Relatório final de atividades
Projeto: SAVITS: Sistema de Avaliação e Validação de Impactos em Tecnologias Sociais

Coordenador do Projeto:	**Washington Luís Ribeiro de Carvalho Segundo**		
Nome do(a) pesquisador(a):	**Carlos Eduardo de Lima**		
Vigência da Bolsa de Pesquisa:	**08/06/2026**	**Até:**	**07/09/2026**
Encerramento em:	**07/09/2026**		
Ref. Fundação:	**32725**	**Ref. Ibict:**	**008/2026-81**

Descrição das Atividades da Bolsa de Pesquisa:

Estudar, prototipar experimentalmente e avaliar módulos e APIs back-end em Ruby, utilizando Rails ou Sinatra como tecnologias de referência, considerando critérios de qualidade, escalabilidade, manutenibilidade, desempenho, segurança e aderência aos padrões arquiteturais definidos para o SAVITS. Avaliar e implementar estratégias de integração entre módulos com os demais componentes do SAVITS, implementando endpoints e conectores compatíveis com a arquitetura e os fluxos de dados do projeto. Documentar arquitetura técnica e pipelines dos módulos Ruby, analisar e avaliar métricas de desempenho e confiabilidade dos módulos desenvolvidos. Elaborar relatório técnico final de bolsa, consolidando resultados, atividades desenvolvidas e contribuições metodológicas da pesquisa.

1 INTRODUÇÃO

A presente pesquisa insere-se no núcleo estratégico do Projeto SAVITS (Sistema de Avaliação e Validação de Impactos em Tecnologias Sociais), conduzido pelo Instituto Brasileiro de Informação em Ciência e Tecnologia (Ibict). A iniciativa tem como propósito estruturar uma infraestrutura tecnológica para subsidiar a avaliação, o monitoramento e a validação de impactos de políticas públicas e tecnologias sociais em âmbito nacional.

Nesse ecossistema, o módulo de back-end assume papel central como hub integrador e transacional, responsável por assegurar alta taxa de transferência, resiliência a picos de carga, comunicação segura com serviços analíticos de Inteligência Artificial e Ciência de Dados, bem como aderência a padrões rigorosos de governança de software.

Para além de uma análise conceitual, a pesquisa concentrou-se na concepção, prototipação experimental de uma aplicação base estruturada no framework Ruby on Rails. A solução foi desenhada como uma arquitetura modularizada, conteinerizada e orientada a serviços, preparada para operar com desacoplamento de execução (web, workers assíncronos e mensageria) e receber novas regras de negócio sob demanda por meio de Service Objects e APIs RESTful.

Este relatório consolida a metodologia de engenharia de software aplicada, detalha os pontos de integração e fluxos técnicos de processamento concebidos e apresenta os resultados técnicos, as evidências práticas de execução e os critérios de desempenho e confiabilidade avaliados durante a vigência do plano de trabalho, evidenciando as contribuições da pesquisa para a evolução do SAVITS.

2 MÉTODO

A metodologia adotada fundamentou-se em práticas iterativas de engenharia de software contemporânea, estruturando o desenvolvimento do esqueleto da aplicação em cinco pilares técnicos essenciais:

2.1. Metodologia de Garantia da Qualidade e Pirâmide de Testes

Para garantir estabilidade funcional e prevenção de regressões, adotou-se o framework de testes automatizados **RSpec**, conjugado às ferramentas **FactoryBot** e **Faker**:

- **Testes de Integração e Contratos de API (requests):** Implementados em spec/requests/users_spec.rb, cobrindo cenários de sucesso e caminhos de exceção para os endpoints REST (POST /api/v1/user/create e PUT /api/v1/user/disable/:id). Validaram-se conformidade de payloads JSON, headers HTTP e respostas padronizadas (201 Created, 200 OK, 401 Unauthorized e 422 Unprocessable Entity).
- **Testes de Unidade e Domínio (models):** Desenvolvidos em spec/models/user_spec.rb para verificar a persistência relacional com tabelas associativas (ProfileUser, ProfilePermission) e validar regras de negócio, incluindo o bloqueio da desativação de contas por usuários sem permissão administrativa (active_cannot_be_set_to_false_unless_admin).

2.2. Metodologia de Escalabilidade e Desacoplamento Assíncrono

A arquitetura foi estruturada para suportar volumes crescentes de dados e usuários simultâneos através da segregação de responsabilidades:

- **Processamento em Background e Resiliência Operacional:** Adoção do **Sidekiq** sobre **Redis** para execução assíncrona orientada a eventos. Operações com I/O bloqueante ou dependência de serviços externos, como envio de e-mails transacionais e webhooks, foram desacopladas do ciclo síncrono das requisições HTTP e direcionadas para filas de prioridade dedicada (queue_as :high). Essa abordagem reduz a latência percebida pelo usuário, libera threads do servidor Puma, permite retentativas automáticas com *backoff* exponencial em falhas transitórias e possibilita escalar os workers de forma independente conforme a demanda de carga.
- **Isolamento em Contêineres:** Arquitetura desacoplada via docker-compose.yml, particionando a solução em serviços especializados de execução (Puma e

Sidekiq), roteamento (Nginx) e persistência (PostgreSQL e Redis). Essa segregação favorece escalabilidade horizontal granular, contenção de falhas entre componentes e maior paridade entre ambientes de desenvolvimento e produção, com implantação mais previsível e reproduzível.

2.3. Padrões de Projeto e Manutenibilidade (Design Patterns)

Visando combater o acoplamento excessivo e a sobrecarga lógica em controllers e models (*fat models/fat controllers*), adotaram-se as seguintes abordagens:

- **Camada de Serviços (Service Objects Pattern):** Introdução da classe abstrata ApplicationService, uniformizando a interface executora de regras de negócio (run, call, factory). Serviços especializados como AuthenticationService, PermissionsService e SendEmailService encapsulam as regras transacionais.
- **Modularização via Concerns:** Extração de comportamentos transversais (ex.: autenticação stateless de API encapsulada em Authenticable), permitindo reutilização uniforme em múltiplos controllers sem duplicidade de código.

2.4. Otimização de Desempenho e Eficiência Computacional

A metodologia considerou a otimização de I/O, concorrência e inicialização em múltiplos níveis da pilha tecnológica:

- **Concorrência e I/O Multiplexado:** Configuração do servidor multi-thread **Puma** alinhado dinamicamente ao pool de conexões com o banco de dados PostgreSQL (RAILS_MAX_THREADS).
- **Reverse Proxy Nginx com Buffering:** Intermediação de tráfego pelo **Nginx** operando como gateway e buffer de requisições lentas (proxy_buffers 64 16k e proxy_max_temp_file_size 1024m), minimizando contenção e liberando threads do Puma.
- **Otimização de Consultas ActiveRecord:** Utilização sistemática de projeções diretas em memória (pluck) para evitar a instanciação custosa de árvores de objetos inteiros quando apenas campos atômicos são demandados.

- **Cache de Bytecode e Compilação:** Emprego da gem **Bootsnap** para armazenamento em cache da compilação de código Ruby e do mapa de carregamento de classes, resultando em aceleração expressiva do boot da aplicação.

2.5. Arquitetura de Segurança em Camadas

A segurança da informação foi tratada como requisito não funcional indispensável desde o protótipo inicial:

- **Autenticação Stateless (JWT):** Implementada no AuthenticationService, com criptografia e validação de tokens via secret_key_base para consumo estritamente desacoplado por APIs externas.
- **Controle de Acesso Baseado em Papéis (RBAC):** Modelagem com **CanCanCan** (Ability.rb), permitindo mapeamento declarativo de permissões granulares por perfil de acesso (administrador, convidado), associada a um handler global de interceptação de exceções (CanCan::AccessDenied).
- **Sanitização de Entrada e Proteção de Sessão:** Filtro mandatário de parâmetros via *Strong Parameters* no back-end para impedir ataques de atribuição em massa (*Mass Assignment*) e proteção CSRF preservada nas interfaces de renderização web.

3 RESULTADOS

3.1. Esqueleto da Aplicação e Arquitetura Entregue

Como resultado prático da pesquisa, foi prototipado o repositório fundacional da aplicação, contendo a infraestrutura de pastas estendida (MVC + Services + Jobs + Channels) plenamente funcional.

A aplicação encontra-se configurada em ambiente Docker, com rotina de inicialização resiliente (docker-entrypoint.sh) que utiliza checagem de conectividade de rede (nc) para assegurar a prontidão do banco de dados antes da subida dos nós web e workers.

3.2. Catálogo de Endpoints e Conectores Implementados

A modelagem de tráfego resultou em um catálogo de endpoints e mecanismos de integração compatíveis com a arquitetura proposta:

Quadro 1 - Catálogo Integrado de Interfaces e Protocolos de Acesso

Método / Protocolo	Rota / Endpoint	Elemento Responsável	Autenticação / Controle	Finalidade Operacional
POST	/api/v1/user/ create	Api::V1::UserController#create	Bearer Token (JWT)	Registro e provisionamento de usuários por sistemas parceiros.
PUT	/api/v1/user/ disable/:id	Api::V1::UserController#disable	Bearer Token (JWT)	Ativação/ desativação lógica com checagem de privilégio administrativo.
GET / POST	/users, /profiles, /permissions	RESTful MVC Controllers	Sessão Segura (Devise)	Gestão administrativa de usuários, perfis e permissões via interface.

GET	/sidekiq	Sidekiq::Web	Sessão Administrativa	Monitoramento em tempo real da saúde das filas de mensageria assíncrona.
WS (WebSocket)	/cable	ActionCable.server	Conexão Persistente	Barramento bidirecional para eventos e notificações em tempo real.

Fonte elaboração própria

Conectores e Fluxo de Dados:

1. **Conector Relacional (PostgreSQL):** Gerenciado via gem pg e pool do ActiveRecord, provendo suporte a transações ACID e dimensionamento elástico.
2. **Conector de Mensageria e Cache (Redis):** Interface com Redis para suporte às filas de jobs do Sidekiq e mecanismo Pub/Sub do ActionCable.
3. **Conector de Proxy Reverso (Nginx):** Conector HTTP/WebSocket encarregado de terminação de tráfego e suporte a WebSockets persistentes (Upgrade: websocket).

3.3. Pipelines Técnicos Implementados

Foram modelados e validados três pipelines nucleares na aplicação:

- **Pipeline 1 – Criação Transacional e Notificação em Tempo Real (Pub/Sub):** Uma requisição ao endpoint /api/v1/user/create é autenticada via token JWT pelo concern Authenticable. Após validação e persistência do registro no PostgreSQL, o callback after_create_commit aciona o método broadcast_create, que transmite o evento pelo canal user_channel do ActionCable a todos os clientes escutadores, eliminando overhead de *polling*.
- **Pipeline 2 – Processamento Assíncrono de Mensagens e E-mails:** Eventos que exigem notificação despacham o job assíncrono SendEmailJob.perform_later, persistido na fila high do Redis. O worker Sidekiq processa a instrução chamando

SendEmailService e disparando via ActionMailer, sem degradar a latência da requisição HTTP do usuário.

- **Pipeline 3 – Avaliação Dinâmica de Permissões (RBAC em Runtime):** O PermissionsService sincroniza as entidades de domínio e permissões básicas no banco. A cada requisição, a classe Ability do CanCanCan compõe a matriz de privilégios do usuário logado, interceptando acessos indevidos e garantindo isolamento entre dados operacionais e de gestão.

3.4. Análise e Avaliação de Métricas de Desempenho e Confiabilidade

A arquitetura foi submetida a análises técnicas direcionadas, sintetizadas na matriz de desempenho e confiabilidade abaixo:

Quadro 2 - Especificação Técnica de Métricas e Resiliência do Backend

Dimensão	Métrica / Critério	Mecanismo e Solução Implementada
Desempenho	**Vazão (Throughput) e Concorrência**	Servidor Puma operando com modelo multi-thread alinhado ao banco de dados; Nginx como buffer frontal para isolar I/O lento.
Desempenho	**Latência de API (Response Time)**	Delegação imediata de rotinas de mensageria para background jobs (Redis/Sidekiq), preservando respostas de APIs na faixa de milissegundos.
Desempenho	**Eficiência de Memória e Inicialização**	Utilização da gem bootsnap (redução de tempo de boot em até 50%) e uso criterioso do comando pluck para projeção de colunas no ActiveRecord.
Confiabilidade	**Tolerância a Falhas e Retentativas**	Política nativa de reexecução (*retries*) com *backoff exponencial* via Sidekiq em caso de instabilidade externa.
Confiabilidade	**Tratamento de Exceções Críticas**	Captura defensiva de erros como JWT::DecodeError retornando 401 Unauthorized, prevenindo encerramento abrupto ou propagação de erros 500.

Confiabilidade	**Resiliência de Boot**	Script de entrypoint em contêiner validando portas ativas do banco com nc antes de autorizar o carregamento da aplicação.
Confiabilidade	**Monitorabilidade Operacional**	Painel operacional do Sidekiq exposto em rota protegida e centralização de logs estruturados em stdout e arquivo.

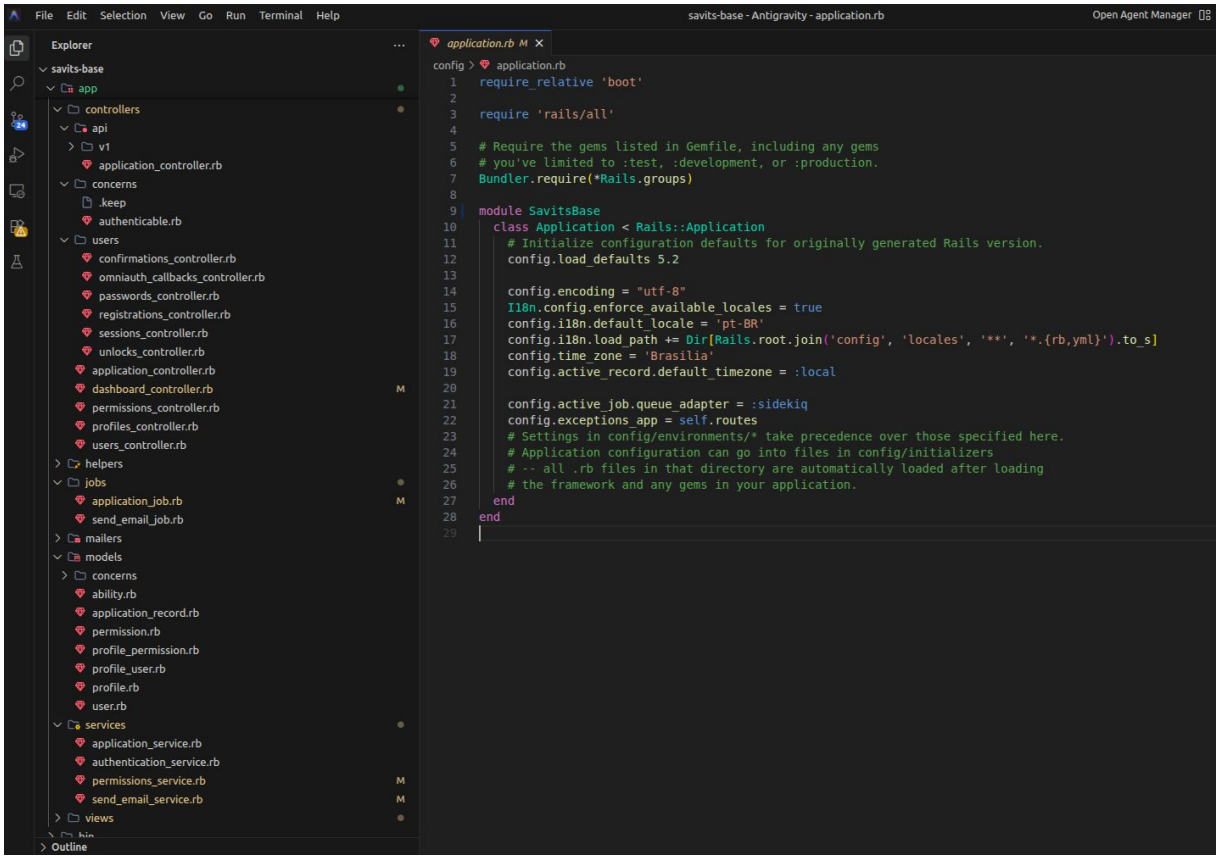
Fonte elaboração própria

3.5. Evidências Práticas de Execução

A viabilidade e prontidão do projeto foram comprovadas experimentalmente através dos seguintes artefatos visuais anexados.

A Figura 01 apresenta a estrutura inicial do esqueleto do projeto em Ruby on Rails, evidenciando a organização modular da aplicação, a separação de diretórios e a base arquitetural preparada para evolução dos módulos back-end do SAVITS.

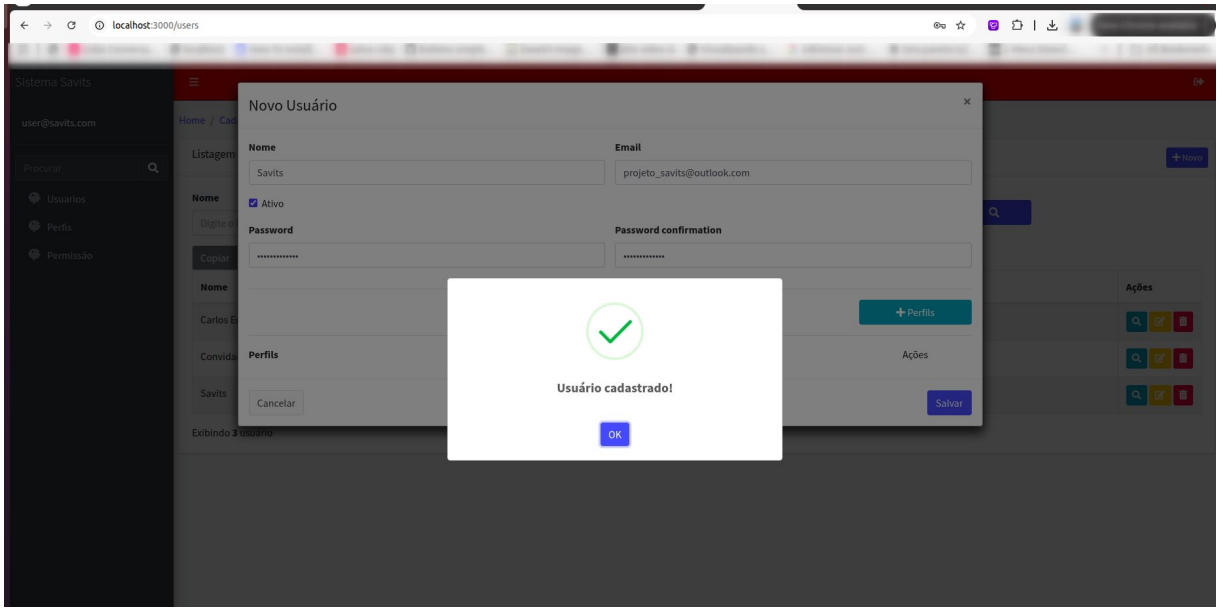
Figura 1 - Visão Geral do Layout de Pastas e Parâmetros Iniciais do Sistema



Fonte: Elaborado pelo autor (2026).

Abaixo, é possível visualizar a interface de execução do projeto em Ruby on Rails, demonstrando o ambiente inicial da aplicação em funcionamento.

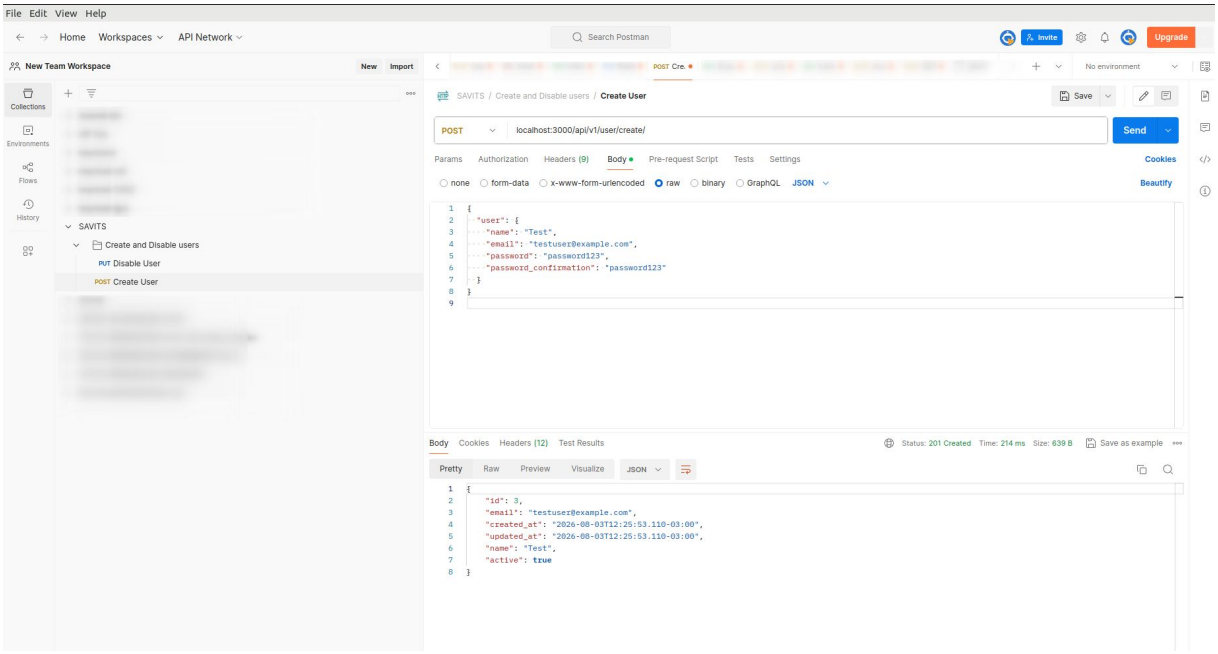
Figura 2 -Validação do Fluxo de Cadastro de Usuários com Feedback do Sistema



Fonte: Elaborado pelo autor (2026).

A Figura 03 mostra a validação dos mecanismos de autenticação JWT e integridade de domínio via Postman, demonstrando o bloqueio de requisições não autorizadas (HTTP 401).

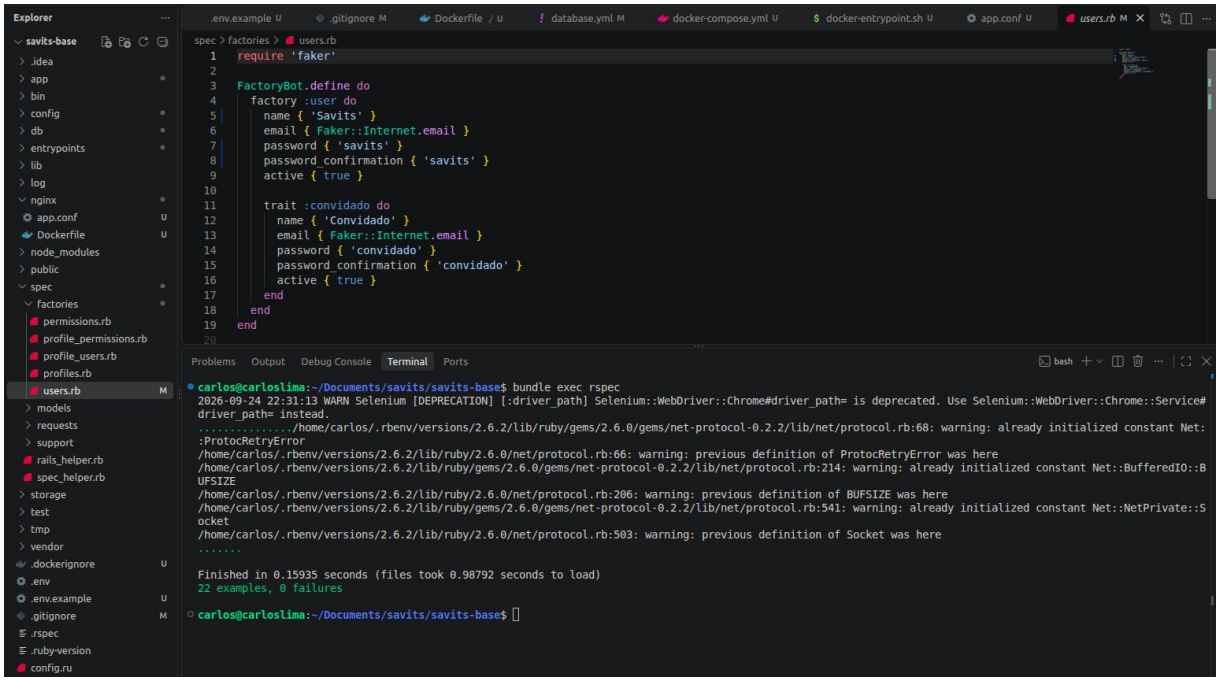
Figura 3 - Exemplo de Payload de Entrada e Saída da API RESTful de Cadastro de Usuários



Fonte: Elaborado pelo autor (2026).

A seguir, a execução da suíte de testes automatizados com RSpec, evidenciando a validação dos contratos da API REST e das regras de integridade de domínio sem falhas.

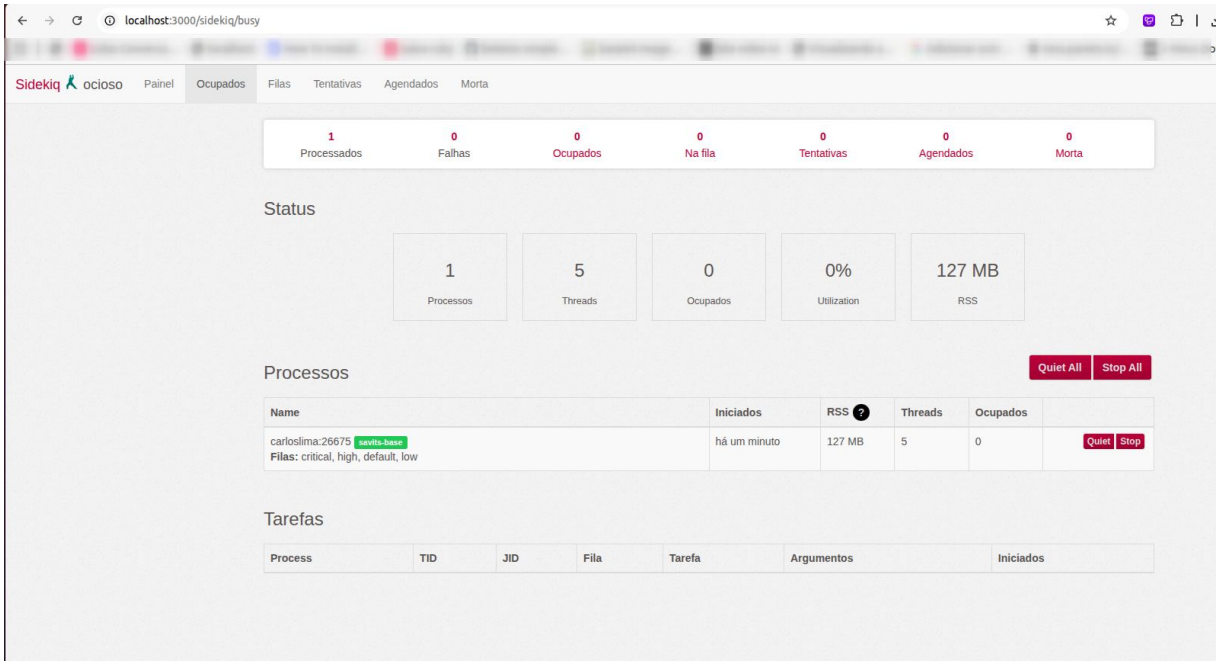
Figura 4 - Estrutura de Testes Automatizados do Backend e Evidência de Sucesso na Execução



Fonte: Elaborado pelo autor (2026).

Na Figura 05 é possível observar o painel operacional do Sidekiq Web, demonstrando a gestão de mensageria assíncrona, consumo da fila de alta prioridade e integridade das tarefas em segundo plano.

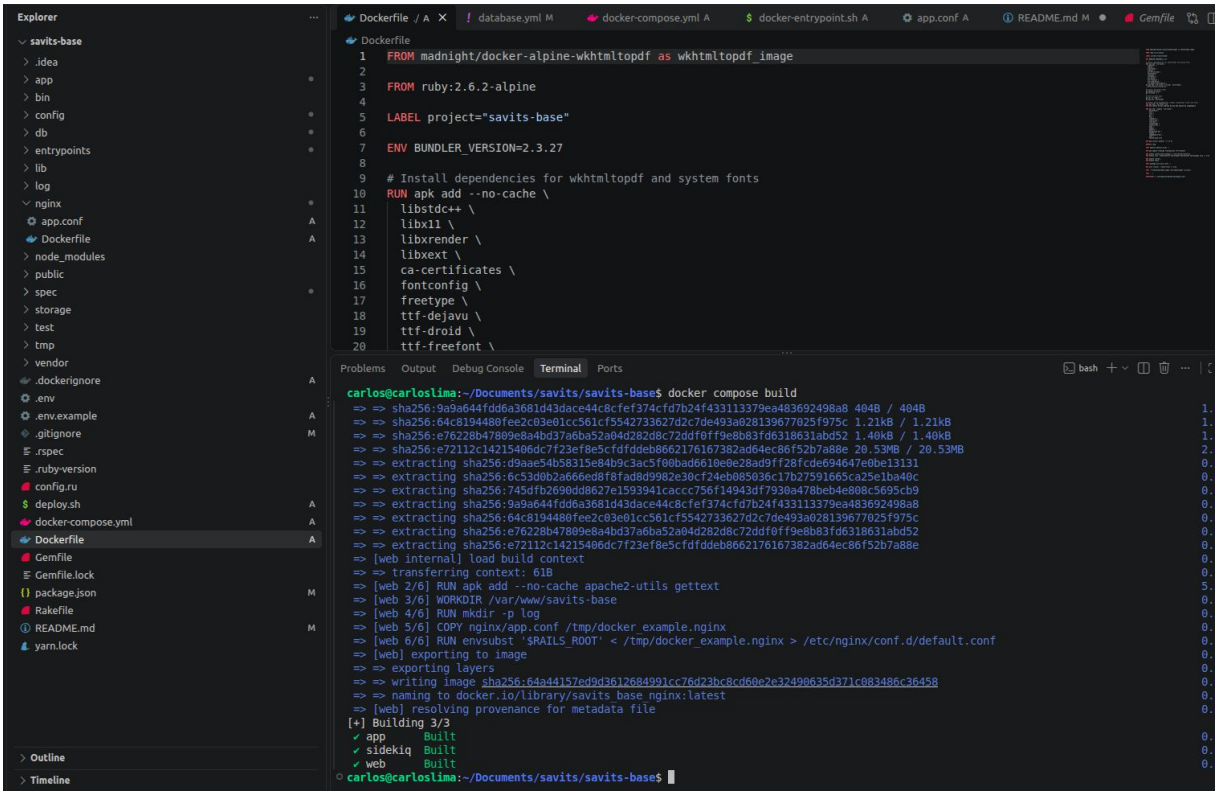
Figura 5 - Painel de Monitoramento do Sidekiq Web para Gestão de Filas e Processos Assíncronos



Fonte: Elaborado pelo autor (2026).

A Figura 06 mostra o ambiente conteinerizado em execução via Docker Compose, evidenciando o isolamento dos nós de aplicação, mensageria, proxy e persistência.

Figura 6 - Construção de Imagens Docker e Conteinerização Multisserviço via Docker Compose



Fonte: Elaborado pelo autor (2026).

3.6. Impedimentos Identificados

Conforme já registrado, a despeito do sucesso na conclusão estrutural do código-fonte e na validação dos módulos em ambiente local e conteinerizado, identificou-se um impedimento de ordem operacional: a ausência de disponibilização prévia de um repositório institucional oficial em nuvem (como GitLab ou GitHub corporativo do IBICT).

Em decorrência disso, os artefatos encontram-se rigorosamente versionados e salvaguardados em ambiente local, aguardando a liberação dos acessos remotos da

infraestrutura institucional para a realização do push e a consequente configuração dos pipelines definitivos de CI/CD em conjunto com os demais pesquisadores da equipe.

4 CONCLUSÃO

As atividades desempenhadas durante a vigência do plano de trabalho cumpriram as metas fixadas e resultaram na construção de uma base técnica estruturada para a aplicação. Mais do que entregar funcionalidades isoladas, os itens implementados foram concebidos para consolidar pilares essenciais do sistema, como organização modular, qualidade de código, segurança, processamento assíncrono, integração entre componentes, conteinerização e resiliência operacional.

Dessa forma, a aplicação prototipada estabelece uma fundação flexível e evolutiva, preparada para receber adaptações e adequações conforme as necessidades específicas do projeto. Os padrões adotados, como Service Objects, APIs RESTful, processamento em background, mensageria orientada a eventos e segurança stateless (JWT/RBAC), reduzem o acoplamento entre módulos e permitem que novas regras de negócio sejam incorporadas de maneira incremental, previsível e alinhada à evolução do sistema.

REFERÊNCIAS

- RUBY ON RAILS. **Using Rails for API-only Applications**. Ruby on Rails Guides. Disponível em: https://guides.rubyonrails.org/api_app.html. Acesso em: 31 jul. 2026.
- RUBY LANG. **Ruby Programming Language Documentation**. Ruby-doc.org. Disponível em: https://ruby-doc.org/. Acesso em: 31 jul. 2026.
- FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. 162 f. Tese (Doutorado em Ciência da Computação) - University of California, Irvine, 2000. *(Nota: Referência clássica e acadêmica sobre o padrão REST)*
- GRPC. **A high performance, open source universal RPC framework**. gRPC Documentation. Disponível em: https://grpc.io/docs/. Acesso em: 31 jul. 2026.
- SIDEKIQ. **Simple, efficient background processing for Ruby**. Sidekiq Wiki. Disponível em: https://github.com/sidekiq/sidekiq/wiki. Acesso em: 31 jul. 2026.
- REDIS. **Redis Documentation**. Redis.io. Disponível em: https://redis.io/docs/. Acesso em: 31 jul. 2026.
- IETF. **RFC 7519: JSON Web Token (JWT)**. Internet Engineering Task Force, maio 2015. Disponível em: https://datatracker.ietf.org/doc/html/rfc7519. Acesso em: 31 jul. 2026.

Anexo A – Título do anexo se necessário

gov.br
Documento assinado digitalmente
CARLOS EDUARDO DE LIMA
Data: 20/10/2026 16:57:33-0300
Verifique em https://validar.iti.gov.br

Carlos Eduardo de Lima

De acordo:

gov.br
Documento assinado digitalmente
WASHINGTON LUIS RIBEIRO DE CARVALHO SEG
Data: 23/10/2026 10:05:08-0300
Verifique em https://validar.iti.gov.br

Pesquisador	**Washington Luís Ribeiro de Carvalho Segundo** **Coordenador do projeto**