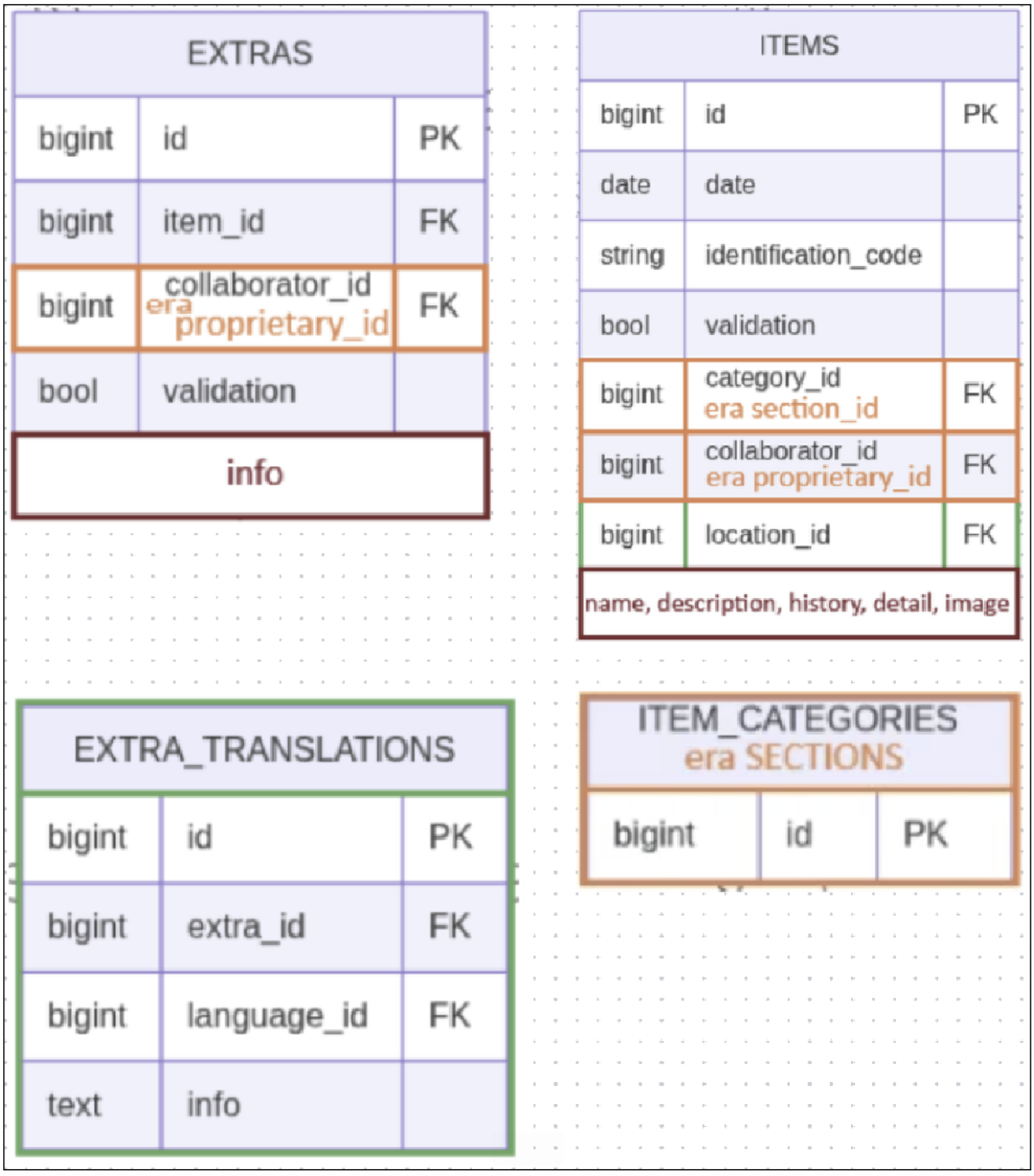


Figura 15 – Tabelas do domínio de catálogo (parte 2; continuação): `item_translations`, `item_images`, `item_component` e `item_tag`.

Fonte: Autoria própria.

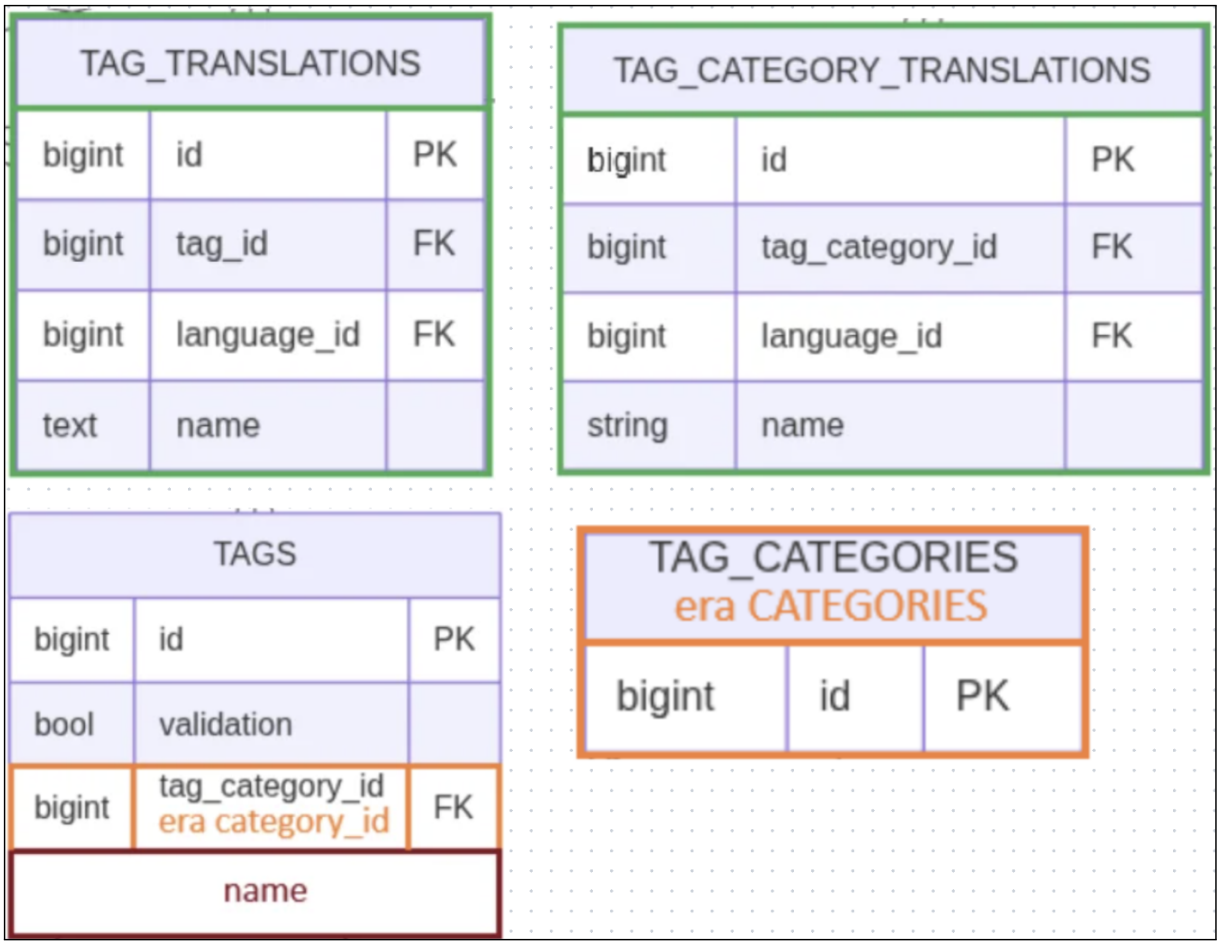


Figura 16 – Tabelas de taxonomia (etiquetas, categorias de etiqueta e vínculos com itens).

Fonte: Autoria própria.

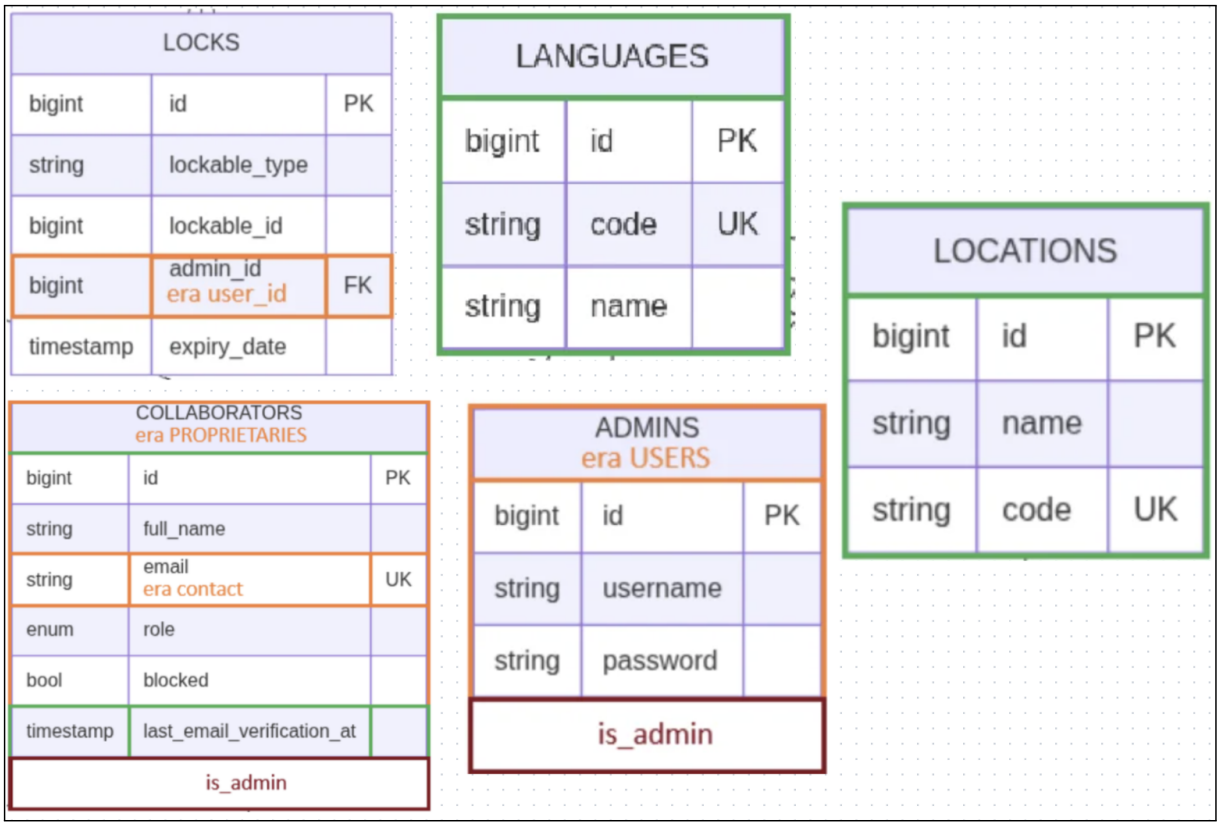


Figura 17 – Demais entidades (colaboradores, administradores, localizações e tabelas auxiliares).
Fonte: Autoria própria.

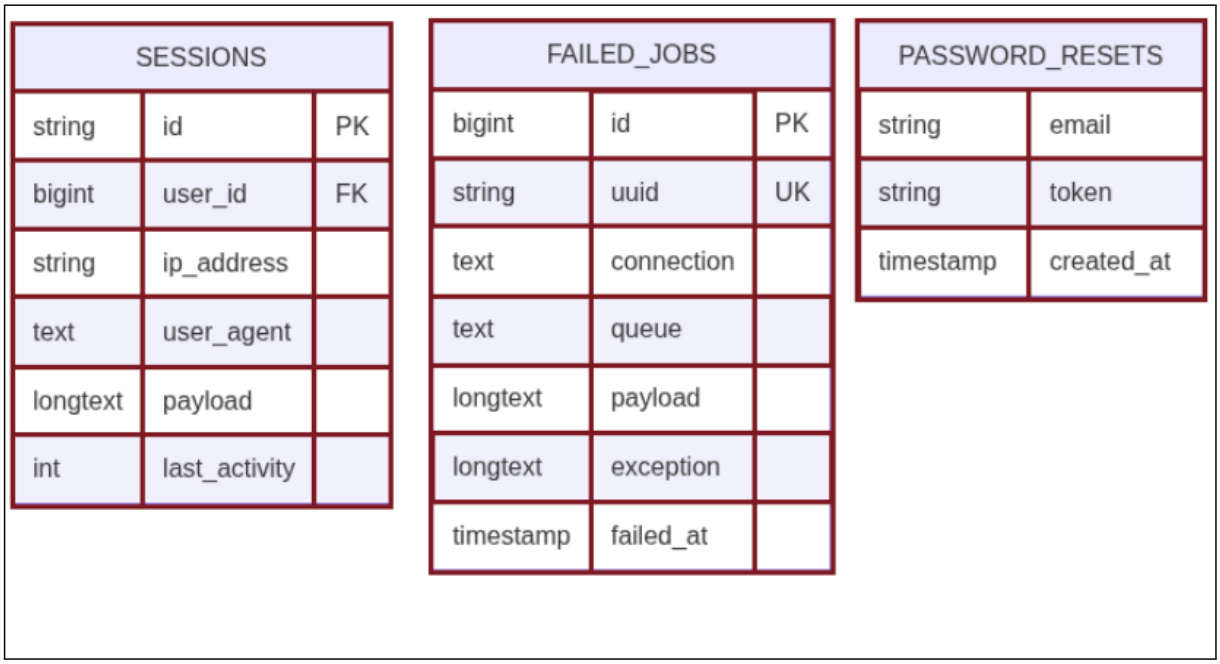


Figura 18 – Estruturas removidas ou substituídas em relação ao legado (riscadas na legenda).
Fonte: Autoria própria.

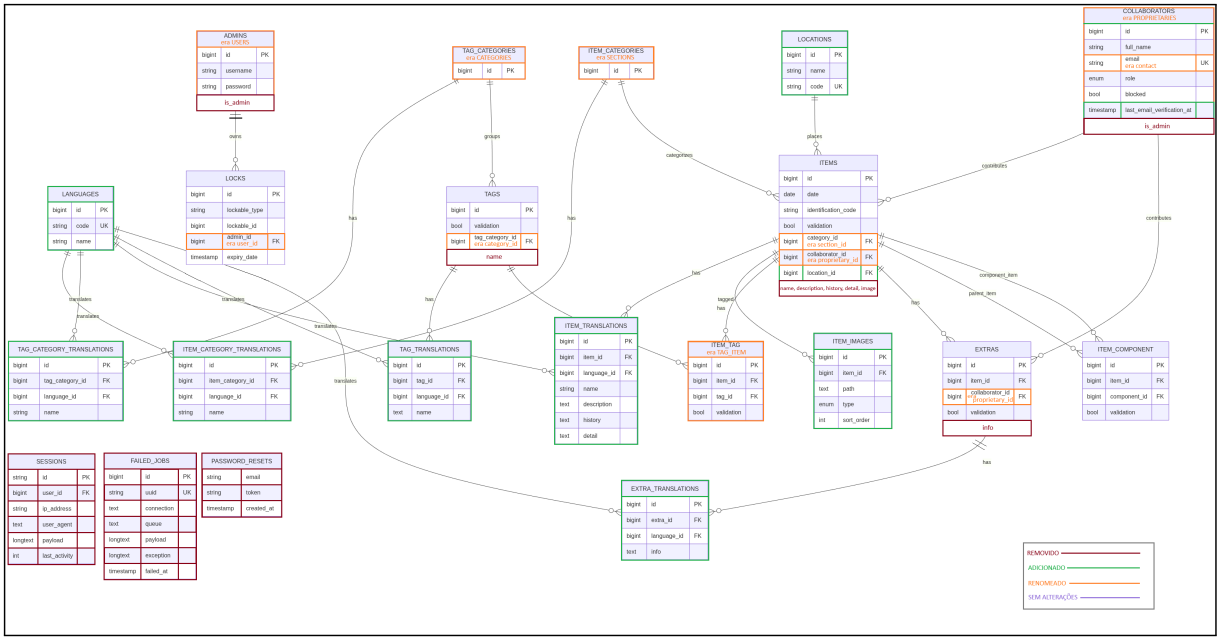


Figura 19 – Modelo relacional do E-Museu refatorado (visão consolidada).

Fonte: Autoria própria. Legenda: removido (riscado); novo ou renomeado (verde); previsto para i18n ou `locations` (laranja); suporte a traduções no painel (roxo).

A reorganização por domínios e a adoção de armazenamento por proxy (Subseção 4.4.1) antecederam a *release* `3.0.0-beta`. Posteriormente, o pacote de renomeações (Subseção 4.4.2) alinhou o vocabulário físico do banco ao das pastas de aplicação: `users` para `admins`, `sections` para `item_categories`, `proprietaries` para `collaborators`, `categories` para `tag_categories`, além da tabela `item_images` e da associação `item_tag`. A *release* `4.0.0-beta` promoveu esse conjunto a `main`.

Entregas posteriores (internacionalização completa do acervo, domínio `Location` e atualização para Laravel 13, Seção 4.7) acrescentaram ou consolidaram tabelas e colunas indicadas em laranja e roxo na Figura 19; o detalhamento cronológico permanece no Capítulo 4, enquanto esta seção fixa o estado final do modelo para leitura comparativa.

5.3 Evolução do software em relação ao legado

Em relação à versão descrita por Gonzaga (2024), o E-Museu refatorado passou a operar com ambientes distintos de desenvolvimento local, *staging* e produção; pilha atualizada (*PHP* 8.5, Laravel 13); organização por domínios sob `app/Models/`, `app/Services/` e `app/Http/Controllers/` (por exemplo `Catalog/`, `Taxonomy/` e `Identity/`); armazenamento de mídia em objeto (MinIO) com URLs servidas por proxy; dezenas de testes automatizados de fluxo e integração; *pipeline* de qualidade com Pint, PHPStan e testes no *GitHub Actions*; e documentação de fluxo de desenvolvimento no perfil *full* do Gitflow (Novacoski, Vinicius Ferreira, 2026).

A base legada concentrava regras em controladores extensos, vocabulário misto (`users`, `proprietaries`, `sections`) e pouca cobertura de testes. A versão atual distribuiu as regras entre serviços, *Actions* e *Form Requests*, expôs *endpoints* JSON por domínio e removeu fluxos não utilizados (registro público genérico, assistente “Ada”). Funcionalidades priorizadas no MoSCoW (internacionalização, múltiplas imagens, QR Code, Turnstile, tradução assistida no painel) integram-se a esse núcleo; requisitos de menor prioridade permanecem fora deste ciclo (Seção 4.6.5).

5.3.1 Raiz do repositório no GitHub

Além das mudanças no código sob `app/`, o trabalho alterou a forma como o projeto se apresenta no *GitHub*: pastas de infraestrutura, automação e qualidade passaram a conviver com a estrutura Laravel na raiz do repositório. As Figuras 20, 21 e 22 reproduzem a página principal (*Code*) de cada repositório para comparar o legado público de Gonzaga (2024) com a versão institucional entregue neste ciclo.

A Figura 20 corresponde a https://github.com/tankesho/e-museu: cerca de 67 *commits*, pastas típicas de um projeto Laravel recém-criado (`app`, `config`, `resources`, entre outras) e poucos artefatos na raiz além de `composer.json`, `package.json`, `phpunit.xml` e `vite.config.js`. Não há diretórios `docker/`, `docs/` ou `scripts/ci/`, nem ficheiros de *compose* ou de análise estática; o `Procfile` remete ao *deploy* em PaaS usado na implementação inicial.

As Figuras 21 e 22 mostram https://github.com/e-museu-utfpr-gp/e-museu em sequência (listagem longa da raiz). O histórico visível ultrapassa 200 *commits*; surgem `.github/` (ações de CI), `docker/`, `docs/` e `scripts/ci/`, vários `Dockerfile` e `docker-compose.*.yml` para ambientes local, *staging* e produção, o script `run` para tarefas no container, ficheiro `VERSION` e ferramentas de qualidade (`phpcs.xml`, `phpstan.neon`, `pint.json`, `eslint.config.js`, entre outras). As mensagens recentes nos ficheiros alinham-se ao Capítulo 4: refatoração por domínios, internacionalização, Redis, testes, Coolify e atualização para Laravel 13. Essa organização torna explícito, para quem abre o repositório, que o E-Museu deixou de ser apenas uma aplicação monolítica e passou a incluir *pipeline*, documentação e rotinas de implantação.

Figura 20 – Raiz do repositório legado `tankesho/e-museu` no GitHub.
Fonte: https://github.com/tankesho/e-museu (captura de maio/2026).

5.3.2 Interface pública e painel administrativo

A identidade visual e o *layout* geral da página inicial e do painel administrativo permanecem próximos aos da versão de Gonzaga (2024): a restilização ampla da interface não entrou no escopo deste TCC, cujo foco foi refatoração, infraestrutura, testes e funcionalidades de acervo

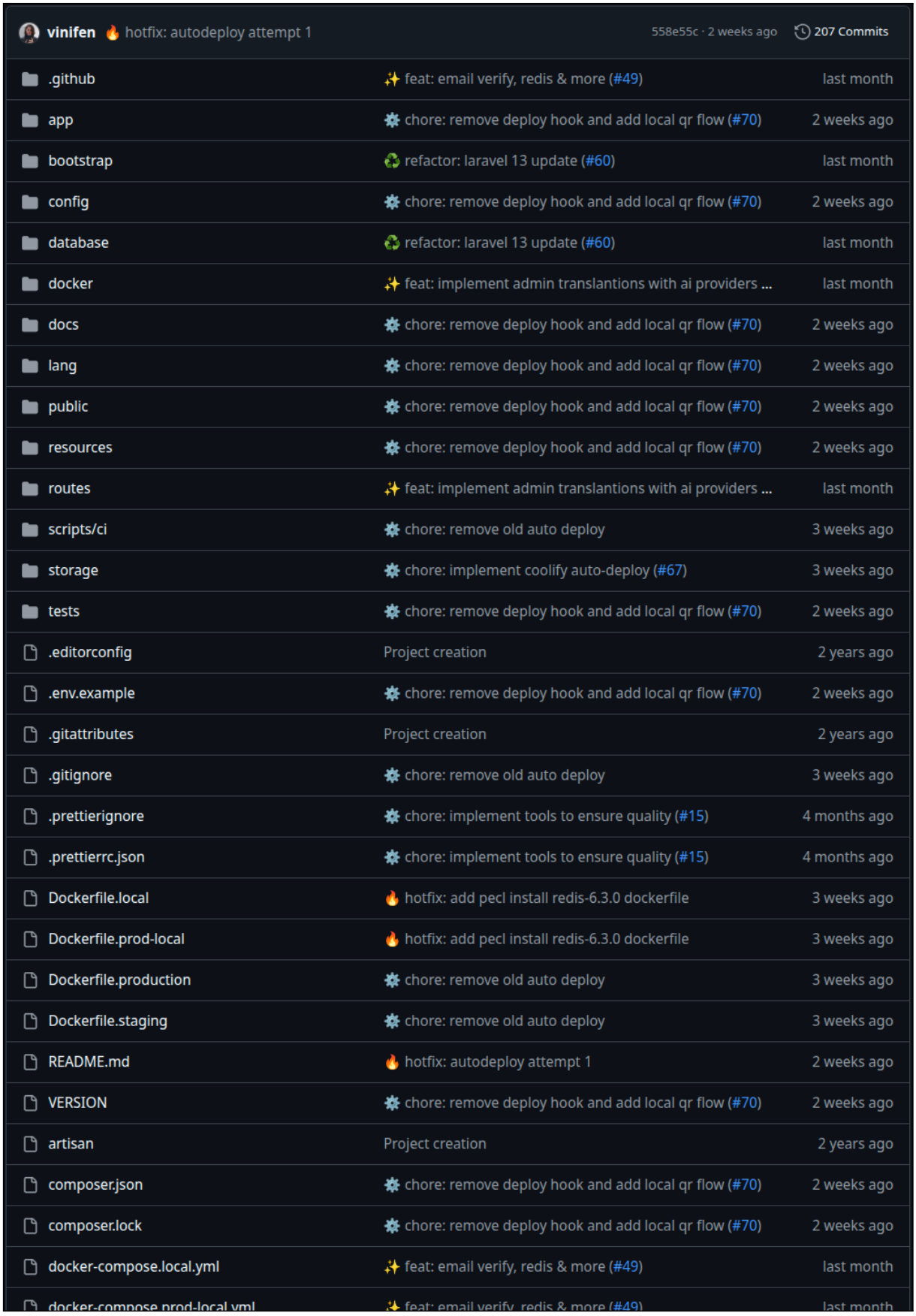


Figura 21 – Raiz do repositório institucional `e-museu-utfpr-gp/e-museu` no GitHub (parte 1).

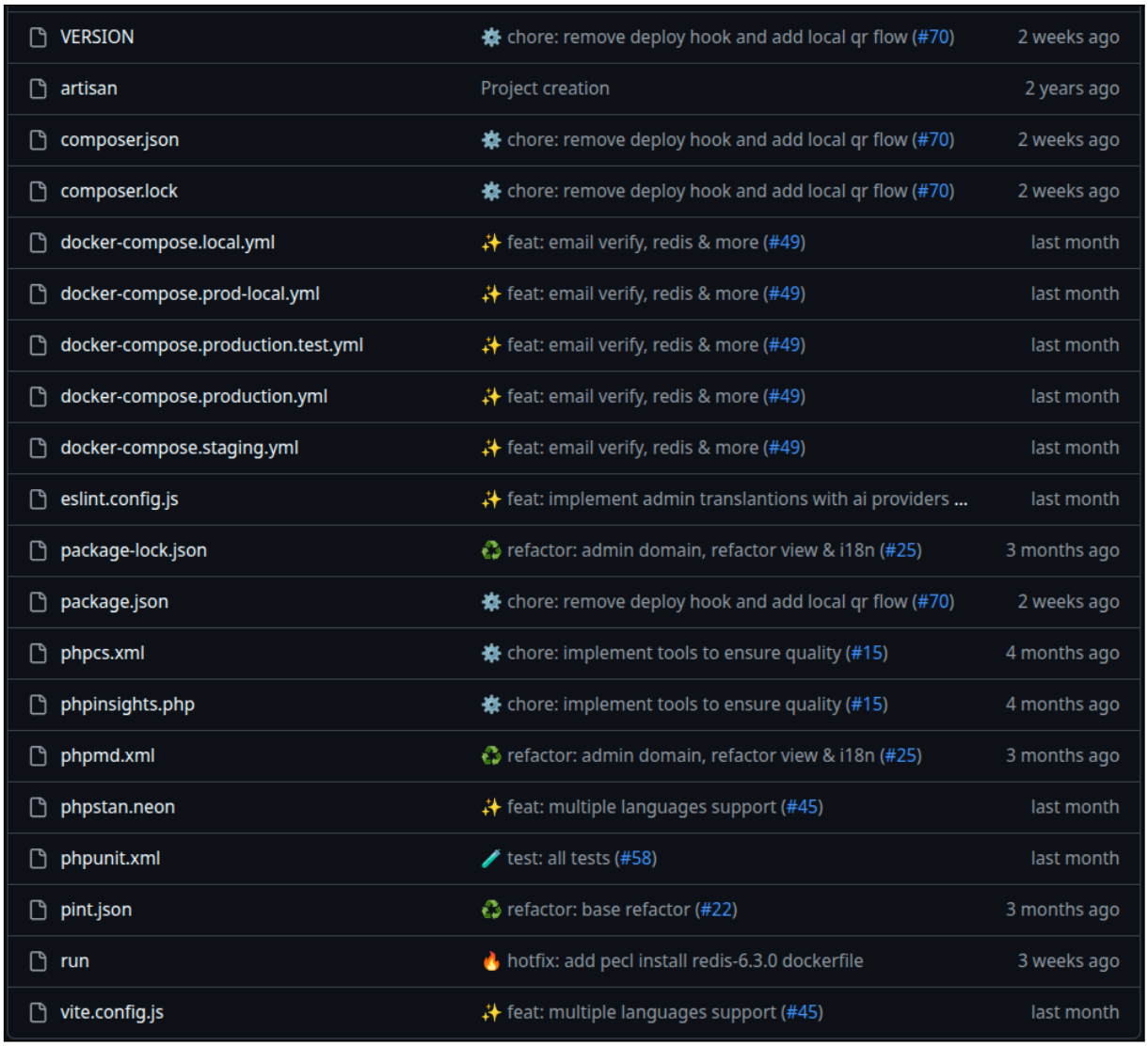


Figura 22 – Raiz do repositório institucional `e-museu-utfpr-gp/e-museu` no GitHub (parte 2; continuação).
Fonte: https://github.com/e-museu-utfpr-gp/e-museu (capturas de maio/2026).

descritas no Capítulo 4. Ainda assim, as Figuras 23 e 24 registram como o sistema se apresenta em produção após as entregas, para o leitor visualizar o produto final sem depender do site.

A Figura 23 mostra a página inicial pública em https://e-museu.gp.utfpr.edu.br/home: cabeçalho com navegação (*Explorar*, contribuição, sobre), seletor de idioma e bloco institucional sobre o museu virtual de informática da parceria UTFPR/UNICENTRO. As mudanças mais visíveis ao visitante concentram-se no catálogo e nos fluxos novos (internacionalização do acervo, contribuição com verificação por e-mail, Turnstile), ilustrados nas Seções 4.6 e 4.6.1, e não na *landing page*.

Figura 23 – Página inicial do E-Museu em produção (https://e-museu.gp.utfpr.edu.br/home). Fonte: Autoria própria (produção, maio/2026).

A Figura 24 apresenta o painel administrativo na listagem de itens do acervo (https://e-museu.gp.utfpr.edu.br/admin/catalog/items): menu lateral com entidades alinhadas ao modelo refatorado (categorias de item, colaboradores, administradores, entre outras), busca por campo, tabela com código de identificação, localização, validação e ações de visualização, edição e exclusão. O vocabulário do menu reflete renomeações do banco (Seção 5.2); telas específicas de tradução assistida, múltiplas imagens e QR Code aparecem no Capítulo 4.

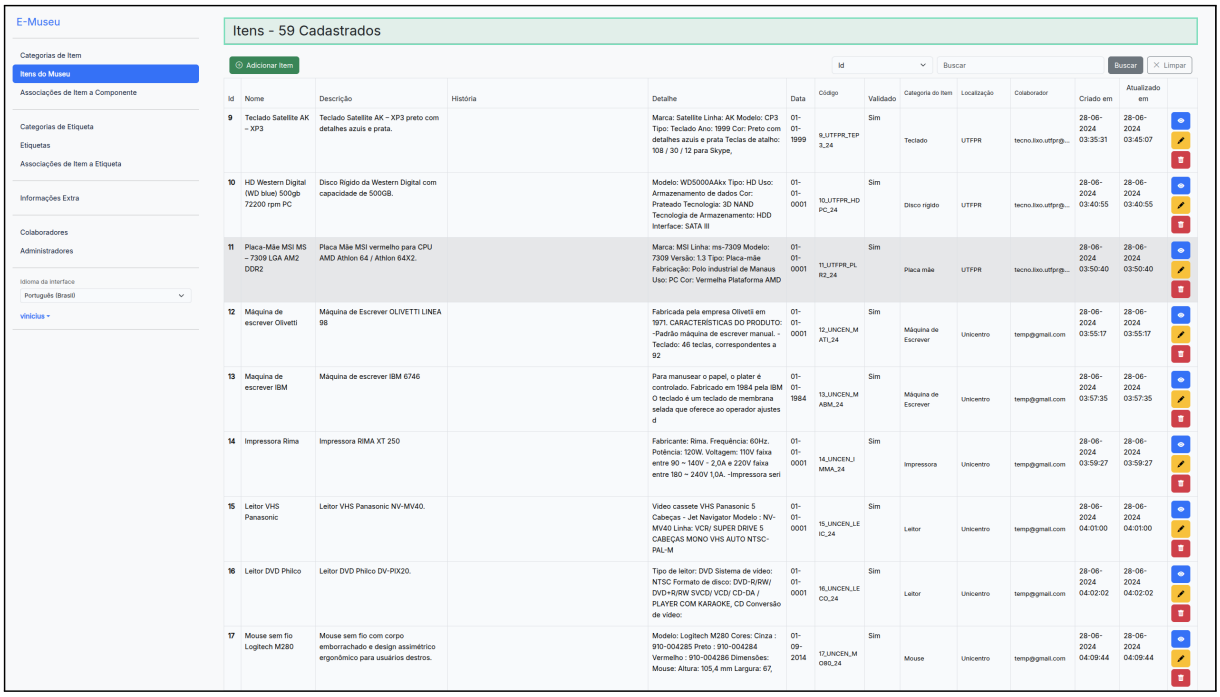


Figura 24 – Painel administrativo: listagem de itens do acervo (https://e-museu.gp.utfpr.edu.br/admin/catalog/items).

Fonte: Autoria própria (produção, maio/2026).

5.4 Indicadores quantitativos do trabalho

Esta seção reúne métricas verificáveis do ciclo: indicadores do repositório institucional e da linha legada, além do tempo dedicado e dos gastos diretos registrados ao longo do TCC.

5.4.1 Repositório e entregas no *GitHub*

Os indicadores abaixo foram obtidos ao final do TCC a partir do repositório institucional `e-museu-utfpr-gp/e-museu` e da linha legada usada para comparação (Tankensho), sem incluir dependências de terceiros (`vendor/`, `node_modules/`). Servem como ordem de grandeza da ampliação do esforço de engenharia; contagens locais (arquivos em `tests/` ou linhas em `app/`) podem ser recalculadas com `cloc` ou `git diff --stat` entre *tags*.

A Figura 25 reproduz a visão *Contributors* do *GitHub* para o ramo `main` (commits por semana, excluindo commits de *merge*), disponível em https://github.com/e-museu-utfpr-gp/e-museu/graphs/contributors?all=1. O gráfico superior mostra dois momentos distintos: um pico em meados de 2024, quando o usuário `tankesho` concentrou a implementação inicial herdada do trabalho de Gonzaga (2024); um intervalo com pouca atividade; e, a partir de janeiro de 2026, commits semanais contínuos até maio de 2026, período em que a refatoração foi realizada, a infraestrutura e as funcionalidades documentadas no Capítulo 4. Nos cartões individuais, `vinifen` aparece com 116 commits e cerca de 86,9 mil linhas adicionadas e 47,5

mil removidas; `tankesho`, com 66 commits e cerca de 29,3 mil adições e 4,9 mil remoções. Esses totais do *GitHub* contam adições e remoções em todo o histórico visível do repositório (código, testes, Docker, documentação e demais arquivos versionados), e não se limitam ao ciclo do TCC 2 nem a um único diretório. O saldo líquido de linhas de `vinifen` fica em torno de 39 mil, coerente com a magnitude do refactor documentado nesta monografia.

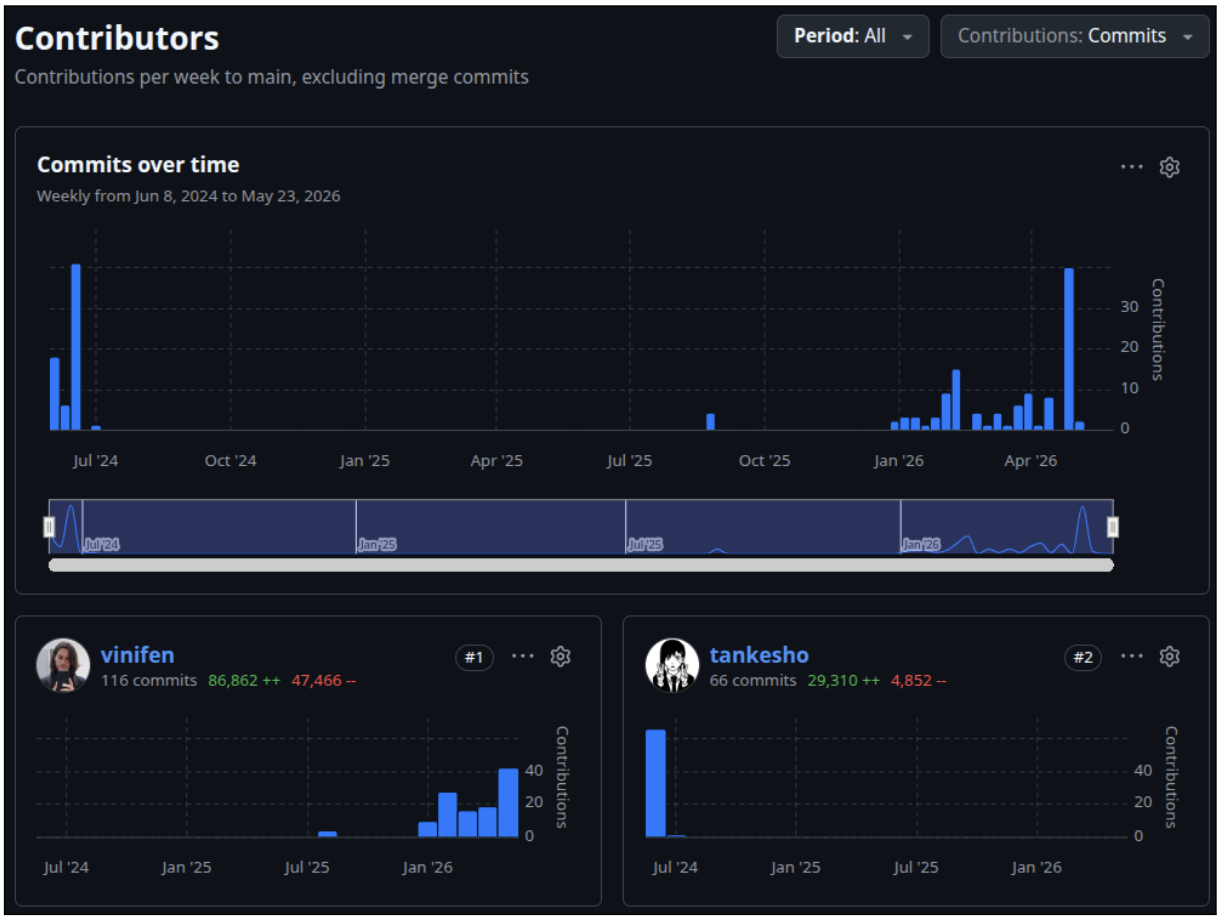


Figura 25 – Contribuições por commit no repositório `e-museu-utfpr-gp/e-museu` (ramo `main`).

Fonte: *GitHub Insights – Contributors* (https://github.com/e-museu-utfpr-gp/e-museu/graphs/contributors?all=1), captura de maio/2026.

- *Pull requests* documentados no histórico do TCC 2: 56 entregas exportadas do *GitHub* da organização `e-museu-utfpr-gp` (arquivo de apoio ao trabalho).
- Arquivos de teste PHP em `tests/`: 2 no legado; 92 na versão refatorada (inclui testes de fluxo, integração e unidade adicionados entre março e abril de 2026).
- Linhas em arquivos `.php` sob `app/`: aproximadamente 3,3 mil no legado; aproximadamente 13,8 mil na versão refatorada (contagem local com `cloc`, sem `vendor/`). O aumento reflete novos domínios, serviços e políticas de validação; é uma métrica complementar às adições e remoções do *GitHub*, que abrangem todo o repositório.

- *Releases* SemVer em `main`: `2.0.0-beta` e `2.1.0-beta` (infraestrutura e qualidade); `3.0.0-beta` (domínios, MinIO e base Laravel 12); `4.0.0-beta` (vocabulário do banco, `item_images` e camada de serviços); `5.0.0-beta` (7 de abril de 2026) e `1.0.0` (25 de abril de 2026), com internacionalização, `Location`, Turnstile, tradução assistida e Laravel 13 (Capítulo 4). Os sufixos `-beta` indicaram versões de homologação; `1.0.0` marca a primeira entrega estável promovida a produção institucional.
- Atividade em `main` no *GitHub* (Figura 25): pico de commits em maio de 2026, alinhado ao encerramento da implantação institucional (Seção 4.8).

5.4.2 Tempo e recursos dedicados

Complementando as métricas do repositório, registraram-se o tempo de trabalho e os gastos diretos do ciclo. As sessões foram anotadas diariamente no *Notion* (Seção 3.1.19), com a tarefa realizada e a duração de cada período.

Entre dezembro de 2025 e maio de 2026, o total dedicado ao E-Museu e a este TCC ficou em cerca de 250 horas, incluindo implementação, infraestrutura, documentação, deslocamentos ao campus da UTFPR e redação da monografia (também no período de férias). Quanto aos gastos diretos, o montante ao longo do percurso foi de aproximadamente R$ 400,00: assinatura do *Cursor*, VPS na *DigitalOcean* para ensaios com *Coolify*, deslocamentos à UTFPR, pen drive do pacote de continuidade entregue à orientação e licença do *Overleaf* no TCC 1, quando o serviço ainda não era oferecido gratuitamente pelos professores.

5.5 Entrega operacional

O sistema encontra-se publicado em https://e-museu.gp.utfpr.edu.br, com validação prévia em https://staging.e-museu.gp.utfpr.edu.br. O ciclo de promoção (*staging* antes de `main`), a infraestrutura Docker/*Coolify* e os artefatos de continuidade (sem exposição de credenciais neste documento) estão descritos nas Seções 4.8 e 4.3.

A Figura 26 resume a topologia implantada na máquina virtual da UTFPR: um único servidor (2 CPU, 6 GB de RAM, cerca de 40 GB de disco) executa o *Coolify*, que provisiona dois ambientes isolados. Em cada um há o serviço principal (aplicação Laravel e *web* com *Nginx*), além de *MySQL*, *Redis* e *MinIO*. A homologação (*staging*) replica essa composição com variáveis e domínios próprios e recebe *deploy* a partir do ramo `develop`; a produção, a partir de `main`, conforme o Gitflow da Seção 4.8. O esquema condensa, em visão estática, a composição de serviços registrada no painel do *Coolify* (Figura 11, Capítulo 4).

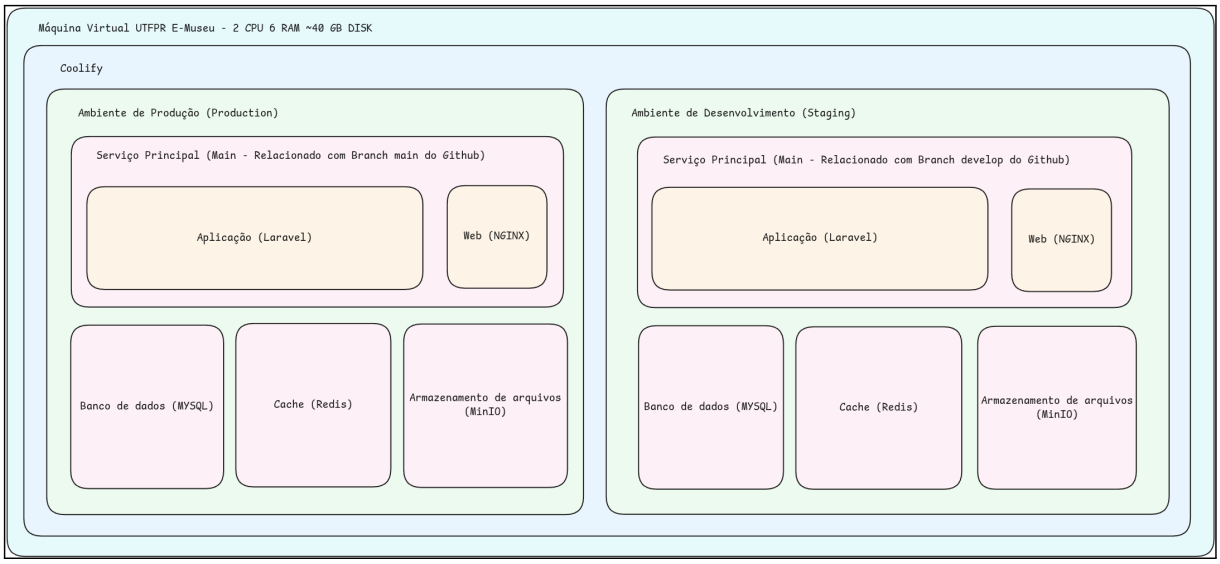


Figura 26 – Infraestrutura do E-Museu na máquina virtual da UTFPR (*staging* e produção sob *Coolify*).

Fonte: Autoria própria no *Excalidraw* (maio/2026).

Em síntese, a entrega reúne as condições para a manutenção e a evolução do projeto na instituição, sem depender da estrutura monolítica e pouco testada da implementação anterior.

6 CONSIDERAÇÕES FINAIS

Este capítulo encerra a monografia: retoma os objetivos à luz do que foi realizado, relata as dificuldades do percurso, explicita as limitações do escopo, indica caminhos de continuidade do E-Museu, registra o uso de inteligência artificial e apresenta a conclusão do trabalho. O processo de desenvolvimento está no Capítulo 4; a consolidação dos resultados, na Tabela 2 e nas demais seções do Capítulo 5.

6.1 Retomada dos objetivos

O objetivo geral deste trabalho consistiu em refatorar e implementar novas funcionalidades no E-Museu, aplicando boas práticas de programação, padrões de projeto e testes automatizados, de modo a melhorar a manutenibilidade e a qualidade do código (Seção 1.1.1).

Em relação aos objetivos específicos (Seção 1.1.2), a Tabela 2 registra atendimento pleno a todas as metas: estrutura de versionamento no *GitHub*, configuração de ambientes, refatoração do código legado, ampliação da suíte de testes automatizados de fluxo e integração (Seção 4.5), novas funcionalidades priorizadas no MoSCoW (internacionalização, múltiplas imagens por item, identificação com QR Code, Turnstile, tradução assistida no painel, entre outras, Seção 4.6) e aplicação de padrões e boas práticas descritos no referencial teórico.

Em síntese, o trabalho partiu das limitações da versão anterior (GONZAGA, 2024) e concentrou-se em estabelecer uma base técnica mais organizada para o projeto institucional, conforme os Capítulos 4 e 5.

6.2 Dificuldades encontradas

Além das limitações de escopo citadas adiante, o desenvolvimento enfrentou obstáculos concretos em cada fase. No início, a maior dificuldade foi definir com precisão o escopo e a estrutura do projeto: alinhar requisitos com a orientação, priorizar com MoSCoW e decidir o que entrava neste ciclo demandou várias rodadas de conversa antes do ritmo estável de entregas (Seção 4.1).

Posteriormente, vieram as atualizações amplas de versões (*PHP*, Laravel e dependências) e a correção do legado à luz das novas ferramentas de análise estática e de testes. Cada mudança de versão exigiu ajustes em massa no código até a *pipeline* voltar a passar (Seções 4.3.2 e 4.7). A adaptação ao *Coolify* também foi trabalhosa: redes Docker, variáveis de ambiente, separação entre homologação e produção e falhas de *deploy* apareceram tanto no ensaio em VPS (Seção 3.1.4) quanto no servidor da UTFPR (Seção 4.8). O ensaio fora do campus mostrou-se necessário para não descobrir problemas no servidor institucional.

A estruturação por domínios e a organização da engenharia do repositório exigiram várias iterações. Parte do acervo e das funcionalidades foi entregue antes da organização final por domínios em `app/` (Catalog, Taxonomy, entre outros), o que gerou retrabalho para alinhar pastas, vocabulário do banco e camada de serviços sem repetir lógica e de forma legível para quem abre o código (Seção 4.4). Documentar tudo em paralelo ao código também foi desafiante. Com muitas frentes ao mesmo tempo, era fácil perder a ordem cronológica ou deixar de registrar uma decisão; o registro diário (Seção 3.1.19) e os arquivos em `docs/` e no pacote de continuidade (Seção 4.8.5) ajudaram, mas a disciplina de documentação competiu com o tempo de implementação.

Na implantação presencial na UTFPR, apesar dos ensaios prévios, surgiram problemas de ordem prática. A VM, no começo, tinha hardware insuficiente para manter *Coolify* e os serviços dos dois ambientes (detalhe na Subseção 4.8.2). Houve ainda algumas questões sobre o painel do *Coolify* (uso restrito ao campus versus subdomínio público com autenticação), o que motivou tentativas intermediárias de automação de *deploy* antes da solução estável por *webhooks* (Subseção 4.8.3). Outro ponto crítico foi o volume de senhas, *tokens* e chaves espalhados entre aplicação, *MySQL*, *Redis*, *MinIO*, *Coolify* e *GitHub*: um campo incorreto ou esquecido quebrava o ambiente. Para reduzir esse risco em manutenções futuras, reuniram-se exports e modelos de configuração em arquivos de apoio (copiar e colar no painel), no pacote de continuidade (Subseção 4.8.5), sem expor segredos nesta monografia.

6.3 Limitações

Este trabalho não esgotou todas as possibilidades de evolução do E-Museu. O escopo concentrou-se em refatoração, infraestrutura, testes e funcionalidades de acervo priorizadas no MoSCoW (Seção 4.1), com um único desenvolvedor e recursos de tempo e financeiros limitados (Seção 5.4.2). Não houve avaliação formal com usuários do acervo nem estudo sistemático de usabilidade ou acessibilidade da interface.

Essas condições delimitam o que foi possível validar neste ciclo; requisitos e melhorias que ficaram de fora do escopo atendido são indicados na Seção 4.6.5 e detalhados como propostas de continuidade na seção seguinte.

6.4 Trabalhos futuros e continuidade do E-Museu

O E-Museu é um projeto de longa duração, vinculado à UTFPR e a iniciativas de extensão relacionadas à preservação da história da informática. O presente trabalho deixou o sistema e o repositório em condições mais sustentáveis para quem der continuidade ao desenvolvimento, seja em novo Trabalho de Conclusão de Curso, seja em extensão ou na manutenção do sistema na UTFPR. O repositório institucional, a wiki de fluxo de trabalho, a documentação

de implantação e o pacote de continuidade descritos no Capítulo 4 servem de ponto de partida para não repetir a reorganização básica já feita neste ciclo.

A principal frente sugerida para trabalhos posteriores é a restilização da interface, pública e administrativa. O foco deste TCC foi a base técnica; as capturas em produção (Seção 5.3.2) mostram que *layout*, tipografia e navegação permanecem próximos aos de Gonzaga (2024). Uma reforma visual planejada com a orientação poderia melhorar usabilidade, acessibilidade e percepção do museu virtual sem comprometer a arquitetura já entregue.

Em seguida, destacam-se evoluções operacionais e de gestão do acervo:

- Backup automático do banco de dados e das imagens do acervo (por exemplo agendamento no servidor ou integração com o *MinIO* e o *MySQL*), complementando o pacote manual entregue à orientação (Seção 4.8.5) e reduzindo dependência de exportações pontuais.
- Relatórios de acesso ao catálogo público e ao painel (visitas, rotas mais usadas, período), para apoiar decisões de extensão e de divulgação do museu.
- Registro de logs das alterações feitas por administradores (quem editou ou excluiu item, categoria ou colaborador, e quando), reforçando rastreabilidade da curadoria.
- Newsletter por e-mail, aproveitando a infraestrutura de mensagens transacionais já usada na verificação de colaboradores e na contribuição (Seção 4.6.2), para avisar interessados sobre novidades no acervo ou no projeto.
- Campo de link de vídeo de demonstração por item (URL externa), para enriquecer a ficha da peça sem armazenar o arquivo de vídeo no servidor.

Outras melhorias podem ampliar a confiabilidade e a qualidade percebida: mais testes automatizados (incluindo usabilidade ou acessibilidade), avaliação com usuários do acervo, monitoramento do ambiente *Coolify* e redução de intervenções presenciais na máquina da UTFPR, quando a política institucional permitir.

Recomenda-se que futuros trabalhos formalizem requisitos e cronograma com a orientação, reutilizando o fluxo Gitflow documentado (Novacoski, Vinicius Ferreira, 2026), de modo a acumular esforço em funcionalidades e em validação sobre a base técnica já consolidada.

6.5 Uso de inteligência artificial

Tanto a implementação do E-Museu quanto a redação desta monografia contaram com apoio de ferramentas de inteligência artificial, em especial o *Cursor* (Capítulo 3). Esse auxílio aumentou a produtividade em relação ao tempo disponível: ajudou em rascunhos de código e de texto, em buscas no repositório, em revisões incrementais e em tarefas repetitivas que, feitas

só manualmente, exigiriam bem mais horas do que as cerca de 250 registradas para o conjunto do projeto (Seção 5.4.2).

Esse uso também se alinha ao que o mercado de desenvolvimento de software vem exigindo na prática: assistentes de IA integrados à *IDE*, à revisão de código e à documentação deixaram de ser curiosidade e passaram a fazer parte do fluxo de trabalho de equipes e de profissionais autônomos. O ganho de produtividade é grande quando a ferramenta apoia quem já domina o problema (arquitetura, testes, *deploy*), em vez de substituir o raciocínio técnico. O volume entregue neste TCC, refatoração ampla, infraestrutura, testes e monografia no mesmo ciclo, ilustra esse efeito no contexto de um único desenvolvedor.

Isso não significa que o trabalho tenha sido terceirizado. Permaneceu sob responsabilidade deste TCC a condução das decisões de arquitetura (organização por domínios, ambientes *Docker* e *Coolify*, fluxo *Gitflow*, escopo MoSCoW), a validação técnica (*pull requests*, testes, *deploy* na UTFPR), o alinhamento com a orientação e o conteúdo final deste documento. A IA executou ou acelerou partes concretas, mas o desenho do sistema, a ordem das entregas e a revisão do que foi integrado ao repositório e ao texto foram conduzidos neste ciclo.

6.6 Finalização

Encerra-se o Trabalho de Conclusão de Curso com o E-Museu em operação nos domínios institucionais, repositório organizado e objetivos específicos atendidos conforme documentado nesta monografia. O sistema segue disponível para visitantes e para a gestão do acervo; o texto registra o percurso técnico realizado, as dificuldades enfrentadas e as bases deixadas para que orientação, docentes e quem assumir a manutenção possam dar continuidade ao museu virtual como espaço de preservação e divulgação da memória tecnológica na UTFPR.

REFERÊNCIAS

Agile Business Consortium. **MoSCoW Prioritisation**. 2025. Acesso em: 28 out. 2025. Disponível em: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html.

Anysphere, Inc. **Cursor: AI-Powered Code Editor**. 2026. Acesso em: 17 maio 2026. Disponível em: https://cursor.com.

Bootstrap Team. **Bootstrap: Powerful, Extensible, and Feature-Packed Frontend Toolkit**. 2026. Acesso em: 17 maio 2026. Disponível em: https://getbootstrap.com.

CapRover Team. **CapRover: Build your own PaaS in a few minutes!** 2025. Acesso em: 14 out. 2025. Disponível em: https://caprover.com.

Cloudflare, Inc. **Turnstile**. 2026. Acesso em: 17 maio 2026. Disponível em: https://developers.cloudflare.com/turnstile/.

Composer Contributors. **Composer: Dependency Manager for PHP**. 2026. Acesso em: 17 maio 2026. Disponível em: https://getcomposer.org.

CoolLabs. **Coolify: An Open-Source Self-Hostable Heroku Alternative**. 2025. Acesso em: 14 out. 2025. Disponível em: https://coolify.io.

DEACON, J. **Model View Controller (MVC) Architecture**. 2009. PDF. Acesso em 26 de novembro de 2025. Disponível em: https://d1wqtxts1xzle7.cloudfront.net/50526307/MVC-libre.pdf.

Docker, Inc. **Docker Compose: Define and Run Multi-Container Applications**. 2025. Acesso em: 14 out. 2025. Disponível em: https://docs.docker.com/compose/.

Docker, Inc. **Docker: Empowering App Development for Developers**. 2025. Acesso em: 14 out. 2025. Disponível em: https://www.docker.com.

e-museu-utfpr-gp. **E-Museu Repository**. 2026. Acesso em: 19 mai. 2026. Disponível em: https://github.com/e-museu-utfpr-gp/e-museu.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. Addison-Wesley Professional, 2004. Disponível em: https://books.google.com.br/books?hl=pt-BR&lr=&id=xCoIAAPGubgC&oi=fnd&pg=PR9&dq=domain-driven+design+book&ots=qdYD8gPK3q&sig=besaDRaOWUTqRwniCWgyUGXM1jg&redir_esc=y#v=onepage&q=domain-driven%20design%20book&f=false.

F5, Inc. **nginx: High Performance Load Balancer, Web Server, & Reverse Proxy**. 2026. Acesso em: 17 maio 2026. Disponível em: https://nginx.org.

FOWLER, M. **Refatoração**. Bookman, 2004. ISBN 9788577804153. Disponível em: https://books.google.com.br/books?hl=pt-BR&lr=&id=p97eDwAAQBAJ&oi=fnd&pg=PT4&dq=refatora%C3%A7%C3%A3o&ots=l4ruH06Xda&sig=BlsHSJsKhhMPPhvTbGs91JnUeRM&redir_esc=y#v=onepage&q=refatora%C3%A7%C3%A3o&f=false.

Git Documentation Team. **Git - Documentation**. 2025. Acesso em: 14 out. 2025. Disponível em: https://git-scm.com/doc.

Git SCM. **Git: Distributed Version Control System**. 2025. Acesso em: 14 out. 2025. Disponível em: https://git-scm.com.

GITHUB. **GitHub**. 2025. Acesso em: 3 set. 2025. Disponível em: https://github.com.

GitHub, Inc. **About Pull Requests - GitHub Docs**. 2025. Acesso em: 14 out. 2025. Disponível em: https://docs.github.com/en/pull-requests.

GitHub, Inc. **GitHub Actions Documentation**. 2026. Acesso em: 17 maio 2026. Disponível em: https://docs.github.com/en/actions.

GitHub, Inc. **GitHub Models**. 2026. Acesso em: 17 maio 2026. Disponível em: https://docs.github.com/en/github-models.

GONZAGA, A. T. O. **E-Museu: explorando a história da informática em um ambiente virtual**. 2024. Monografia (Graduação) — Universidade Tecnológica Federal do Paraná, Guarapuava, 2024. Acesso em 17 de março de 2025. Disponível em: https://e-museu.gp.utfpr.edu.br/.

Groq, Inc. **Groq: Fast AI Inference**. 2026. Acesso em: 17 maio 2026. Disponível em: https://groq.com.

IEEE Computer Society. **The Importance of Software Testing**. 2025. Acesso em: 31 ago. 2025. Disponível em: https://www.computer.org/resources/importance-of-software-testing.

Laravel Lang Contributors. **Laravel Lang**. 2026. Acesso em: 17 maio 2026. Disponível em: https://laravel-lang.com.

Laravel Team. **Laravel: The PHP Framework for Web Artisans**. 2025. Acesso em: 14 out. 2025. Disponível em: https://laravel.com.

Laravel Team. **Laravel Pint**. 2026. Acesso em: 17 maio 2026. Disponível em: https://laravel.com/docs/pint.

MARTIN, R. C. **Código Limpo: Habilidades Práticas do Agile Software**. Alta Books, 2009. Tradução da obra *Clean Code*. ISBN 9788576082675. Disponível em: https://books.google.com.br/books?id=bmpeDwAAQBAJ.

MCCARTNEY, S. **ENIAC: The Triumphs and Tragedies of the World's First Computer**. Walker Company, 1999. Disponível em: https://dl.acm.org/doi/abs/10.5555/520152.

MinIO, Inc. **MinIO: High Performance Object Storage**. 2026. Acesso em: 17 maio 2026. Disponível em: https://min.io.

Notion Labs, Inc. **Notion: The AI Workspace**. 2026. Acesso em: 17 maio 2026. Disponível em: https://www.notion.so.

Novacoski, Vinicius Ferreira. **E-Museu Repository (2.1.0-beta)**. 2025. Acesso em: 19 mai. 2026. Disponível em: https://github.com/vinifen/e-museu-2.1.0-beta.

Novacoski, Vinicius Ferreira. **Gitflow Documentation for Project Development**. 2026. Acesso em: 19 mai. 2026. Disponível em: https://github.com/vinifen/gitflow-documentation.

OpenJS Foundation. **Node.js**. 2026. Acesso em: 17 maio 2026. Disponível em: https://nodejs.org.

OpenRouter, Inc. **OpenRouter: Unified API for LLMs**. 2026. Acesso em: 17 maio 2026. Disponível em: https://openrouter.ai.

Oracle Corporation. **MySQL: The World's Most Popular Open Source Database**. 2025. Acesso em: 14 out. 2025. Disponível em: https://www.mysql.com.

PHPMD Contributors. **PHPMD: PHP Mess Detector**. 2026. Acesso em: 17 maio 2026. Disponível em: https://phpmd.org.

PHPStan Contributors. **PHPStan: PHP Static Analysis Tool**. 2026. Acesso em: 17 maio 2026. Disponível em: https://phpstan.org.

Preston-Werner, Tom. **Semantic Versioning 2.0.0**. 2026. Acesso em: 23 maio 2026. Disponível em: https://semver.org/.

Redis Ltd. **Redis: The Real-time Data Platform**. 2026. Acesso em: 17 maio 2026. Disponível em: https://redis.io.

RIBEIRO, J. V. *et al.* Tecno-lixo: Oficina do aprender - relatos de experiência de um projeto de extensão. *In*: **Anais do XIV Seminário de Extensão e Inovação & XXIX Seminário de Iniciação Científica e Tecnológica da UTFPR - SEI/SICITE 2024**. Francisco Beltrão: [*s.n.*], 2024. p. 1–6.

Ré, A. M. D.; THOMEN, M. A. F. **Cartilha de sugestões para atividades em oficinas pedagógicas utilizando de informática LIXOELETRÔ**. 2021. https://www3.unicentro.br/decomp/wp-content/uploads/sites/77/2021/09/CartilhaLixoEletronico.pdf. Acesso em: 25 set. 2022.

Sass Contributors. **Sass: CSS with superpowers**. 2026. Acesso em: 23 maio 2026. Disponível em: https://sass-lang.com/.

Sebastian Bergmann. **PHPUnit**. 2026. Acesso em: 17 maio 2026. Disponível em: https://phpunit.de.

The PHP Group. **PHP: Hypertext Preprocessor**. 2025. Acesso em: 14 out. 2025. Disponível em: https://www.php.net.

The phpMyAdmin Team. **phpMyAdmin**. 2026. Acesso em: 17 maio 2026. Disponível em: https://www.phpmyadmin.net.

Vite Contributors. **Vite: Next Generation Frontend Tooling**. 2026. Acesso em: 17 maio 2026. Disponível em: https://vite.dev.

World Wide Web Consortium (W3C). **CSS: Cascading Style Sheets**. 2026. Acesso em: 23 maio 2026. Disponível em: https://www.w3.org/Style/CSS/.

World Wide Web Consortium (W3C). **HTML: HyperText Markup Language**. 2026. Acesso em: 23 maio 2026. Disponível em: https://www.w3.org/html/.