Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Reengenharia do Sistema SisGera

Matheus Lopes Moreira

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:

Euler Horta Marinho

COORIENTAÇÃO:

Fernando Bernardes de Oliveira

Julho, 2026
João Monlevade–MG

Matheus Lopes Moreira

Reengenharia do Sistema SisGera

Orientador: Euler Horta Marinho
Coorientador: Fernando Bernardes de Oliveira

Monografia apresentada ao curso de Engenharia de Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto
João Monlevade
Julho de 2026

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

M838r Moreira, Matheus Lopes.
Reengenharia do Sistema SisGera. [manuscrito] / Matheus Lopes Moreira. - 2026.
77 f.: il.: color., tab..

Orientador: Prof. Dr. Euler Horta Marinho.
Coorientador: Prof. Dr. Fernando Bernardes de Oliveira.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Engenharia de Computação .

1. Arquitetura de software. 2. Serviços de bombeiros voluntários (Brasil). 3. Sistemas de informação gerencial. 4. Software - Refatoração. I. Marinho, Euler Horta. II. Oliveira, Fernando Bernardes de. III. Universidade Federal de Ouro Preto. IV. Título.

CDU 004.41:004.2

Bibliotecário(a) Responsável: Flavia Reis - CRB6/2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Matheus Lopes Moreira

Reengenharia do Sistema Sisgera

Monografia apresentada ao Curso de Engenharia de Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação

Aprovada em 27 de julho de 2026

Membros da banca

Prof. Dr. Euler Horta Marinho - Orientador (Universidade Federal de Ouro Preto)
Prof. Dr. Fernando Bernardes de Oliveira - Coorientador (Universidade Federal de Ouro Preto)
Prof. Dr. Igor Muzetti Pereira - (Universidade Federal de Ouro Preto)
Prof. Dr. Luiz Carlos Bambirra Torres - (Universidade Federal de Ouro Preto)

Euler Horta Marinho, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 09/08/2026

Documento assinado eletronicamente por **Euler Horta Marinho**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 09/08/2026, às 11:32, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1155502** e o código CRC **A27CEDFC**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho a meus pais, meu irmão e minha esposa, que sempre me incentivaram a estudar, acreditaram e confiaram em mim apesar das dificuldades no caminho.

Agradecimentos

Agradeço primeiramente a Deus, por me dar força e sabedoria para concluir mais esta etapa e por colocar em meu caminho as pessoas que tornaram tudo isso possível.

Aos meus pais e ao meu irmão, agradeço por serem a base que me sustentou ao longo de toda a graduação. Sempre acreditaram que eu chegaria até aqui e não me deixaram desistir, e a eles devo o sacrifício e o apoio que tornaram este momento possível.

À minha esposa, agradeço pelo apoio e pela paciência durante toda esta jornada. Seu incentivo foi essencial para que eu não desanimasse diante das adversidades da graduação.

Agradeço também aos meus familiares e amigos, pelas palavras de incentivo e pela torcida ao longo desses anos, que me deram ânimo nos momentos mais difíceis.

Ao meu orientador, Euler Horta Marinho, e ao meu coorientador, Fernando Bernardes de Oliveira, agradeço pela orientação, pela disponibilidade e pelos ensinamentos que guiaram este trabalho, e aos professores do curso de Engenharia de Computação da Universidade Federal de Ouro Preto, pela formação que recebi ao longo da graduação.

Por fim, agradeço às corporações de bombeiros voluntários que deram origem e propósito ao SisGera, e a todos que, de alguma forma, contribuíram para a realização deste trabalho.

"O sucesso não é por acaso. É trabalho duro, perseverança, aprendizado, estudo, sacrifício e, acima de tudo, amor por aquilo que se faz."

— Pelé (1940 – 2022)

Resumo

O Sistema de Gerenciamento e Registro de Atividades (SisGera) apoia as atividades operacionais e administrativas de corporações de bombeiros voluntários e, após anos de evolução como uma aplicação monolítica, passou a apresentar limitações de manutenção, testabilidade e escalabilidade. Este trabalho teve como objetivo refatorar o SisGera para uma arquitetura de microsserviços, com a modernização de sua base tecnológica, a adaptação do *pipeline* de integração e entrega contínuas desenvolvido em trabalho anterior e a validação do sistema por meio de testes automatizados. A metodologia partiu da análise do sistema legado e de seus problemas, seguida da decomposição do *back-end* em três serviços definidos por contextos de negócio (identidade e acesso, ocorrências e inventário), implementados com NestJS e TypeScript, cada um com um esquema de banco de dados PostgreSQL próprio. Foram implementados um modelo de controle de acesso por papéis e permissões, um mecanismo de auditoria automática das operações, uma esteira de integração e entrega contínuas com GitHub Actions e a publicação dos serviços em nuvem com monitoramento de erros. Como resultados gerais, os três serviços foram implantados em ambiente de homologação, cobertos por 259 testes unitários com cobertura superior a 90% na camada de serviço, e cada limitação identificada no sistema legado foi tratada por uma solução correspondente na nova arquitetura. Conclui-se que a refatoração corrige os problemas estruturais da versão anterior e estabelece uma base sustentável para a evolução do sistema em trabalhos futuros.

Palavras-chave: Microsserviços. Refatoração de software. Arquitetura de software. Domain-Driven Design.

Abstract

The Activity Management and Registration System (SisGera) supports the operational and administrative activities of volunteer fire departments and, after years of evolution as a monolithic application, began to show limitations in maintainability, testability, and scalability. This work aimed to refactor SisGera into a microservices architecture, to modernize its technology stack, to adapt the continuous integration and delivery pipeline developed in previous work, and to validate the system through automated tests. The methodology started with the analysis of the legacy system and its problems, followed by the decomposition of the back end into three services defined by business contexts (identity and access, occurrences, and inventory), implemented with NestJS and TypeScript, each backed by its own PostgreSQL schema. A role-based access control model, an automatic audit mechanism, a continuous integration and delivery pipeline with GitHub Actions, and cloud deployment with error monitoring were also implemented. As general results, the three services were deployed to a staging environment, covered by 259 unit tests with over 90% coverage of the service layer, and each limitation identified in the legacy system was addressed by a corresponding solution in the new architecture. The refactoring fixes the structural problems of the previous version and establishes a sustainable foundation for the evolution of the system in future works.

Key-words: Microservices. Software refactoring. Software architecture. Domain-Driven Design.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

2FA *Two-Factor Authentication*

API *Application Programming Interface*

BO Boletim de Ocorrência

CD *Continuous Delivery*

CI *Continuous Integration*

CORS *Cross-Origin Resource Sharing*

DDD *Domain-Driven Design*

DECSI Departamento de Computação e Sistemas

DTO *Data Transfer Object*

e2e *end-to-end*

HTTP *HyperText Transfer Protocol*

ICEA Instituto de Ciências Exatas e Aplicadas

JM João Monlevade

JWT *JSON Web Token*

MVC *Model-View-Controller*

ORM *Object-Relational Mapping*

PaaS *Platform as a Service*

RBAC *Role-Based Access Control*

REST *Representational State Transfer*

SisGera Sistema de Gerenciamento e Registro de Atividades

SSH *Secure Shell*

TCC Trabalho de Conclusão de Curso

UFOP Universidade Federal de Ouro Preto

Sumário

1 Introdução

O Sistema de Gerenciamento e Registro de Atividades (SisGera) foi desenvolvido inicialmente por Arantes (2018) com o objetivo de informatizar o controle de ocorrências das corporações de bombeiros voluntários de São Domingos do Prata e Barão de Cocais. O sistema atendeu a uma necessidade real das corporações, que até então faziam seus registros de forma manual, o que deixava os dados sujeitos a perdas e inconsistências e dificultava sua consulta.

Com o sucesso da iniciativa, novos trabalhos ampliaram o escopo do sistema ao longo dos anos e acrescentaram funcionalidades para a gestão administrativa das corporações, como controle de inventário, patrimônio e doações Oliveira (2018). Cada nova versão ampliou o sistema, mas, à medida que o SisGera cresceu, limitações estruturais tornaram-se evidentes. A arquitetura monolítica adotada nas versões anteriores, adequada ao escopo original, passou a apresentar dificuldades de manutenção, escalabilidade restrita e forte acoplamento entre funcionalidades distintas.

Diante desse cenário, o presente trabalho propõe a refatoração do SisGera para uma nova arquitetura, que moderniza sua base tecnológica sem abandonar o propósito original do sistema. A reestruturação busca corrigir as limitações arquiteturais acumuladas e estabelecer uma base sustentável para que futuros Trabalhos de Conclusão de Curso possam continuar a evoluir a ferramenta.

1.1 Problema

Após oito anos de desenvolvimento, o sistema conta com uma estrutura extensa e de difícil manutenção, que concentra múltiplas funcionalidades em arquivos volumosos, além de conter código legado e funcionalidades que nunca foram implantadas em produção. No nível do banco de dados, foram identificadas inconsistências de modelagem, alto acoplamento entre entidades, tabelas com todas as colunas nuláveis e tabelas com dados imutáveis, que poderiam ser representados como enumerações na camada de aplicação.

Além das deficiências internas do sistema, a própria arquitetura adotada ao longo das versões anteriores é uma limitação estrutural. O SisGera foi desenvolvido sob um modelo monolítico, no qual *front-end* e *back-end* coexistem em um único projeto acoplado. Essa escolha, comum em sistemas desenvolvidos de forma incremental por diferentes trabalhos acadêmicos, dificulta a separação de responsabilidades, impede que as camadas evoluam de forma independente e torna o sistema cada vez mais resistente a mudanças.

Diante desse contexto, o presente trabalho propõe a reestruturação do SisGera

em uma arquitetura de microsserviços, com o objetivo de eliminar o acoplamento entre as camadas, corrigir as deficiências de modelagem identificadas e estabelecer uma base sustentável para a evolução contínua do sistema.

1.2 Objetivos

O presente trabalho consiste no desenvolvimento, na implantação e na validação da nova arquitetura e infraestrutura do SisGera, para tornar o projeto mais eficiente, seguro e sustentável a longo prazo. Esse objetivo é alcançado pela construção prática do *back-end*, pela modernização do banco de dados e pela adaptação do *pipeline* de integração e entrega contínuas (CI/CD) desenvolvido por Campos (2025), o que reduz falhas, facilita a manutenção e amplia a capacidade de evolução futura do sistema.

Durante o desenvolvimento do projeto, são utilizadas ferramentas que apoiam a criação desse tipo de aplicação, e as etapas de desenvolvimento do sistema são apresentadas com base em conceitos de Engenharia de *Software*.

1.2.1 Objetivo geral

Desenvolver uma nova arquitetura para o SisGera, com foco em longevidade, modernização tecnológica, escalabilidade e facilidade de manutenção a longo prazo.

1.2.2 Objetivos específicos

Este trabalho possui os seguintes objetivos específicos:

- Implementação da nova arquitetura de microsserviços;
- Adaptação do *pipeline* de CI/CD desenvolvido por Campos (2025);
- Validação por meio de testes automatizados;
- Avaliação de desempenho e qualidade entre a arquitetura original e a nova.

1.3 Organização do trabalho

O restante deste trabalho é organizado como se segue. O Capítulo 2 apresenta uma revisão da bibliografia, no qual são mencionados temas relacionados ao desenvolvimento do projeto e sistemas correlatos. O Capítulo 3 descreve as escolhas tecnológicas e as abordagens utilizadas no processo de desenvolvimento da nova arquitetura do SisGera, desde as etapas iniciais e decisões arquiteturais até a validação das funcionalidades. O Capítulo 4 apresenta os resultados da refatoração, comparando o sistema legado com a

nova arquitetura e descrevendo o *pipeline* em operação. Por fim, a conclusão retoma os objetivos do trabalho e aponta sugestões para trabalhos futuros.

2 Revisão bibliográfica

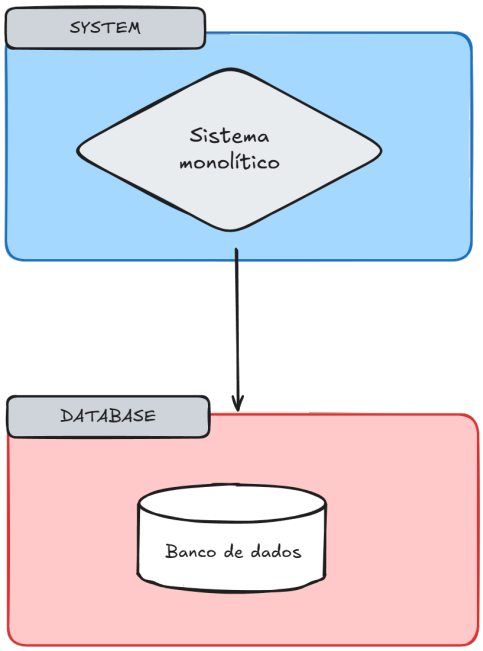
2.1 Considerações iniciais

Antes de apresentar o desenvolvimento da nova arquitetura do SisGera, é necessário estabelecer a base conceitual que fundamenta as decisões tomadas ao longo do trabalho. Este capítulo revisa os principais temas envolvidos: a comparação entre a arquitetura monolítica e a de microsserviços, os padrões de comunicação entre serviços, os conceitos de *Domain-Driven Design* que orientam a divisão do sistema, a autenticação *stateless* com JWT e o papel dos testes automatizados. Cada um desses temas é retomado nos capítulos seguintes, quando as escolhas do projeto são apresentadas e justificadas.

2.2 Arquitetura monolítica versus microsserviços

Antes de apresentar o novo modelo de arquitetura do SisGera, é preciso entender o conceito de monólito. Segundo Newman (2020), quando todas as funcionalidades de um sistema estão implantadas em conjunto, esse sistema é monolítico. Tais sistemas podem ter várias instâncias de processo por questões de robustez ou escala, mas todo o código está contido em um único processo, e são considerados simples porque leem ou armazenam dados em um único banco de dados, como na Figura 1, onde o sistema e o banco de dados são componentes separados, mas fortemente acoplados, rodando juntos em um único bloco ou executável.

Figura 1 – Esquema de sistema monolítico



Fonte: elaborado pelo autor.

Os sistemas monolíticos frequentemente são mais vulneráveis aos problemas causados pelo acoplamento elevado entre seus componentes. Um desses problemas se manifesta diretamente nas equipes de desenvolvimento, pois à medida que mais pessoas trabalham sobre a mesma base de código, fica mais difícil implantar novas funcionalidades sem introduzir regressões, já que qualquer alteração em um módulo pode impactar comportamentos em outros. Esse fenômeno, conhecido como acoplamento de equipe, tende a desacelerar o ritmo de entrega conforme o sistema cresce.

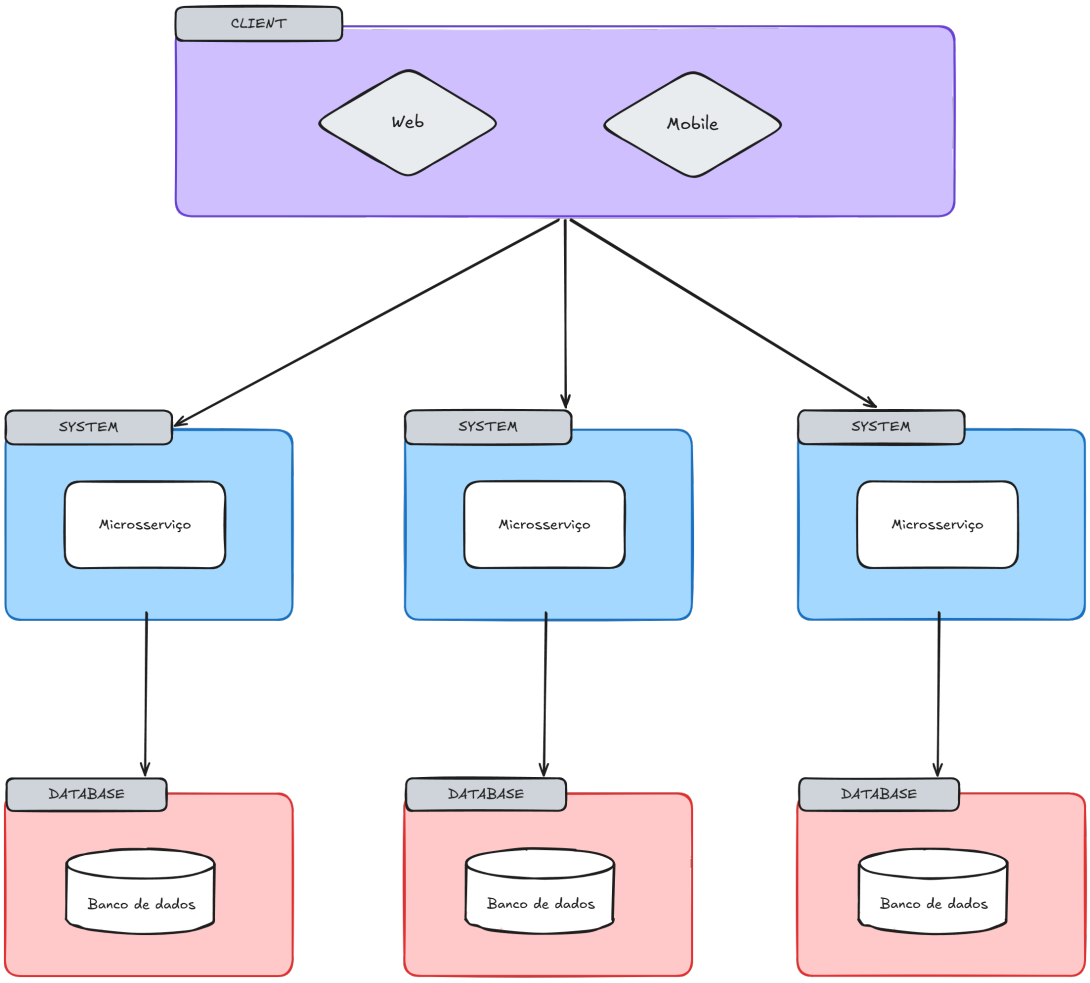
No nível técnico, o acoplamento da base de código impõe um custo elevado a cada mudança estrutural: modificar a lógica de um componente frequentemente exige ajustes em outros pontos do sistema, tornando a manutenção onerosa e arriscada. Associado a isso, a escalabilidade dos sistemas monolíticos é limitada: como todos os módulos compõem um único processo, qualquer necessidade de maior capacidade computacional obriga a replicação de toda a aplicação, e não apenas do componente sobrecarregado, o que resulta em desperdício de recursos.

Por fim, o processo de implantação também é afetado, pois qualquer alteração, por menor que seja, exige a construção e o *redeploy* de toda a aplicação, aumentando o risco de falhas e o tempo de inatividade. Em sistemas com múltiplas equipes e ciclos de entrega distintos, esse modelo se torna um gargalo significativo para a evolução do *software*.

Com as limitações impostas pela arquitetura monolítica, os microsserviços surgem como uma alternativa que busca segmentar o sistema e seus problemas em serviços menores, com responsabilidades bem definidas. Segundo Newman (2020), microsserviços são serviços

pequenos que colaboram entre si, cada um modelado em torno de um domínio de negócio específico, executando em seu próprio processo e se comunicando por meio de redes, como na Figura 2, em que existe um cliente comum, mas cada serviço possui seu bloco de código e banco de dados próprio. Uma das principais vantagens é a possibilidade de implantações independentes, já que cada serviço é uma unidade autônoma, que pode ser alterada e implantada em produção sem impactar os demais. Isso reduz bastante o risco de cada entrega e permite que diferentes equipes trabalhem em ciclos de desenvolvimento independentes, aumentando a velocidade e a frequência das implantações.

Figura 2 – Esquema de microsserviços



Fonte: elaborado pelo autor.

A escalabilidade também é beneficiada de forma direta. Em um sistema distribuído por microsserviços, é possível alocar recursos computacionais apenas para os componentes que apresentam maior demanda, sem a necessidade de replicar toda a aplicação. Em um sistema de gestão de ocorrências, por exemplo, o módulo responsável pelo registro de chamados pode ser escalado de forma isolada em períodos de alta demanda, sem que isso afete os demais serviços.

Outro benefício relevante é a flexibilidade tecnológica: como cada serviço é independente, é possível adotar a linguagem, o banco de dados ou o *framework* mais adequado para cada contexto, sem que essa escolha imponha restrições aos demais serviços. Por fim, a falha de um serviço isolado não necessariamente compromete o funcionamento do sistema como um todo, o que aumenta a resiliência geral da aplicação.

Apesar das vantagens apresentadas, adotar microsserviços não é isento de desafios. Distribuir o sistema em múltiplos serviços introduz uma complexidade operacional que não existe em arquiteturas monolíticas, já que gerenciar cada serviço de forma independente exige maturidade da equipe e infraestrutura mais elaborada.

Outro problema é a consistência de dados. Em uma arquitetura de microsserviços, cada serviço gerencia seu próprio banco de dados, o que impossibilita o uso de transações distribuídas tradicionais. Manter a consistência entre serviços exige estratégias específicas, que aumentam a complexidade do desenvolvimento.

A comunicação via rede também impõe um custo técnico adicional, já que as operações deixam de ser chamadas internas de funções e passam a ser requisições HTTP entre serviços, sujeitas a latência, falhas de rede e necessidade de tratamento de erros distribuídos. Isso exige que os serviços sejam projetados com tolerância a falhas, lidando com cenários de *timeout* e indisponibilidade de dependências.

Por fim, testar o sistema como um todo torna-se mais complexo. Enquanto os testes unitários permanecem mais simples dentro de cada serviço, validar o comportamento entre múltiplos serviços requer estratégias de teste e integração mais elaboradas. Dessa forma, a escolha pela arquitetura de microsserviços deve ser ponderada em função da complexidade do domínio e da capacidade da equipe de absorver essa sobrecarga operacional.

2.3 Padrões de comunicação entre serviços

Em uma arquitetura de microsserviços, os serviços não compartilham memória nem chamadas diretas de funções. Toda interação ocorre por meio de comunicação explícita pela rede, classificada em síncrona e assíncrona.

Na comunicação síncrona, o serviço que origina a requisição aguarda a resposta antes de prosseguir com sua execução. Esse modelo é direto e intuitivo, adequado para operações que exigem uma resposta imediata, como consultas e validações em tempo real. Na comunicação assíncrona, o serviço emissor publica uma mensagem em um intermediário e segue sua execução sem aguardar resposta. Esse modelo é mais adequado para operações que podem ser processadas em segundo plano, como notificações e integrações entre sistemas com volumes elevados de dados.

O estilo arquitetural predominante para comunicação síncrona em microsserviços é

o REST, que utiliza o protocolo HTTP como base e organiza a comunicação em torno de recursos identificados por URLs, operados por meio de verbos semânticos: GET para leitura, POST para criação, PUT e PATCH para atualização, e DELETE para remoção. Uma de suas características centrais é a ausência de estado (*stateless*): cada requisição deve conter todas as informações necessárias para ser processada, sem que o servidor mantenha contexto entre chamadas. Essa propriedade simplifica a escalabilidade horizontal, pois qualquer instância do serviço pode atender qualquer requisição.

Alternativas ao REST, como o gRPC, que utiliza contratos definidos em *Protocol Buffers* e comunicação binária, oferecem melhor desempenho em cenários com alto volume de dados e comunicação intensiva entre serviços internos. No entanto, sua adoção implica maior complexidade de configuração e menor legibilidade para fins de desenvolvimento e depuração. Para o contexto do SisGera, cujos serviços possuem volume de dados moderado e necessidade de interface documentada para consumo por clientes *web*, o REST mostrou-se a escolha mais adequada, complementado pela especificação *OpenAPI* para geração automática de documentação interativa via *Swagger*.

2.4 *Domain-Driven Design* e *Bounded Contexts*

Como dito anteriormente, modelar serviços em torno de um domínio de negócio traz vantagens importantes para a arquitetura de microsserviços. Parte-se do princípio de que o *software* deve refletir com fidelidade o domínio que representa. Assim, em vez de organizar o sistema em torno de camadas técnicas, o *Domain-Driven Design* propõe que a estrutura do código reflita o modelo do problema que está sendo resolvido, de modo que desenvolvedores e especialistas compartilhem os mesmos conceitos e uma mesma linguagem, denominada linguagem ubíqua.

Um contexto delimitado (*bounded context*) é uma fronteira explícita dentro de um sistema, na qual responsabilidades bem definidas são atribuídas, o que torna o modelo de domínio válido e coerente. Dentro desse limite, os termos, as regras e as entidades possuem significado preciso e consistente. Fora dele, o mesmo termo pode ter um significado diferente ou simplesmente não existir. Essa delimitação evita que um modelo único e genérico tente representar realidades distintas, o que inevitavelmente gera ambiguidades e acoplamento indesejado.

A relação entre DDD e microsserviços é clara: cada *bounded context* é um candidato natural a se tornar um microsserviço independente. Ao alinhar os limites dos serviços com os limites do domínio, garante-se que cada serviço possua alta coesão interna, com responsabilidades relacionadas entre si, e dependa minimamente dos demais para cumprir sua função.

No contexto do SisGera, essa análise identificou três domínios naturalmente distintos:

o gerenciamento de identidade e acesso, que engloba usuários, perfis e permissões; o registro e acompanhamento de ocorrências atendidas pelos bombeiros; e o controle de inventário, composto por patrimônio, produtos em estoque e doações recebidas. Cada um desses contextos possui entidades, regras de negócio e ciclos de vida próprios, o que justifica sua implementação como serviços independentes, a *auth-api*, a *occurrence-api* e a *inventory-api*, decisão detalhada no Capítulo 3.

2.5 Autenticação *stateless* com JWT

Em sistemas distribuídos, a autenticação pode ser implementada de duas formas principais: *stateful* e *stateless*. Na primeira, o servidor mantém uma sessão ativa para cada usuário autenticado, armazenando o estado em memória ou banco de dados. A cada requisição, o servidor consulta esse estado para verificar a identidade do usuário, o que exige que requisições de um mesmo usuário sejam sempre direcionadas à mesma instância do servidor, ou que o estado da sessão seja compartilhado entre instâncias, aumentando a complexidade e o acoplamento da infraestrutura.

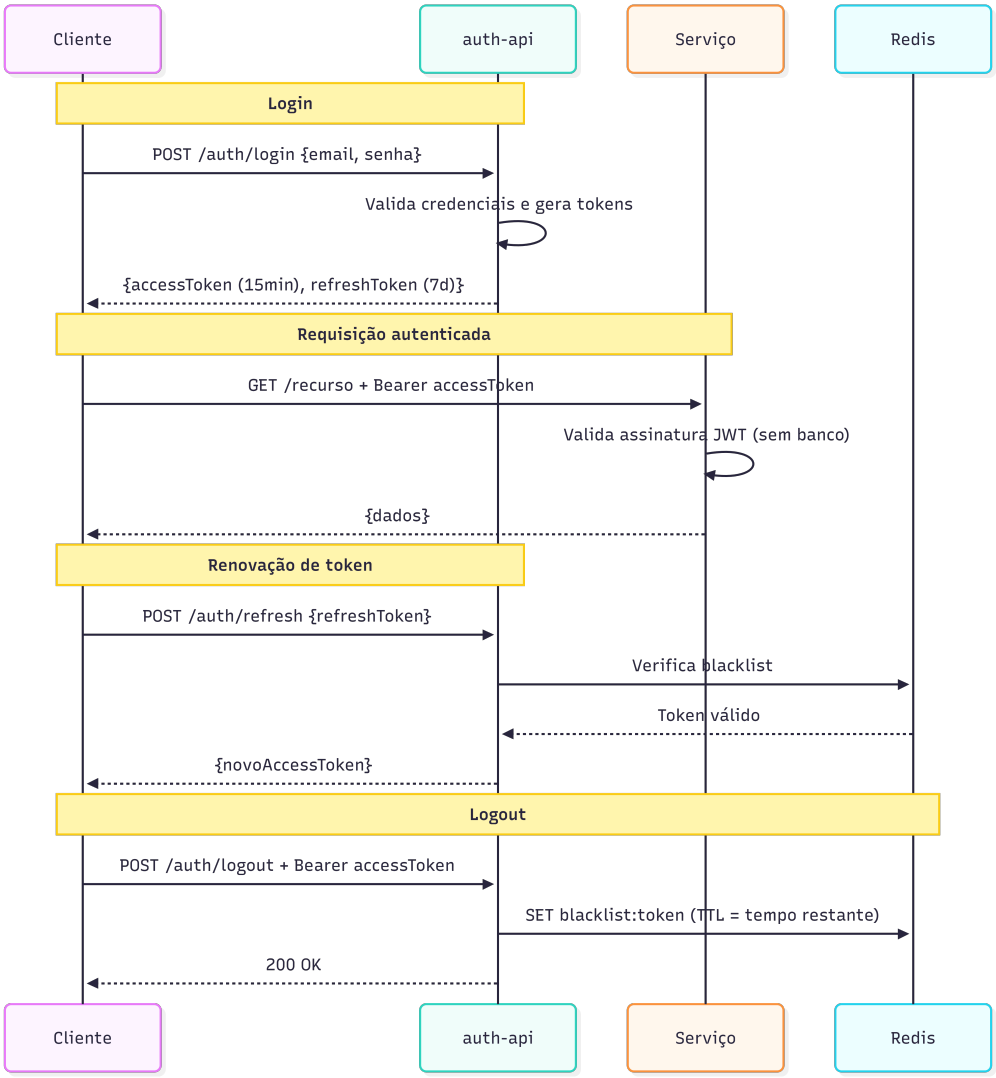
A autenticação *stateless* elimina a dependência de manter o estado da sessão no servidor e o transfere para o próprio cliente, por meio de um *token* assinado. O padrão mais adotado é o JWT, composto por três partes codificadas em Base64 e separadas por pontos: o cabeçalho (*header*), que armazena informações como o tipo de *token* e o algoritmo usado para gerar a assinatura; a carga útil (*payload*), que guarda as informações do *token*, como o momento de geração, a data de expiração e o identificador do usuário; e a assinatura (*signature*), que garante a integridade do *token* e assegura que ele foi gerado pela aplicação e não foi modificado, sendo um *hash* da junção do *header* e do *payload*, ambos em *base64url*.

Para equilibrar segurança e usabilidade, é comum adotar uma estratégia de dois *tokens*: o *access token*, de curta duração (15 minutos a 1 hora), utilizado para autenticar as requisições, e o *refresh token*, de longa duração, utilizado para obter um novo *access token* quando o anterior expira. Essa separação limita a janela de exposição em caso de comprometimento do access *token*, sem obrigar o usuário a autenticar-se novamente com frequência.

Uma limitação inerente ao JWT é a impossibilidade de invalidação antes do vencimento: como a validação é feita pela assinatura, um *token* válido permanece válido até expirar, mesmo após o *logout* do usuário. Para contornar isso, adota-se uma *blacklist* de *tokens* invalidados, armazenada em uma estrutura de acesso rápido. No SisGera, essa estratégia foi implementada com o *Redis*: ao realizar *logout*, o *token* é inserido na *blacklist* com um tempo de expiração equivalente ao tempo restante de sua validade. A cada requisição, antes de processar qualquer operação, a *auth-api* verifica se o *token* está

presente na *blacklist*, garantindo que *tokens* revogados não sejam aceitos pelo sistema. A Figura 3 ilustra o fluxo completo de autenticação e invalidação de *tokens* adotado no SisGera.

Figura 3 – Diagrama de fluxo JWT implementado no SisGera



Fonte: elaborado pelo autor.

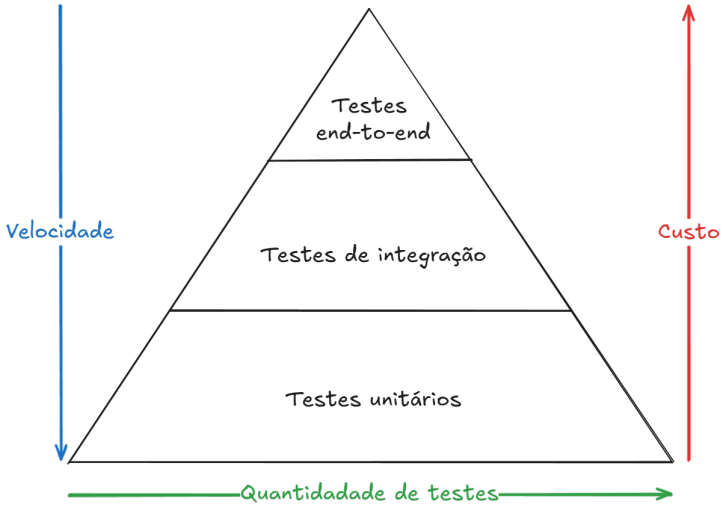
2.6 Testes automatizados

Adotar testes automatizados é essencial no desenvolvimento de *software* moderno, especialmente em arquiteturas de microsserviços, em que a independência entre serviços exige que cada unidade ofereça garantias de que tudo está funcionando corretamente antes de se integrar aos demais. Sem um conjunto de testes adequado, qualquer alteração em um serviço representa um risco potencial de regressão, difícil de rastrear.

A pirâmide de testes de Cohn (2011), vista na Figura 4, é uma forma consagrada

de organizar os diferentes tipos de teste. Na base estão os testes unitários, que verificam o comportamento de unidades isoladas do código, como funções, métodos ou classes, sem dependências externas. No meio estão os testes de integração, que validam a interação entre os componentes, como a comunicação entre um serviço e seu banco de dados. No topo estão os testes de ponta a ponta (e2e), que simulam o comportamento real do sistema do ponto de vista do usuário e são mais custosos de manter. Conceitualmente, a pirâmide orienta que a maior parte dos testes esteja na base, garantindo cobertura ampla com baixo custo de execução.

Figura 4 – Pirâmide de testes de Cohn



Fonte: elaborado pelo autor.

Em microsserviços, os testes unitários assumem papel central porque cada serviço deve ser verificável de forma autônoma. Para isso, é necessário isolar as dependências externas, como repositórios de dados e serviços de terceiros, por meio de objetos substitutos chamados de *mocks*. Um *mock* simula o comportamento de uma dependência real, permitindo que o teste foque exclusivamente na lógica do componente em análise, sem necessidade de conexão com banco de dados ou rede.

Uma métrica comumente utilizada para avaliar a abrangência dos testes é a cobertura de código, que indica a proporção das linhas ou ramificações do código exercitadas durante a execução dos testes. Embora uma cobertura elevada não garanta a ausência de defeitos, ela serve como indicador de que os principais caminhos lógicos do sistema foram considerados. No SisGera, foram implementados 259 testes unitários distribuídos entre as três APIs, cobrindo os serviços de domínio com cobertura acima de 90%, utilizando o *framework Jest* em conjunto com o módulo de testes do *NestJS*.

2.7 Considerações finais

Este capítulo apresentou os fundamentos que sustentam a refatoração do SisGera: as diferenças, vantagens e custos das arquiteturas monolítica e de microsserviços, as formas de comunicação entre serviços, a delimitação de contextos de negócio proposta pelo *Domain-Driven Design*, o modelo de autenticação *stateless* com JWT e a organização dos testes automatizados. Com esses conceitos estabelecidos, o próximo capítulo analisa o sistema legado, apresenta os problemas que motivaram a refatoração e descreve as decisões de projeto e a implementação da nova arquitetura.

3 Desenvolvimento

3.1 Considerações iniciais

Com a base conceitual estabelecida no Capítulo 2, este capítulo apresenta o desenvolvimento da nova arquitetura do SisGera. O percurso parte da análise do sistema existente e dos problemas que motivaram a refatoração, passa pelas tecnologias adotadas e pelas decisões de arquitetura e de modelagem de dados, e detalha os mecanismos transversais implementados: o controle de acesso, a auditoria automática, o monitoramento de erros, a estratégia de testes, a esteira de integração e entrega contínuas e a infraestrutura de implantação.

3.2 Análise do sistema existente

3.2.1 O que era o SisGera antes

O SisGera teve origem em trabalhos acadêmicos anteriores Arantes (2018) e foi concebido como uma aplicação *web* voltada ao apoio das atividades administrativas e operacionais de corporações de bombeiros voluntários. Seu propósito central era substituir os registros manuais de papel por um meio digital, no qual as informações de cada atendimento e dos recursos da corporação pudessem ser armazenadas e consultadas.

Em termos de escopo funcional, o sistema contemplava um conjunto amplo de módulos. O principal era o registro de ocorrências, que documentava diferentes tipos de Boletim de Ocorrência (BO), como resgate, incêndio e simplificado, nos quais eram registrados os dados do atendimento, das vítimas e da avaliação clínica realizada em campo. Complementavam o sistema funcionalidades administrativas de gestão de patrimônio, controle de produtos e estoque, registro de doações e administração de usuários e suas permissões de acesso.

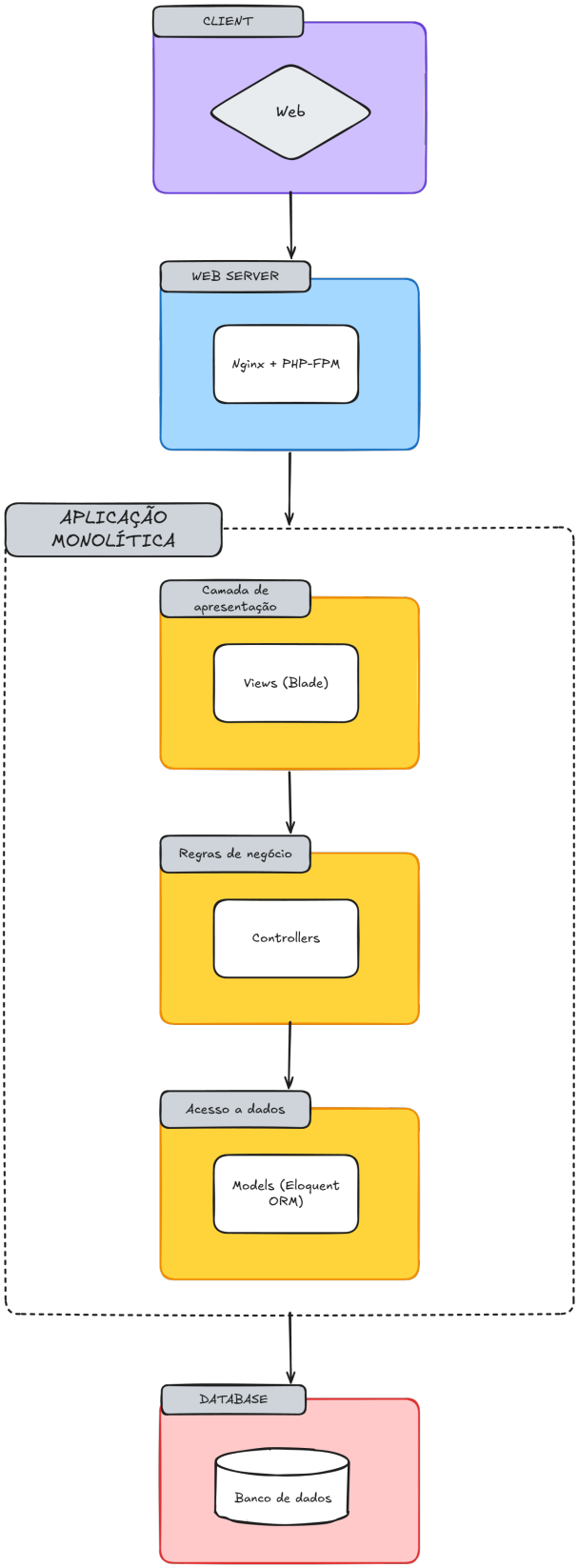
Em síntese, antes da refatoração o SisGera já era uma ferramenta funcional e em uso, que abrangia desde o registro detalhado das ocorrências até a gestão dos recursos materiais da corporação. As limitações que motivaram sua reformulação, porém, não vinham do escopo funcional, mas da forma como o sistema havia sido arquitetado e implementado, discutida nas seções seguintes.

3.2.2 Arquitetura monolítica original

Do ponto de vista arquitetural, o SisGera foi construído como uma aplicação monolítica, seguindo o modelo descrito na Seção 2, em que todas as funcionalidades residem em uma única base de código e compartilham um mesmo banco de dados (Figura 1). O sistema foi desenvolvido em PHP com o *framework* Laravel e adotava o padrão arquitetural *Model-View-Controller* (MVC), com renderização de páginas no lado do servidor por meio do motor de *templates* Blade.

A organização do código refletia diretamente as três camadas do MVC: os *Controllers* recebiam as requisições e concentravam o tratamento das regras de negócio; os *Models*, implementados com o ORM Eloquent, representavam as entidades e mediavam o acesso ao banco de dados MySQL; e as *Views*, escritas em Blade, eram responsáveis pela apresentação. Todos os módulos do sistema coexistiam nessa mesma estrutura e compartilhavam o mesmo processo de execução e o mesmo esquema de banco de dados. A Figura 5 ilustra essa organização no SisGera.

Figura 5 – Esquema da arquitetura monolítica do SisGera



Fonte: elaborado pelo autor.

Quanto à infraestrutura, o projeto era containerizado com Docker, com uma compo-

sição de contêineres que usava o servidor web Nginx e o PHP-FPM para o processamento das requisições. O repositório também contava com uma esteira de integração e entrega contínua (CI/CD) configurada via GitHub Actions por Campos (2025), responsável por automatizar a implantação da aplicação em um ambiente de hospedagem compartilhada baseado em cPanel.

Essa organização caracteriza um monólito típico, coeso e relativamente simples de implantar em sua concepção inicial, porém com todas as suas partes fortemente acopladas e dependentes entre si. Conforme detalhado na seção seguinte, foi justamente esse acoplamento, somado a outras limitações de implementação, que passou a dificultar a manutenção e a evolução do sistema.

3.2.3 Problemas identificados

A análise da base de código e da arquitetura do SisGera revelou um conjunto de limitações que, embora não comprometessem o funcionamento imediato do sistema, dificultavam progressivamente sua manutenção e evolução. Tais limitações não se restringiam à escolha pelo modelo monolítico em si, mas decorriam, sobretudo, da forma como o sistema havia sido implementado. Os principais problemas identificados são descritos a seguir.

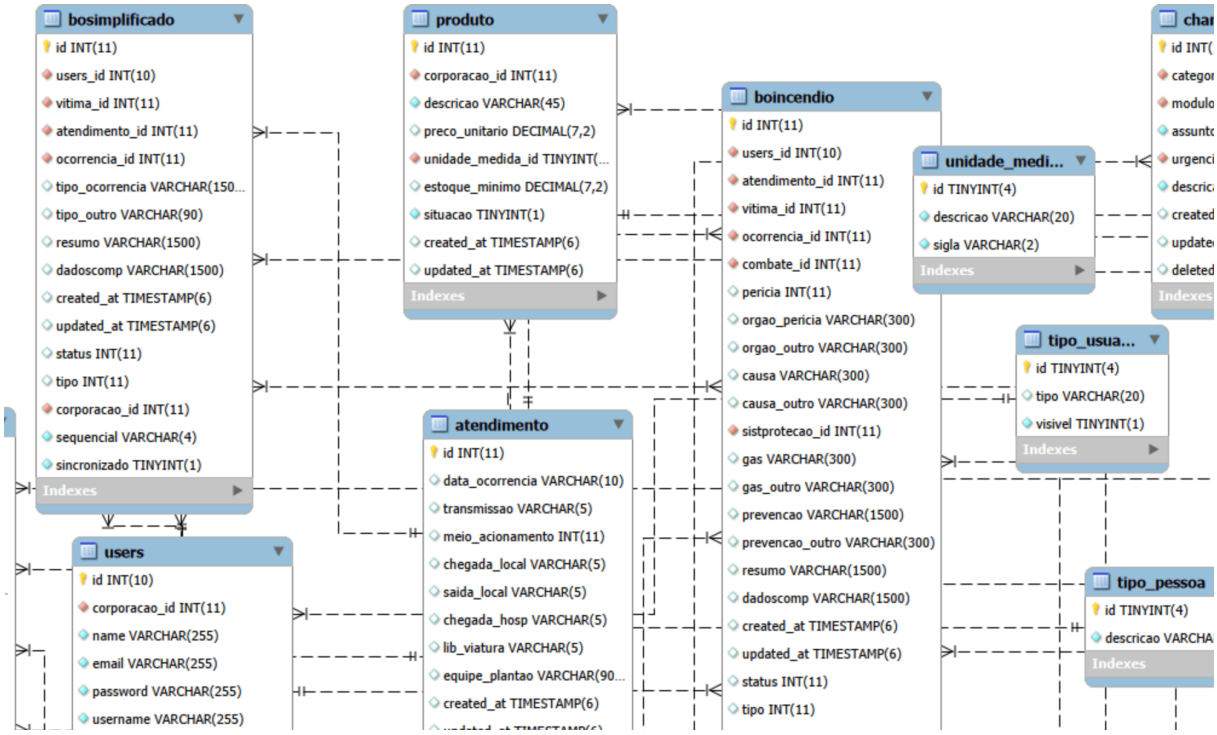
- **Forte acoplamento e ausência de camadas:** a lógica de negócio encontrava-se concentrada nos *controllers*, que acumulavam validação de dados, regras de negócio e acesso ao banco em um mesmo ponto. A ausência de uma camada de serviço (*Service Layer*) e de repositórios resultava em métodos extensos e de difícil reaproveitamento.
- **Baixa testabilidade:** como consequência direta do acoplamento, a lógica de negócio não podia ser testada de forma isolada. O projeto não dispunha de uma suíte de testes automatizados significativa, o que tornava qualquer alteração arriscada, por não haver garantias automáticas contra regressões.
- **Controle de acesso frágil:** a verificação de permissões era feita de maneira dispersa e fixa no código, por meio de comparações diretas de tipos de usuário. Não havia um modelo estruturado de papéis e permissões RBAC (*Role-Based Access Control*), o que dificultava a gestão de quem podia executar cada ação e tornava o sistema suscetível a inconsistências.
- **Dados regionais fixos no código:** informações que variam por corporação encontravam-se codificadas diretamente na aplicação. Um exemplo representativo era a lista de hospitais de destino, embutida no formulário de atendimento, o que impedia que cada corporação gerenciasse seus próprios dados e comprometia o caráter multicorporação do sistema.

- **Obsolescência tecnológica:** o sistema era baseado em versões do *framework* e da linguagem já fora do período de suporte oficial, situação que traz risco de segurança, dificulta a aplicação de correções e limita o uso de recursos mais recentes do ecossistema.

- **Escalabilidade limitada:** por ser um monólito, o sistema só podia ser escalado como um todo. Não era possível alocar recursos especificamente para os módulos de maior demanda, como o de registro de ocorrências, uma vez que todas as funcionalidades compartilhavam o mesmo processo de execução.

- **Modelagem de dados inadequada:** o banco de dados do sistema legado reunia um grande número de tabelas e relacionamentos em um único esquema. A Figura 6 apresenta um recorte do seu diagrama entidade-relacionamento; o diagrama completo, cuja extensão e densidade evidenciam a ausência de uma organização clara, encontra-se no Apêndice A. Essa falta de organização dificulta a compreensão e a manutenção do esquema. Uma análise mais detalhada revelou problemas pontuais de tipagem e de normalização, discutidos a seguir.

Figura 6 – Recorte do diagrama entidade-relacionamento do banco de dados do sistema legado



Fonte: elaborado pelo autor, com base na estrutura de banco de dados de Arantes (2018).

Além dos problemas citados por Campos (2025), o esquema do banco apresentava inconsistências de tipagem e de normalização. Diversos campos numéricos eram

armazenados como texto, e dados estruturados eram persistidos de forma não relacional, o que comprometia a integridade, a validação e a possibilidade de realizar consultas e estatísticas confiáveis sobre os dados.

Esse problema de modelagem fica evidente em alguns exemplos. Na tabela `sinais-vitais` (Figura 7), as colunas eram definidas como campos de texto (`varchar`) nuláveis, de modo que era possível salvar dados completamente vazios e armazenar valores heterogêneos e inconsistentes, como `"+29"`, `"130x10"` ou mesmo `"000x00"`. Tal escolha impedia validações automáticas, cálculos e a extração de indicadores clínicos. Já na tabela `vitima_boresgate` (Figura 8), os procedimentos realizados eram gravados em um único campo, como um vetor serializado pela linguagem (por exemplo, `a:3:{i:0;s:1:"4";...}`), em vez de representados por uma tabela de relacionamento. Na refatoração, esses pontos foram corrigidos com a adoção de tipos adequados, relacionamentos explícitos e enumerações, conforme detalhado nas seções seguintes.

Figura 7 – Tabela sinaisvitais com valores inconsistentes

id	pressao_arterial	freq_cardiaca	freq_respiratoria	sp02	glicemia
65	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
66	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
67	110x70	120	14	96	*NULL*
68	110x70	+29	138	96	*NULL*
69	*NULL*	14	110	95	*NULL*

Fonte: elaborado pelo autor.

Figura 8 – Tabela `vitima_boresgate` com valores sem padrão

proc_efetuados	id	nome	data_nasc
a:4: {i:0;s:1:"1";i:1;s:1:"4";i:2;s:1:"5";i:3;s:1:"...	1	Maxuel [redacted]	24/04/1996
a:7: {i:0;s:1:"1";i:1;s:1:"4";i:2;s:1:"5";i:3;s:1:"...	3	Maciel [redacted]	08/04/1986
a:7: {i:0;s:1:"1";i:1;s:1:"4";i:2;s:1:"5";i:3;s:1:"...	4	Inez da [redacted]	29/02/1964
a:6: {i:0;s:1:"1";i:1;s:1:"4";i:2;s:1:"5";i:3;s:1:"...	5	Luiz Felipe [redacted]	20/10/2013

Fonte: elaborado pelo autor.

Outro ponto recorrente era o uso de tabelas para representar dados de domínio estáticos, que não variam entre corporações e raramente são alterados. O banco do sistema antigo continha diversas tabelas desse tipo, como `tipo_doacao`, `tipo_pessoa`,

`unidade_medida`, `uf`, `tipo_sanguineo`, `estado_civil` e `escolaridade`, cada uma armazenando apenas um identificador e uma descrição. Embora funcional, essa abordagem exigia junções (*joins*) constantes para recuperar valores que, na prática, são constantes do domínio. Na refatoração, esses conjuntos foram representados como enumerações (*enums*) diretamente no código da aplicação, o que eliminou tabelas desnecessárias, reduziu o número de junções e tornou esses valores explícitos e fortemente tipados. Optou-se por manter como tabelas apenas os dados que efetivamente variam por corporação ou que demandam gestão pelo usuário, como é o caso dos hospitais.

Em conjunto, esses problemas mostram que as principais barreiras do SisGera estavam relacionadas a atributos de qualidade como manutenibilidade, testabilidade e escalabilidade. Mais do que justificar a simples adoção de uma nova arquitetura, esse diagnóstico orientou as decisões de projeto da refatoração, apresentadas na seção seguinte.

3.2.4 Motivação para a refatoração

Como apresentado nas seções anteriores, as limitações do SisGera não estavam em seu escopo funcional, mas na forma como havia sido arquitetado e implementado. Foram esses problemas concretos, e não a mera intenção de reescrever o sistema, que motivaram e orientaram a refatoração. Cada limitação identificada deu origem a um objetivo de projeto correspondente.

O forte acoplamento e a ausência de camadas motivaram a separação das responsabilidades em serviços independentes e a adoção de uma arquitetura em camadas bem definidas (controlador, serviço e repositório), o que favorece a manutenibilidade e o reaproveitamento de código. Os problemas de modelagem orientaram o redesenho do banco de dados, com tipagem adequada, relacionamentos explícitos e uso de enumerações. O controle de acesso frágil motivou a implementação de um modelo estruturado de papéis e permissões (RBAC), e a obsolescência tecnológica justificou a migração para uma plataforma moderna e com suporte ativo.

A escolha pela arquitetura de microsserviços, em particular, fundamentou-se nas características discutidas no Capítulo 2. A divisão do sistema em serviços autônomos permite desenvolver, implantar e escalar cada parte de forma independente, além de favorecer o isolamento de falhas, de modo que um problema em um serviço não comprometa os demais. Para um sistema com módulos de naturezas distintas, como o registro de ocorrências e a gestão de inventário, essa autonomia é um ganho importante em organização e evolução.

Reconhece-se, contudo, que a arquitetura de microsserviços introduz custos e desafios próprios, como a maior complexidade operacional e a necessidade de comunicação

entre serviços, conforme apontado no Capítulo 2. A decisão por essa abordagem levou em conta esse *trade-off* e considerou-a adequada aos objetivos deste trabalho, que se propõe a conduzir e documentar a refatoração do SisGera como um estudo de caso da migração de um sistema monolítico para uma arquitetura de microsserviços.

3.3 Tecnologias utilizadas

Esta seção apresenta as principais tecnologias adotadas no desenvolvimento da nova arquitetura do SisGera, bem como as justificativas para sua escolha.

3.3.1 Linguagem e framework

O *back-end* foi desenvolvido em TypeScript (Microsoft, 2025), um superconjunto do JavaScript que adiciona tipagem estática e favorece a detecção de erros ainda em tempo de desenvolvimento, além de melhorar a manutenibilidade do código. Como *framework*, adotou-se o NestJS (NestJS, 2025), que oferece uma arquitetura modular baseada em injeção de dependências e organiza a aplicação em camadas bem definidas (controladores, serviços e provedores), o que se alinha ao objetivo de reduzir o acoplamento identificado no sistema legado.

3.3.2 Banco de dados e ORM

A persistência foi implementada com o sistema gerenciador de banco de dados relacional PostgreSQL (The PostgreSQL Global Development Group, 2025), e o acesso aos dados é intermediado pelo Prisma (Prisma, 2025), um *Object-Relational Mapper* (ORM) que define o esquema do banco de forma declarativa e gera um cliente fortemente tipado, além de gerenciar a evolução do esquema por meio de *migrations* versionadas.

3.3.3 Autenticação

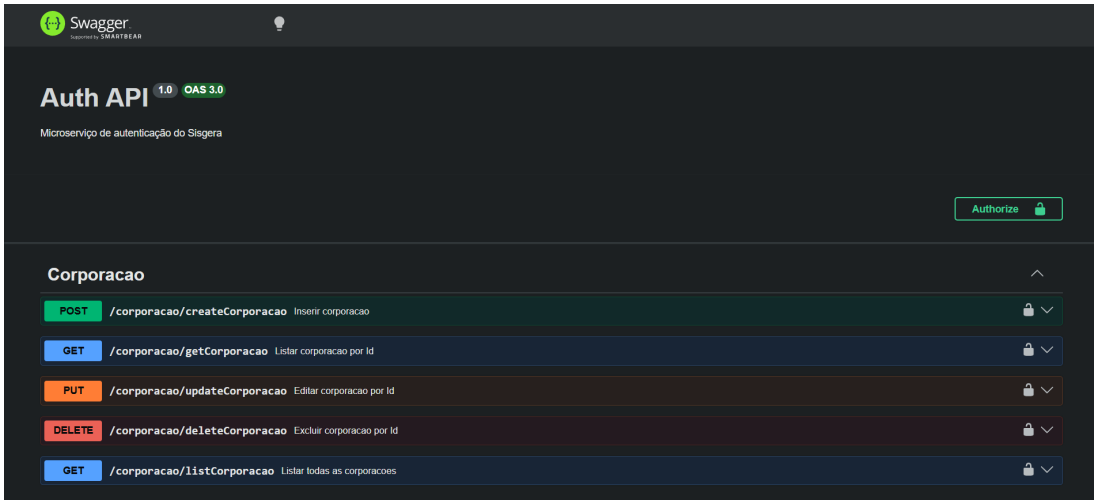
O controle de autenticação utiliza *JSON Web Tokens* (JWT) (Auth0, 2025), em conjunto com a biblioteca Passport (Passport, 2025) para a validação dos *tokens* e com o algoritmo *bcrypt* (npm, 2025) para o armazenamento seguro das senhas por meio de *hash*. O banco de dados em memória Redis (Redis, 2025) é empregado para manter uma lista de revogação (*blacklist*) de *tokens*, que permite invalidar sessões antes do vencimento natural do *token*. Optou-se pelo Redis por ser um armazenamento em memória, o que garante consultas rápidas a cada requisição, e por oferecer expiração automática de chaves, usada para descartar o *token* da *blacklist* assim que sua validade original se encerra.

3.3.4 Validação e documentação

A validação dos dados de entrada é realizada de forma declarativa com a biblioteca *class-validator* (class-validator, 2025), aplicada diretamente aos modelos de entrada.

A documentação das APIs é gerada automaticamente no padrão OpenAPI por meio do Swagger (SmartBear Software, 2025), que disponibiliza uma interface interativa para teste e consulta dos *endpoints* das aplicações, como mostra a Figura 9.

Figura 9 – Interface interativa de documentação gerada pelo Swagger



Fonte: elaborado pelo autor.

3.3.5 Testes e qualidade de código

Os testes automatizados foram escritos com o *framework* Jest (Meta Open Source, 2025). A padronização e a qualidade do código são mantidas com as ferramentas ESLint (OpenJS Foundation, 2025) e Prettier (Prettier, 2025). Utilizou-se ainda o Husky (Husky, 2025) para automatizar a execução da suíte de testes antes de cada envio de código ao repositório, o que previne a introdução de código que não esteja de acordo com as regras de negócio.

3.3.6 Containerização

O ambiente de execução é containerizado com Docker (Docker, Inc., 2025) e orquestrado localmente pelo Docker Compose, que provê os serviços de banco de dados e de *cache* de forma isolada e reproduzível, o que simplifica a configuração do ambiente de desenvolvimento.

3.3.7 Infraestrutura, publicação e monitoramento

O código-fonte é versionado com Git (Software Freedom Conservancy, 2025) e hospedado no GitHub, que centraliza o repositório único (*monorepo*) dos três serviços. Sobre essa hospedagem, a automação de integração e entrega contínua é realizada pelo GitHub Actions (GitHub, 2025), que executa verificações automáticas a cada envio de código e coordena as implantações.

A execução das aplicações em nuvem utiliza a plataforma como serviço (*Platform as a Service*, PaaS) Render (Render, 2025), responsável por hospedar os microsserviços, o banco de dados PostgreSQL e o Redis em ambiente gerenciado. O monitoramento em tempo de execução é feito com o Sentry (Functional Software, Inc., 2025), serviço de rastreamento de erros (*error tracking*) que captura de forma centralizada as exceções ocorridas em cada microsserviço.

O detalhamento do processo de integração e entrega contínuas, da configuração de infraestrutura e do monitoramento de erros é apresentado nas seções seguintes deste capítulo.

3.4 Arquitetura proposta

Definidas as tecnologias, esta seção apresenta a nova arquitetura do SisGera e descreve como o sistema foi decomposto em serviços e como esses serviços se organizam internamente.

3.4.1 Decomposição em microsserviços

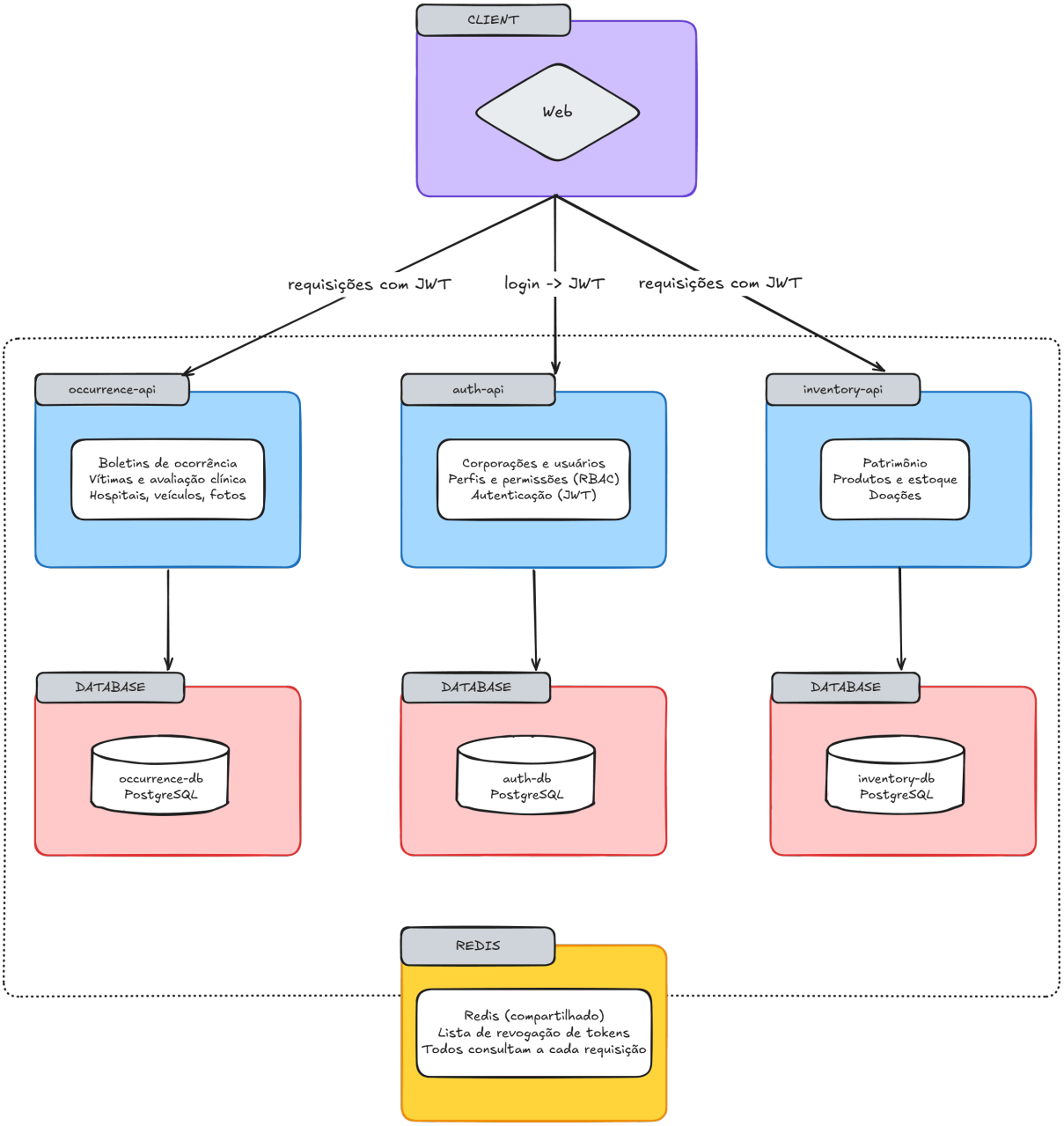
A decomposição do sistema seguiu a noção de *bounded contexts* discutida no Capítulo 2: cada serviço corresponde a um contexto de negócio do domínio, identificado a partir dos módulos do sistema legado. O resultado foi a divisão do *back-end* em três microsserviços:

- **auth-api:** concentra as responsabilidades transversais de identidade e acesso de usuário. É responsável pelo cadastro de corporações, usuários, perfis e permissões, pela autenticação (emissão e revogação de *tokens*) e pelo modelo de controle de acesso baseado em papéis (RBAC), consumido pelos demais serviços;
- **occurrence-api:** implementa o núcleo operacional do sistema, correspondente ao registro de ocorrências. Abrange os boletins de ocorrência, as vítimas e suas avaliações clínicas, os encaminhamentos hospitalares, os veículos envolvidos, as fotos das ocorrências e o cadastro de hospitais de cada corporação;

- **inventory-api:** reúne as funcionalidades administrativas de gestão de recursos materiais da corporação, contemplando patrimônio, produtos, movimentações de estoque e doações.

A Figura 10 apresenta a visão geral da nova arquitetura, em contraste com a organização monolítica da Figura 5.

Figura 10 – Esquema da nova arquitetura de microsserviços do SisGera



Fonte: elaborado pelo autor.

Cada serviço é uma aplicação NestJS independente, com seu próprio processo de execução, suas próprias dependências e sua própria porta de rede (3001, 3002 e 3003,

respectivamente), e pode ser desenvolvido, testado, implantado e escalado de forma isolada dos demais.

3.4.2 Banco de dados por serviço

Em conformidade com o padrão *database per service*, cada microsserviço possui seu próprio banco de dados PostgreSQL, com esquema definido e versionado de forma independente por meio das *migrations* do Prisma. O banco do auth-api armazena as entidades de identidade e acesso (corporações, usuários, perfis e permissões), o do occurrence-api armazena as entidades do domínio de ocorrências, e o do inventory-api, as de gestão de bens e patrimônio da corporação.

A intenção inicial do projeto era que cada esquema residisse em uma instância de banco de dados própria, o que de fato ocorre no ambiente de desenvolvimento local. No ambiente de homologação, no entanto, por uma decisão de custo detalhada na Seção 3.11, os três esquemas passaram a compartilhar uma única instância PostgreSQL, sem prejuízo do isolamento lógico entre eles.

Essa separação elimina, por construção, o acoplamento via banco de dados identificado no sistema legado, em que todos os módulos compartilhavam um mesmo esquema: nenhum serviço tem acesso direto aos dados de outro, e alterações no esquema de um serviço não afetam os demais. Como consequência, entidades de outros contextos são referenciadas apenas por seus identificadores, sem chaves estrangeiras entre bancos, e cabe à camada de aplicação garantir a consistência dessas referências, um dos *trade-offs* da abordagem discutidos no Capítulo 2.

3.4.3 Comunicação e autenticação entre serviços

Uma decisão central do projeto foi evitar a comunicação síncrona direta entre os serviços, logo os microsserviços criados não realizam chamadas HTTP entre eles. A integração ocorre por meio do *token* JWT emitido pelo auth-api no momento da autenticação.

Com base no modelo de autenticação *stateless* apresentado no Capítulo 2, o *payload* do *token* carrega, de forma assinada, as informações de que os demais serviços necessitam para autorizar cada requisição: o identificador do usuário, sua corporação, seu perfil e a lista de permissões. Assim, ao receber uma requisição, o occurrence-api e o inventory-api validam a assinatura do *token* com o segredo compartilhado e extraem dessas *claims* tudo o que precisam, sem consultar o auth-api. É assim que a autonomia entre os serviços foi mantida, uma vez que uma indisponibilidade do serviço de autenticação não impede que os demais continuem atendendo usuários já autenticados no sistema.

Do ponto de vista lógico e arquitetural, a única dependência de infraestrutura

compartilhada entre os serviços é uma instância Redis, utilizada exclusivamente para a lista de revogação de tokens: no *logout*, o auth-api registra o token na *blacklist* com tempo de expiração equivalente à sua validade restante, e todos os serviços consultam essa lista ao validar cada requisição. Esse mecanismo mitiga a principal limitação da autenticação *stateless*, que é a impossibilidade de invalidar um token antes de seu vencimento natural.

Essa afirmação refere-se ao acoplamento entre os serviços em nível de arquitetura, e não à infraestrutura física do ambiente de homologação, que, por uma decisão de custo detalhada na Seção 3.11, também compartilha uma única instância de banco de dados PostgreSQL entre os três esquemas. Nenhum serviço, contudo, acessa o esquema de outro: o compartilhamento da instância é uma questão de hospedagem, não de acoplamento lógico entre os domínios.

3.4.4 Organização interna em camadas

Os três serviços adotam a mesma organização em camadas, o que responde diretamente ao problema de concentração de responsabilidades identificado na Seção 3.2.3. O fluxo de uma requisição atravessa três camadas com papéis bem definidos:

- **Controlador (*controller*):** recebe a requisição HTTP, aciona a validação declarativa dos dados de entrada e delega o processamento à camada de serviço, sem conter regras de negócio;
- **Serviço (*service*):** concentra as regras de negócio, como as verificações de existência, as regras de autorização por corporação e as validações de estado, de forma independente dos detalhes de HTTP e de banco de dados;
- **Repositório (*repository*):** encapsula todo o acesso ao banco de dados por meio do Prisma, isola as consultas em um único ponto e permite que a camada de serviço seja testada com repositórios simulados (*mocks*) por meio de testes unitários.

A validação da entrada e o formato das respostas são definidos por modelos dedicados (objetos de transferência de dados, ou *DTOs*), que estabelecem o contrato de cada *endpoint* de forma separada da lógica de negócio.

Essa separação, combinada à injeção de dependências do NestJS, viabiliza a testabilidade que faltava ao sistema legado: as regras de negócio são exercitadas por testes unitários que substituem os repositórios por simulações, sem necessidade de um banco de dados real.

3.4.5 Organização do repositório

Os três serviços residem em um único repositório (*monorepo*), organizado em um diretório `services/` com uma pasta autocontida por serviço. Embora cada serviço seja implantado de forma independente, essa organização centraliza o acompanhamento do projeto, simplifica a configuração do ambiente de desenvolvimento, no qual um único Docker Compose provisiona os três bancos, o Redis e as três aplicações, e favorece a continuidade do sistema por futuros trabalhos acadêmicos, que encontram todo o código em um único lugar.

Uma consequência dessa independência entre os serviços é a replicação de algumas partes de código, como a estratégia de validação de *tokens* JWT e os *guards* de autorização, presentes de forma idêntica nos três serviços. Optou-se por essa duplicação controlada para preservar a autonomia de implantação de cada serviço e reduzir a complexidade de gerenciamento de dependências internas, ao custo de propagar manualmente eventuais alterações nessas pequenas partes do código.

3.5 Modelagem de dados

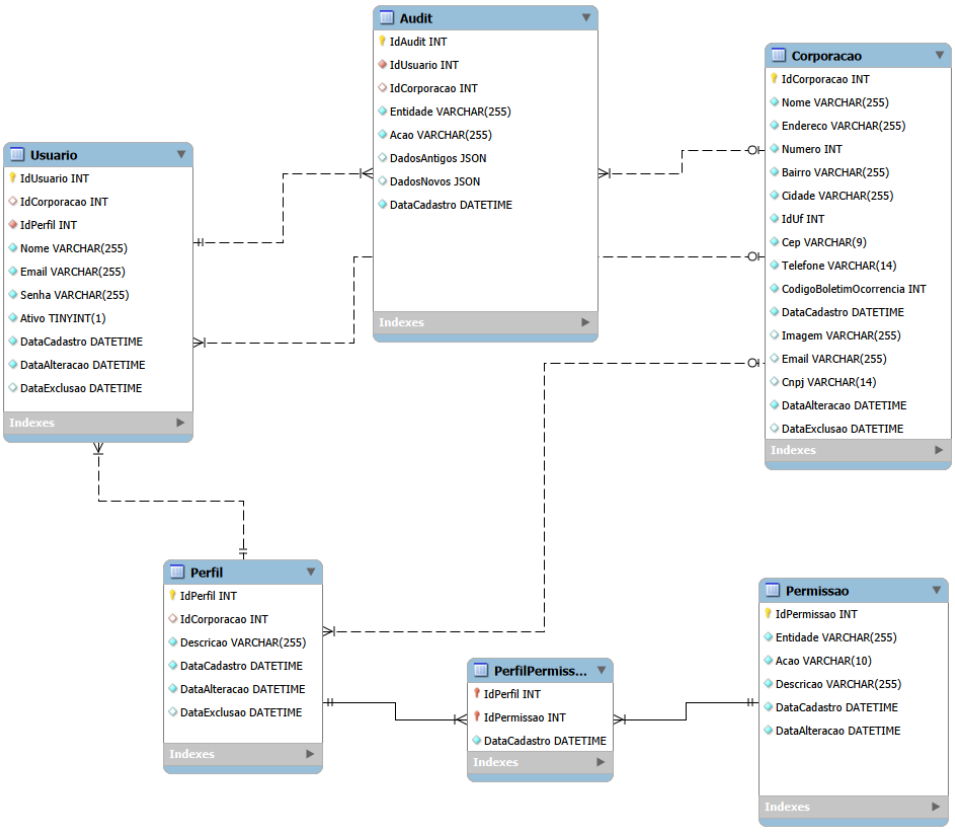
Um dos principais problemas identificados no sistema legado, discutido na Seção 3.2.3, dizia respeito à modelagem de dados. Esta seção apresenta como o esquema foi redesenhado na nova arquitetura e retoma cada uma das deficiências apontadas, com a solução adotada para cada uma.

3.5.1 Divisão e independência dos esquemas

Em coerência com o padrão *database per service* descrito na Seção 3.4, o modelo de dados foi dividido em três esquemas independentes, um para cada microsserviço, definidos de forma declarativa e versionados separadamente por meio das *migrations* do Prisma. Cada esquema contém apenas as entidades do seu próprio contexto de negócio: identidade e acesso na auth-api, ocorrências na occurrence-api e gestão de patrimônio na inventory-api.

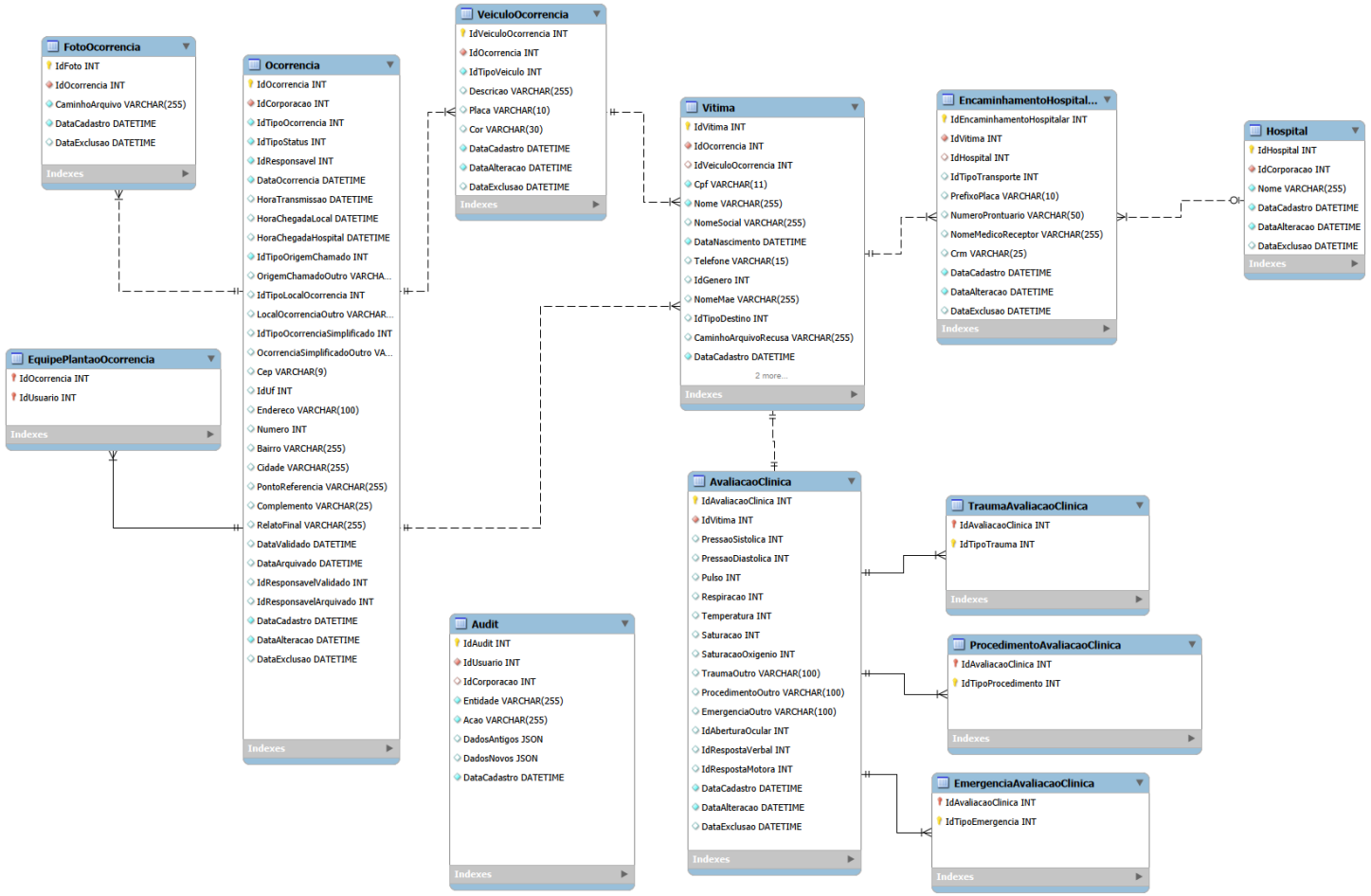
Essa divisão elimina o acoplamento por esquema de banco de dados e produz esquemas individualmente menores e mais coesos que o modelo único do sistema legado. Ao contrário da densidade do diagrama da Figura 6, cada serviço passa a ter um modelo compreensível de forma isolada, como mostram as Figuras 11, 12 e 13.

Figura 11 – Diagrama entidade-relacionamento da auth-api



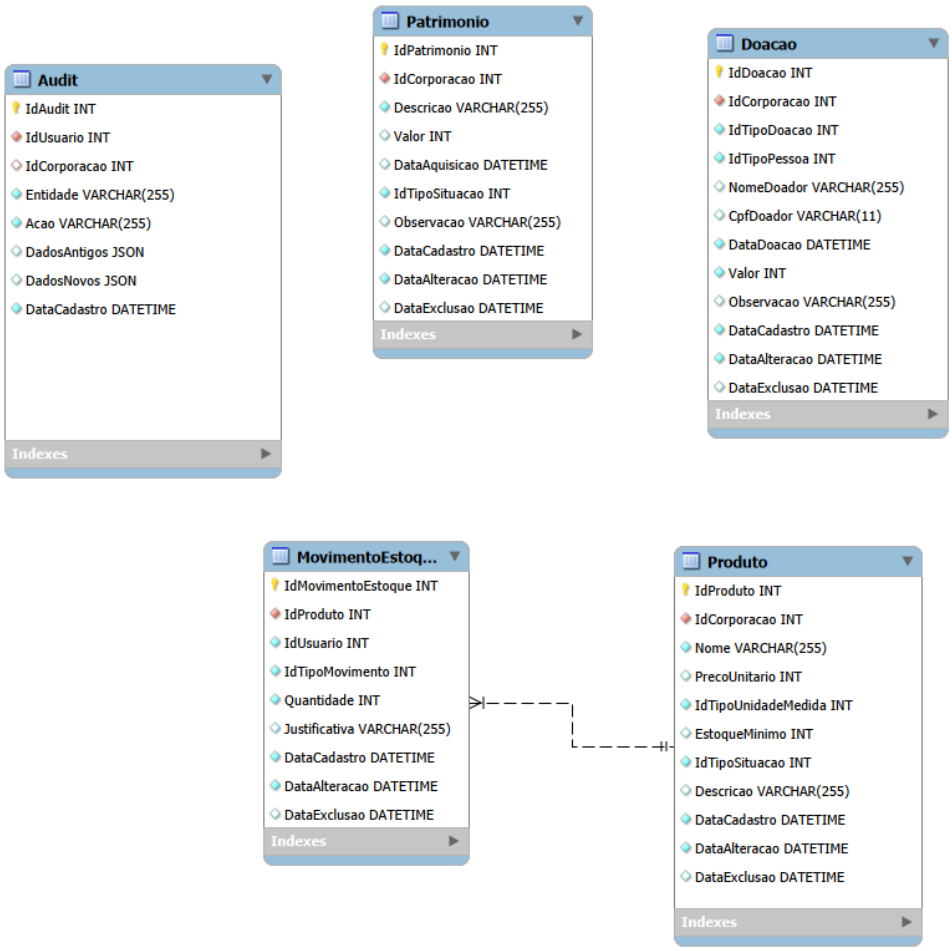
Fonte: elaborado pelo autor.

Figura 12 – Diagrama entidade-relacionamento da occurrence-api



Fonte: elaborado pelo autor.

Figura 13 – Diagrama entidade-relacionamento da inventory-api



Fonte: elaborado pelo autor.

3.5.2 Correções de tipagem e normalização

Os problemas de tipagem e de normalização apontados na Seção 3.2.3 foram corrigidos com a adoção de tipos adequados e de relacionamentos entre tabelas. Os campos que antes armazenavam valores numéricos como texto passaram a ter tipos apropriados, como na entidade `AvaliacaoClinica`, em que os sinais vitais como pressão arterial, pulso, respiração, temperatura e saturação são agora representados por campos inteiros, o que impede o armazenamento de valores heterogêneos e viabiliza validações, cálculos e a extração de indicadores clínicos, algo inviável no modelo anterior.

Os dados estruturados que eram persistidos de forma não relacional também foram normalizados. Os procedimentos, traumas e emergências que, no sistema legado, eram gravados em um único campo como um vetor serializado pela linguagem passaram a ser representados por tabelas de relacionamento dedicadas (`TraumaAvaliacaoClinica`, `EmergenciaAvaliacaoClinica` e `ProcedimentoAvaliacaoClinica`). Cada item selecionado em uma avaliação clínica corresponde agora a uma linha nessas tabelas, o que torna os dados consultáveis e íntegros. Do mesmo modo, a composição da equipe de plantão

de uma ocorrência, uma relação de muitos para muitos entre ocorrências e usuários, é modelada explicitamente pela tabela `EquipePlantaoOcorrencia`.

A normalização também se refletiu na decomposição de entidades. Informações que possuem cardinalidade um para um (1:1) com a vítima, como a avaliação clínica e o encaminhamento hospitalar, foram separadas em entidades próprias (`AvaliacaoClinica` e `EncaminhamentoHospitalar`), relacionadas à `Vitima` por meio de chaves estrangeiras únicas, em vez de acumuladas em uma única tabela.

3.5.3 Enumerações e dados de domínio

Conforme antecipado na Seção 3.2.3, o sistema legado utilizava diversas tabelas apenas para representar dados de domínio estáticos, que não variam entre corporações e raramente são alterados, o que exigia junções constantes para recuperar valores que, na prática, são constantes do domínio. Na refatoração, esses conjuntos foram representados como enumerações (*enums*) diretamente no código da aplicação. Valores como as unidades da federação, o gênero, os tipos de ocorrência, de origem do chamado, de veículo, de transporte, de movimento de estoque e de doação, além das escalas clínicas de abertura ocular e de respostas verbal e motora, tornaram-se enumerações fortemente tipadas, o que eliminou tabelas desnecessárias e reduziu o número de junções no banco de dados.

Em contrapartida, optou-se por manter como tabela apenas os dados que efetivamente variam por corporação ou que demandam gestão pelo usuário. É o caso da entidade `Hospital`, que no sistema legado encontrava-se fixa no código do formulário de atendimento, o que comprometia o caráter multicorporação do sistema. Na nova modelagem, cada corporação gerencia seus próprios hospitais de destino, o que corrige a limitação de dados regionais fixos no código.

3.5.4 Convenções do novo modelo

Algumas convenções foram aplicadas de forma uniforme a todo o modelo do sistema. O caráter multicorporação é garantido pelo campo `idCorporacao`, presente nas entidades de negócio e utilizado para isolar os dados de cada corporação. Nas entidades filhas de uma agregação, como a vítima em relação à ocorrência, o vínculo com a corporação é herdado da entidade raiz, o que evita redundância.

Adotou-se ainda a exclusão lógica (*soft delete*) de forma padronizada: em vez de remover fisicamente os registros, o campo `dataExclusao` marca o momento da exclusão e preserva o histórico. Os campos `dataCadastro` e `dataAlteracao`, gerenciados automaticamente pelo Prisma, registram a criação e a última modificação de cada linha. Os perfis e usuários admitem `idCorporacao` nulo para representar entidades globais, recurso explorado pelo modelo de controle de acesso descrito na Seção 3.6.

Por fim, os campos destinados a arquivos, como o caminho da imagem da corporação e do termo de recusa da vítima, armazenam apenas uma referência lógica ao arquivo, de modo que a integração com um serviço de armazenamento de objetos (*object storage*) possa ser adicionada futuramente sem alteração do esquema, conforme discutido na Seção 5.1.

3.6 Controle de acesso

Entre os problemas descritos na Seção 3.2.3, o controle de acesso frágil foi um dos pontos tratados na refatoração. No sistema legado, a verificação de permissões era feita de forma dispersa e fixa no código, por meio de comparações diretas de tipos de usuário. Em seu lugar, adotou-se um modelo estruturado de papéis e permissões (*Role-Based Access Control*, RBAC), descrito nesta seção.

3.6.1 Modelo de papéis e permissões

O controle de acesso apoia-se em três entidades, definidas no esquema da auth-api. A entidade `Permissao` é uma autorização elementar, expressa pelo par entidade e ação, como visualizar ocorrências ou editar produtos. A entidade `Perfil` é um papel atribuível a usuários, como Socorrista ou Coordenador Geral. A associação entre as duas é feita pela entidade `PerfilPermissao`, que define quais permissões cada perfil concede. Cada usuário está vinculado a um único perfil e herda dele o conjunto de permissões.

A granularidade por entidade e ação permite descrever com precisão o que cada papel pode fazer, ao contrário da verificação por tipo de usuário do sistema legado. Uma mesma permissão pode ainda reger o acesso a funcionalidades correlatas: as ações sobre vítimas, veículos e fotos de uma ocorrência, por exemplo, são governadas pela permissão da entidade `Ocorrencia`, porque pertencem ao mesmo contexto de negócio.

3.6.2 Perfis globais e escopo por corporação

Os perfis podem ser globais ou específicos de uma corporação, distinção viabilizada pelo campo `idCorporacao` opcional na entidade `Perfil`, mencionado na Seção 3.5. Os perfis padrão do sistema, como Socorrista, Chefe de Equipe e Coordenador Geral, são globais e compartilhados por todas as corporações, e são inseridos de forma idempotente durante a carga inicial de dados (*seed*). Há ainda um perfil de administrador, com acesso irrestrito, também global.

Essa distinção sustenta o caráter multicorporação do sistema. Um usuário comum está sempre vinculado a uma corporação e só enxerga os dados dela; o administrador, por não pertencer a nenhuma corporação específica (campo `idCorporacao` nulo), tem visão

global. Esse escopo é aplicado de forma sistemática na camada de serviço, como detalhado adiante.

3.6.3 Autorização distribuída via *token*

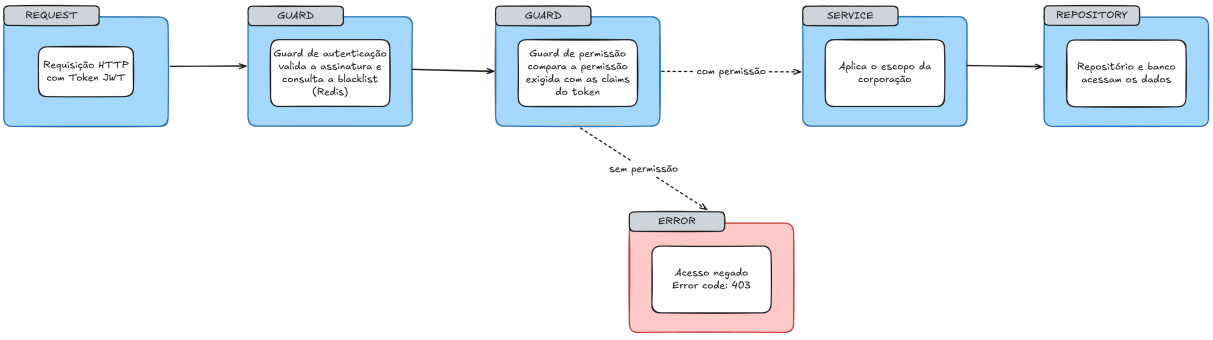
Conforme apresentado na Seção 3.4, o conjunto de permissões do usuário é embutido no *payload* do *token* JWT emitido pela auth-api no momento da autenticação. Assim, ao receber uma requisição, qualquer serviço decide se o usuário está autorizado apenas inspecionando as permissões contidas no *token*, sem uma nova consulta ao serviço de autenticação.

É esse desenho que torna a autorização viável em uma arquitetura de microsserviços sem introduzir acoplamento. Embora o modelo de papéis e permissões seja definido e administrado apenas na auth-api, sua aplicação ocorre de forma autônoma em cada serviço, a partir da informação que já viaja assinada no *token*.

3.6.4 Aplicação declarativa e isolamento de dados

A verificação de permissões é declarativa. Cada *endpoint* que exige autorização é anotado com um decorador que indica a permissão necessária, no formato entidade e ação. Um interceptador de rota global (*guard*), executado antes do método do controlador, compara a permissão exigida com as permissões presentes no *token* e rejeita a requisição quando o usuário não a possui. Assim, a regra de autorização fica concentrada em um único ponto, e não dispersa pelo código como no sistema legado. A Figura 14 resume esse fluxo, desde a chegada da requisição até o acesso ao banco, incluindo a verificação de permissão e o escopo por corporação.

Figura 14 – Fluxo de autorização de uma requisição



Fonte: elaborado pelo autor.

A essa verificação soma-se o isolamento de dados por corporação, aplicado na camada de serviço. Para usuários vinculados a uma corporação, as consultas e operações são restringidas automaticamente ao respectivo `idCorporacao`, seja qual for o parâmetro

informado na requisição, o que impede o acesso a dados de outras corporações. Para o administrador, de escopo global, essa restrição não se aplica, e ele pode operar sobre todas as corporações. Juntos, esses dois mecanismos, autorização por permissão e isolamento por corporação, substituem o controle de acesso frágil do sistema original.

3.7 Auditoria automática

Uma necessidade que não era atendida pelo sistema legado é o registro de quem alterou cada dado e quando. Como o SisGera deverá ser mantido e evoluído por outros trabalhos acadêmicos, adotou-se um requisito de projeto adicional para a auditoria: ela deveria ser impossível de esquecer, ou seja, funcionar sem depender de o desenvolvedor lembrar de instrumentar cada operação.

3.7.1 Registro por interceptação de mutações

Para atender a esse requisito, descartou-se a instrumentação manual de cada serviço, sujeita a omissões, em favor de um mecanismo centralizado. A auditoria foi implementada com um *middleware* do Prisma, que intercepta toda operação de escrita no banco (criação, atualização e remoção) antes que ela seja executada. Assim, qualquer alteração persistida por qualquer repositório é registrada automaticamente, sem que os controladores ou serviços precisem conter código de auditoria.

Para cada mutação, o *middleware* grava uma entrada na tabela `Audit` com a entidade afetada, a ação realizada, os dados anteriores, os dados novos, o usuário responsável, a corporação e o instante do evento. A ação é classificada como cadastro, edição ou exclusão. Como o projeto usa exclusão lógica em vez de remoção física, uma atualização que preenche o campo `dataExclusao` é reconhecida e registrada como exclusão. A própria tabela `Audit` é excluída da interceptação, para evitar que o registro de auditoria gere novas auditorias em cadeia.

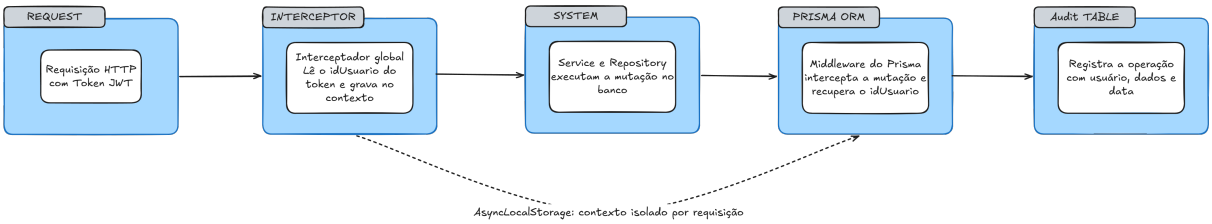
3.7.2 Propagação do usuário autenticado

O principal desafio técnico dessa abordagem é que o *middleware* do Prisma opera na camada de acesso a dados, sem conhecimento direto da requisição HTTP e, portanto, de qual usuário originou a operação. Para resolver isso sem passar o identificador do usuário manualmente por todas as camadas, utilizou-se o recurso de armazenamento de contexto por requisição (*AsyncLocalStorage*) do Node.js.

Um interceptador global, executado no início de cada requisição, lê o identificador do usuário presente no *token* JWT já validado e o armazena em um contexto isolado, associado àquela requisição. Quando o *middleware* do Prisma é acionado para registrar

uma auditoria, ele recupera o usuário desse contexto. Dessa forma, o vínculo entre a operação de banco e o usuário responsável é mantido de ponta a ponta, sem acoplar as camadas intermediárias ao mecanismo de auditoria. A Figura 15 ilustra esse percurso, do recebimento da requisição até a gravação do registro na tabela `Audit`, destacando o contexto isolado que liga o interceptador ao *middleware* do Prisma.

Figura 15 – Fluxo da auditoria automática por interceptação de mutações



Fonte: elaborado pelo autor.

3.7.3 Consulta e restrição de acesso

A consulta à trilha de auditoria é exposta por um *endpoint* dedicado, protegido pelo mesmo modelo de controle de acesso descrito na Seção 3.6. Criou-se para isso uma permissão específica, `Audit/Visualizar`, atribuída apenas aos perfis de Coordenador Geral e de administrador. A consulta também respeita o escopo por corporação: o Coordenador Geral visualiza somente os registros da própria corporação, enquanto o administrador tem acesso irrestrito.

Por se tratar de uma arquitetura com banco de dados por serviço, cada microsserviço possui sua própria tabela `Audit` e sua própria cópia desse mecanismo, aplicado de forma idêntica nos três serviços, seguindo a mesma estratégia de duplicação controlada discutida na Seção 3.4.

3.8 Observabilidade e monitoramento de erros

Em um sistema monolítico, o diagnóstico de falhas se resume a inspecionar um único registro de execução. Em uma arquitetura de microsserviços, cada serviço gera seus próprios registros de forma independente, o que torna a identificação e o acompanhamento de erros mais difíceis. Para lidar com isso, adotou-se uma ferramenta de rastreamento de erros (*error tracking*) que centraliza as falhas dos três serviços.

3.8.1 Escolha da ferramenta

Optou-se pelo Sentry (Functional Software, Inc., 2025), um serviço gerenciado de captura de erros. A alternativa de montar uma infraestrutura própria de agregação de logs

foi descartada por adicionar complexidade operacional desproporcional ao porte do sistema. A escolha por um serviço gerenciado é coerente com as demais decisões de infraestrutura do trabalho, como o Render e o GitHub Actions, que também transferem a operação para plataformas externas.

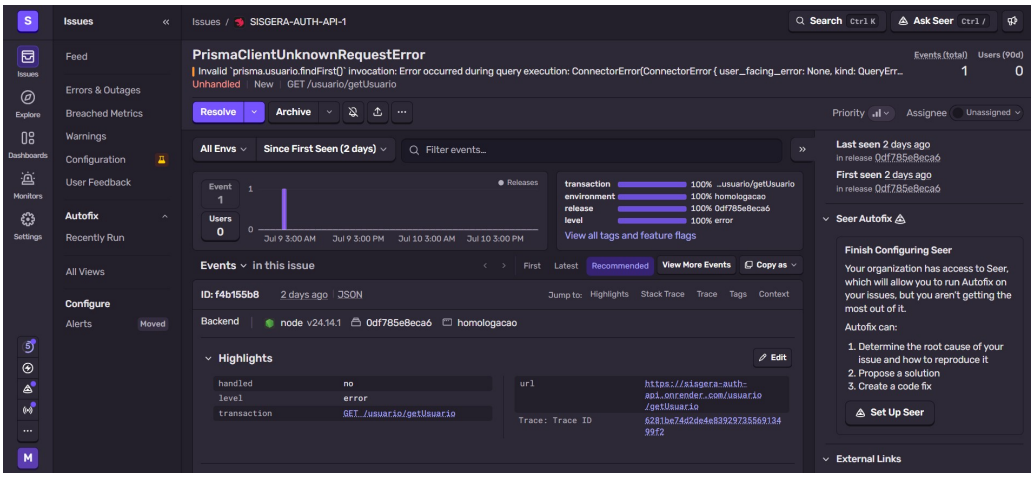
Cada microsserviço corresponde a um projeto próprio no Sentry, o que permite acompanhar a saúde de cada serviço de forma separada. Os eventos são ainda marcados pelo ambiente de origem, o que distingue as falhas de homologação das de produção em um futuro ambiente produtivo.

3.8.2 Integração com os serviços

A integração foi feita por meio do módulo oficial do Sentry para o NestJS. O cliente é inicializado antes do carregamento da aplicação, e um filtro global de exceções é registrado para capturar as falhas. Esse filtro reporta apenas as exceções não tratadas, correspondentes a erros internos do servidor, e não as exceções previstas do fluxo normal, como respostas de não autorizado ou de recurso não encontrado. Com isso, o painel concentra os erros que realmente exigem investigação, sem ruído das validações esperadas. Quando o *token* de configuração do serviço não está definido, como no ambiente de desenvolvimento local, o cliente permanece inativo e não envia eventos.

A Figura 16 apresenta um erro real capturado em homologação, com a mensagem, a rota que originou a falha e o rastreamento da pilha até a linha do código responsável.

Figura 16 – Erro capturado pelo Sentry em ambiente de homologação



Fonte: elaborado pelo autor.

3.9 Estratégia de testes

Com base na pirâmide de testes apresentada na Seção 2, a estratégia de testes do SisGera concentrou-se na base da pirâmide, os testes unitários, aplicados sobre a camada

de serviço de cada microsserviço.

3.9.1 Foco na camada de serviço

A camada de serviço concentra as regras de negócio e é, portanto, onde os testes trazem maior retorno. Para testá-la de forma isolada, as dependências externas, sobretudo os repositórios de acesso ao banco, são substituídas por objetos simulados (*mocks*). Assim, cada teste exercita apenas a lógica do serviço, sem depender de um banco de dados real ou de rede, o que os torna rápidos e determinísticos. Os dados de apoio usados nas verificações são centralizados em um módulo de *fixtures* compartilhado dentro de cada serviço.

Algumas camadas foram deixadas fora do escopo de testes por decisão consciente. Os controladores, por não conterem regras de negócio, e os repositórios, cujo comportamento é delegado ao Prisma, não são cobertos por testes unitários próprios. Testes de ponta a ponta também ficaram fora do escopo deste trabalho, dado o custo de manutenção discutido na Seção 2.

3.9.2 Cobertura e execução

Ao todo, foram implementados 259 testes unitários distribuídos entre os três serviços, com cobertura dos serviços de domínio acima de 90%, utilizando o *framework* Jest em conjunto com o módulo de testes do NestJS. A suíte é executada em dois momentos: localmente, antes de cada envio de código ao repositório, por meio de um gatilho automático de *pré-push*, e remotamente, na esteira de integração contínua descrita na Seção 3.10, que impede a implantação de código cujos testes não passem.

3.10 Integração e entrega contínuas

Um dos objetivos específicos deste trabalho, apresentados na Seção 1.2, foi adaptar o *pipeline* de integração e entrega contínuas desenvolvido por Campos (2025). A esteira original, embora funcional, foi construída para o sistema monolítico em PHP, com implantação por acesso remoto (SSH) a um ambiente de hospedagem compartilhada baseado em cPanel. Por essa razão, sua implementação específica não era reaproveitável, mas sua metodologia sim.

3.10.1 Metodologia reaproveitada

Da esteira de Campos (2025) preservou-se a estrutura em estágios de integração, entrega e reversão, o condicionamento da implantação ao sucesso das verificações automáticas e a segregação de ambientes. Essa metodologia foi reimplementada com o GitHub

Actions e o Render, no lugar do acesso remoto ao cPanel, adaptando-se à arquitetura de microsserviços em monorepo.

3.10.2 Integração contínua

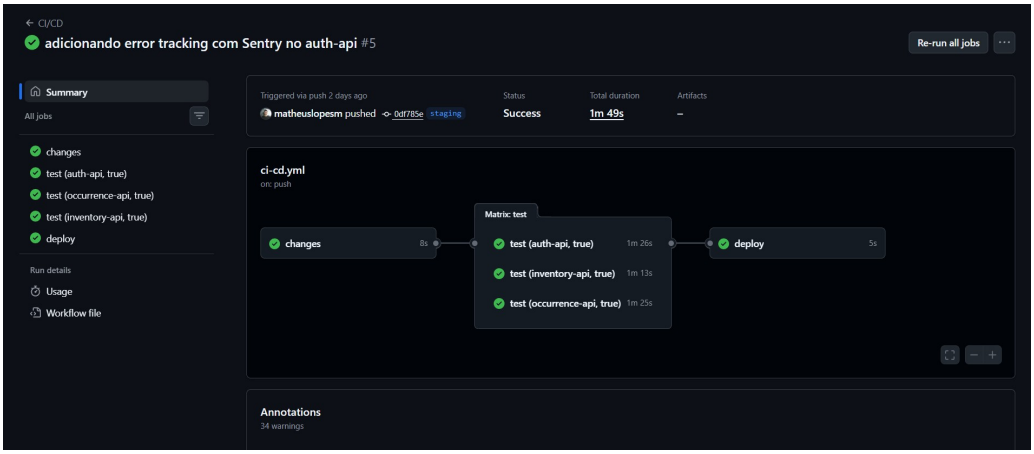
A esteira é disparada a cada envio de código e a cada solicitação de integração. Como os três serviços residem em um único repositório, uma etapa inicial detecta quais serviços foram de fato alterados e restringe as verificações apenas a eles, o que evita reconstruir e reimplantar serviços não afetados. Para cada serviço alterado, a esteira executa, em sequência, a verificação de estilo e padronização do código, a checagem de tipos, a aplicação das *migrations* sobre um banco PostgreSQL temporário, a verificação de divergência entre o esquema declarado e as *migrations*, a execução dos testes unitários e a construção da aplicação. A verificação de divergência garante que o esquema versionado e as *migrations* estejam sempre coerentes entre si.

3.10.3 Entrega contínua e reversão

A etapa de entrega só é executada em envios para o repositório, nunca em solicitações de integração, e apenas quando as verificações da etapa anterior passam. Confirmada essa condição, a implantação de cada serviço alterado é acionada por meio dos ganchos de publicação (*deploy hooks*) do Render, que reconstroem e reiniciam o serviço correspondente. Os envios para o ramo de homologação implantam no ambiente de homologação, e os ganchos para um futuro ambiente de produção já estão previstos na esteira, condicionados ao ramo principal.

A reversão de uma implantação é oferecida como um procedimento manual acionável sob demanda, que utiliza a interface de programação do Render para retornar um serviço à sua versão anterior. Isso cumpre o mesmo papel do estágio de reversão da esteira original, adaptado à plataforma gerenciada.

Figura 17 – Execução da esteira de integração e entrega contínuas no GitHub Actions



Fonte: elaborado pelo autor.

3.11 Infraestrutura e implantação

A implantação em nuvem foi feita na plataforma Render, escolhida por oferecer, de forma gerenciada, os três tipos de recurso de que o sistema necessita: serviços de aplicação, banco de dados PostgreSQL e Redis. A infraestrutura é descrita de forma declarativa em um arquivo de configuração versionado junto ao código, o que permite recriar todo o ambiente a partir do repositório.

3.11.1 Ambiente de homologação

Para o ambiente de homologação, os recursos foram consolidados de modo a caber nos limites do plano gratuito da plataforma. Em vez de um banco de dados por serviço em instâncias separadas, utilizou-se uma única instância PostgreSQL com esquemas distintos, um por serviço, e uma única instância Redis compartilhada. Essa consolidação é uma decisão de custo e não altera a separação lógica entre os serviços, que continuam com esquemas isolados e sem acesso cruzado aos dados. O plano gratuito impõe restrições, como a expiração periódica do banco e a suspensão de serviços ociosos, aceitáveis para um ambiente de homologação.

3.11.2 Configuração por ambiente

As configurações que variam entre ambientes, como as credenciais de banco, os segredos de autenticação e o *token* do serviço de monitoramento, são fornecidas por variáveis de ambiente, sem constar no código. Valores sensíveis são definidos diretamente na plataforma, enquanto valores não sensíveis podem ter um padrão declarado no arquivo de configuração. Essa separação entre código e configuração permite que um futuro ambiente de produção seja provisionado com o mesmo código e um conjunto distinto de variáveis.

A restrição de origens das requisições (CORS) também é configurável por ambiente, o que permite liberar apenas a origem do *front-end* em produção. Vale observar que essa restrição atua somente sobre clientes executados em navegador; aplicativos móveis nativos não são afetados por ela, de modo que a proteção efetiva dos recursos permanece a cargo da autenticação por *token*, válida para qualquer tipo de cliente.

3.12 Considerações finais

Este capítulo percorreu o desenvolvimento da nova arquitetura do SisGera, desde o diagnóstico do sistema legado até a infraestrutura de implantação em nuvem. Cada problema identificado na análise inicial foi tratado por uma decisão de projeto correspondente, e os mecanismos transversais de acesso, auditoria, monitoramento e entrega contínua foram descritos em detalhe. A Tabela 1 consolida as principais estratégias de implementação adotadas, com as tecnologias correspondentes, e serve de referência rápida para as decisões detalhadas ao longo do capítulo.

Tabela 1 – Resumo das estratégias de implementação e tecnologias adotadas

Aspecto	**Estratégia adotada**	**Tecnologias**
Arquitetura	Três microsserviços definidos por domínios de negócio, sem comunicação HTTP entre si	NestJS, TypeScript
Organização interna	Camadas de controlador, serviço e repositório, com injeção de dependências	NestJS
Persistência	Um banco de dados por serviço, com versionamento por *migrations*	PostgreSQL, Prisma
Modelagem	Tipos adequados, relacionamentos explícitos, enumerações para domínio estático, exclusão lógica e escopo multicorporação	Prisma, TypeScript
Autenticação	Tokens *stateless* com lista de revogação compartilhada	JWT, Passport, bcrypt, Redis
Autorização	RBAC declarativo por entidade e ação, com isolamento por corporação	NestJS (*guards* e decoradores)
Auditoria	Interceptação automática de mutações, com propagação do usuário por contexto de requisição	Prisma (*middleware*), Node.js (*AsyncLocalStorage*)
Validação e documentação	Validação declarativa nos modelos de entrada e documentação interativa das APIs	class-validator, Swagger (OpenAPI)
Testes	Testes unitários da camada de serviço com repositórios simulados	Jest
Qualidade de código	Análise estática, formatação padronizada e verificação antes de cada envio	ESLint, Prettier, Husky
Integração e entrega contínuas	Esteira com detecção de mudanças por serviço, verificações automáticas e implantação condicionada	GitHub Actions, Render (*deploy hooks*)
Observabilidade	Captura centralizada de exceções, com um projeto por serviço	Sentry
Infraestrutura	Plataforma gerenciada em nuvem, com configuração declarativa e ambiente local reproduzível	Render (PaaS), Docker Compose

Fonte: elaborado pelo autor.

Com a implementação apresentada, o próximo capítulo avalia os resultados da refatoração, traçando um paralelo entre o sistema legado e a nova arquitetura e descrevendo o *pipeline* de integração e entrega contínuas em operação.

4 Resultados

4.1 Considerações iniciais

Apresentado o desenvolvimento da nova arquitetura no Capítulo 3, este capítulo avalia os resultados da refatoração sob duas perspectivas. Primeiro, traça um paralelo entre o sistema legado e a nova arquitetura, retomando cada problema identificado na Seção 3.2.3 e a melhoria correspondente. Em seguida, descreve o funcionamento do *pipeline* de integração e entrega contínuas em operação, etapa por etapa, e o compara com a esteira original de Campos (2025). Por fim, apresenta como as funcionalidades do sistema foram validadas.

4.2 Paralelo entre o sistema legado e a nova arquitetura

A Tabela 2 resume o comparativo entre as duas versões do sistema, organizado pelos aspectos discutidos ao longo do Capítulo 3. Cada linha retoma uma limitação do sistema legado e a solução implementada na refatoração.

Tabela 2 – Comparativo entre o sistema legado e a nova arquitetura

Aspecto	**Sistema legado**	**Sistema refatorado**
Arquitetura	Monólito em PHP/Laravel, com *front-end* e *back-end* acoplados	Três microsserviços em TypeScript/NestJS, implantados de forma independente
Organização do código	Regras de negócio concentradas nos *controllers*	Camadas de controlador, serviço e repositório com responsabilidades definidas
Testes	Sem suíte de testes significativa	259 testes unitários, com cobertura superior a 90% na camada de serviço
Controle de acesso	Comparações fixas por tipo de usuário, dispersas no código	RBAC com permissões por entidade e ação, aplicado por *guard* global e escopo por corporação
Auditoria	Inexistente	Trilha automática de alterações em todos os serviços
Modelagem de dados	Campos numéricos como texto, vetores serializados e tabelas de domínio estático	Tipos adequados, relacionamentos explícitos e enumerações no código
Dados regionais	Lista de hospitais fixa no código	Cadastro de hospitais gerenciado por cada corporação
Base tecnológica	Versões de linguagem e *framework* sem suporte oficial	Plataforma moderna, com suporte ativo
Implantação	Hospedagem compartilhada (cPanel), implantação via SSH	PaaS em nuvem, com implantação automatizada e monitoramento de erros
Escalabilidade	Sistema escala apenas como um todo	Cada serviço pode ser escalado de forma independente

Fonte: o autor.

4.2.1 Paridade funcional entre os sistemas

Além do contraste arquitetural apresentado na Tabela 2, é importante demonstrar que a refatoração preservou o escopo funcional do sistema legado. A Tabela 3 relaciona as principais funcionalidades, indicando quais já existiam, quais foram mantidas ou aprimoradas, e quais são inteiramente novas.

Tabela 3 – Paridade funcional entre o sistema legado e o sistema refatorado

Funcionalidade	**Sistema legado**	**Sistema refatorado**
Registro de ocorrências (resgate, incêndio, simplificado)	Existia	Mantida
Registro de vítimas e avaliação clínica	Existia, com problemas de tipagem	Aprimorada
Encaminhamento hospitalar	Existia	Mantida
Veículos e equipe de plantão da ocorrência	Existia	Mantida
Fotos da ocorrência	Existia	Mantida
Cadastro de hospitais	Existia, fixo no código	Aprimorada (gerenciado por corporação)
Gestão de patrimônio	Existia	Mantida
Controle de produtos e estoque	Existia	Mantida
Registro de doações	Existia	Mantida
Administração de usuários e corporações	Existia	Mantida
Controle de acesso por papéis e permissões	Existia, de forma frágil	Aprimorada (RBAC declarativo)
Auditoria automática das operações	Não existia	Nova
Monitoramento de erros	Não existia	Nova
Testes automatizados	Não existia	Nova
Documentação interativa da API	Não existia	Nova
Pipeline de integração e entrega contínuas	Existia (CAMPOS, 2025)	Adaptada
Armazenamento de arquivos e imagens	Não existia	Não implementada (trabalho futuro)

Fonte: elaborado pelo autor.

A testabilidade é o contraste mais mensurável, já que o sistema legado não dispunha de testes automatizados relevantes, enquanto a nova arquitetura conta com 259 testes unitários distribuídos entre os três serviços (73 na auth-api, 101 na occurrence-api e 85 na inventory-api), executados a cada envio de código. Isso significa que qualquer alteração passa por uma verificação automática de regressões antes de chegar ao ambiente de homologação, garantia que não existia na versão anterior.

O controle de acesso também mudou de natureza, pois no sistema legado a autorização dependia de comparações de tipo de usuário espalhadas pelo código, difíceis de auditar e de manter. Na nova arquitetura, as permissões são declaradas por *endpoint* e verificadas em um único ponto, e o isolamento por corporação é aplicado de forma sistemática na camada de serviço, como descrito na Seção 3.6. A auditoria automática, por sua vez, é

uma capacidade nova: o sistema legado não registrava quem alterava os dados, e a versão refatorada registra todas as mutações com autor, dados anteriores e novos, sem depender de instrumentação manual.

Na modelagem de dados, os problemas concretos apontados na análise do legado deixaram de existir por construção: sinais vitais que aceitavam valores como `"000x00"` passaram a campos inteiros validáveis, os vetores serializados deram lugar a tabelas de relacionamento e as tabelas de domínio estático viraram enumerações tipadas, conforme a Seção 3.5. Por fim, aspectos que não possuíam equivalente no sistema anterior, como o monitoramento centralizado de erros (Seção 3.8), passaram a fazer parte da operação do sistema.

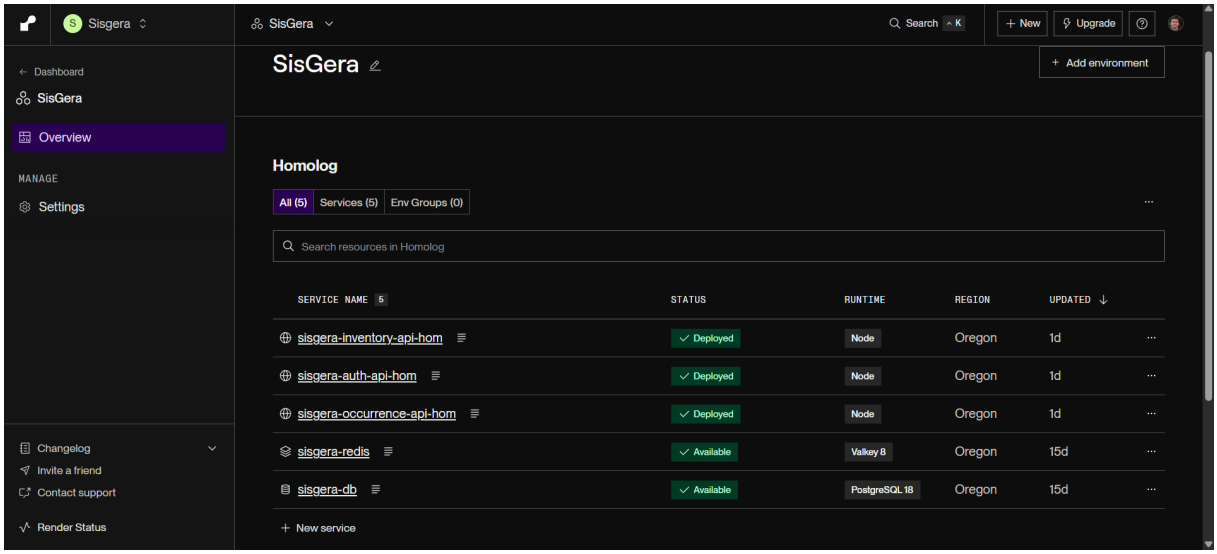
4.2.2 Arquitetura em operação

Além do contraste estrutural apresentado na Tabela 2, o funcionamento da nova arquitetura foi verificado em condições reais, com os três serviços implantados no ambiente de homologação. Esta seção descreve evidências concretas desse funcionamento, organizadas pelos mesmos mecanismos discutidos no Capítulo 3.

4.2.2.0.1 Implantação e persistência independentes.

Os três serviços estão publicados em endereços próprios (`sisgera-auth-api`, `sisgera-occurrence-api` e `sisgera-inventory-api`), cada um com seu próprio processo e ciclo de implantação, como mostra a Figura 18. A separação de dados também se confirma no ambiente de homologação: cada serviço acessa exclusivamente o seu schema no banco PostgreSQL, reproduzindo em nuvem o isolamento definido nos três esquemas do Prisma discutidos na Seção 3.5.

Figura 18 – Serviços implantados de forma independente no Render

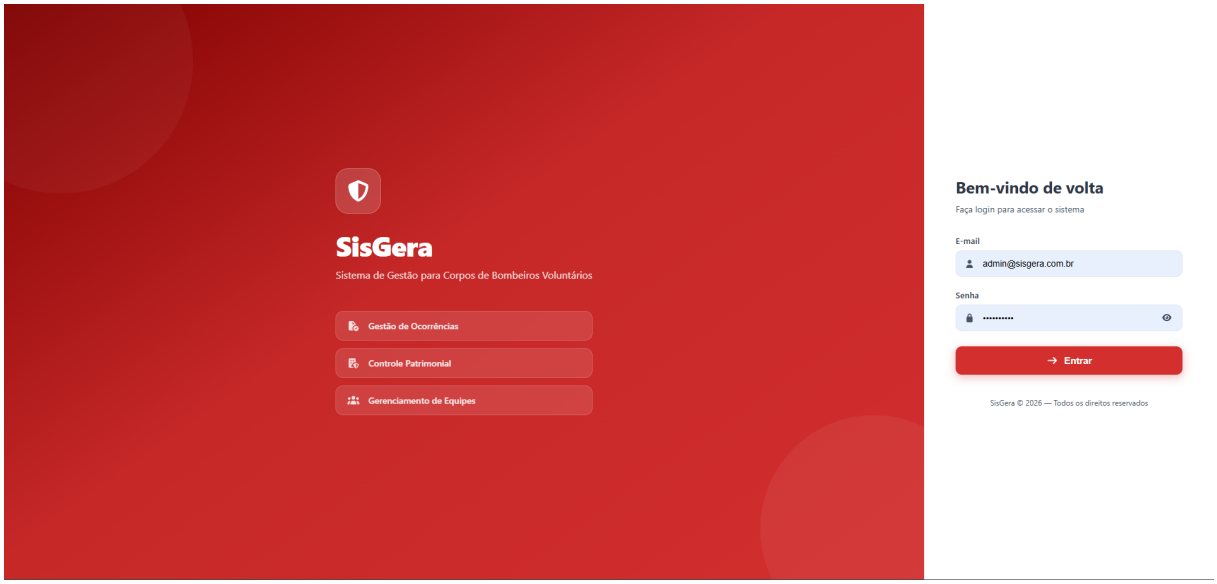


Fonte: elaborado pelo autor.

4.2.2.0.2 Sistema em uso via *front-end*.

Além dos testes realizados diretamente sobre as APIs, a arquitetura foi exercitada por meio de uma aplicação *front-end* publicada, desenvolvida em trabalho paralelo e consumindo os três serviços implantados no ambiente de homologação. A Figura 19 apresenta a tela de autenticação dessa aplicação, ponto de entrada do fluxo descrito na Seção 3.4: ao submeter as credenciais, o *front-end* envia uma requisição à auth-api, recebe o *token* JWT emitido e o armazena no cliente, anexando-o ao cabeçalho de autorização de toda requisição subsequente às demais APIs. É esse mesmo mecanismo, e não uma simulação isolada, que sustenta as evidências de autorização e de isolamento por corporação apresentadas a seguir, obtidas a partir de sessões autenticadas dessa forma.

Figura 19 – Tela de autenticação da aplicação *front-end*

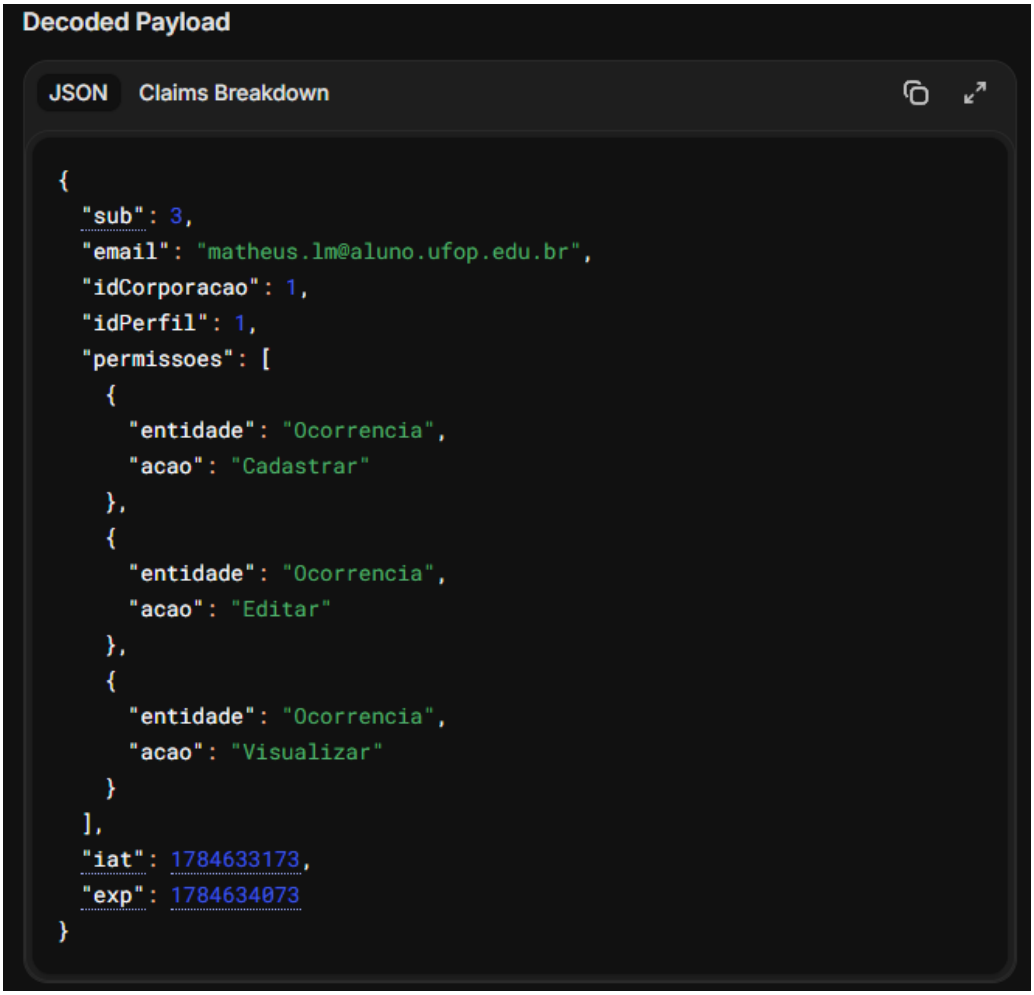


Fonte: elaborado pelo autor.

4.2.2.0.3 Autorização distribuída via token.

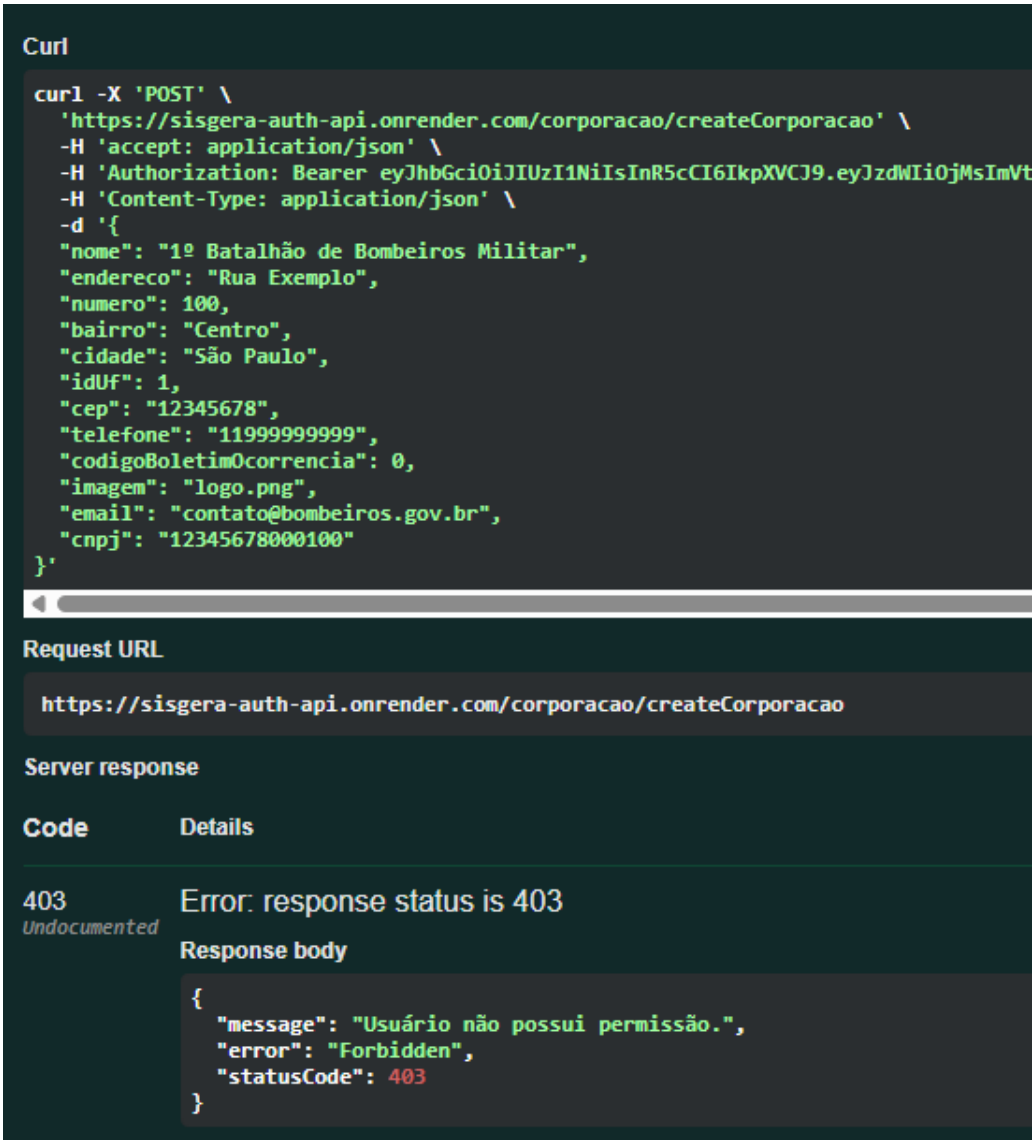
O mecanismo descrito na Seção 3.6 foi exercitado por meio de uma autenticação real na auth-api. O token JWT retornado carrega, entre outras *claims*, o identificador do usuário, sua corporação, seu perfil e a lista de permissões concedidas, como mostra a Figura 20, obtida a partir de um token emitido em homologação e decodificado para fins de verificação. É esse conjunto de permissões que o occurrence-api e o inventory-api inspecionam a cada requisição, sem consultar a auth-api. A Figura 21 mostra a resposta de acesso negado obtida ao solicitar uma ação sem a permissão correspondente, confirmando o comportamento descrito na Seção 3.6.

Figura 20 – Token JWT decodificado, com as *claims* de autorização



Fonte: elaborado pelo autor.

Figura 21 – Resposta de acesso negado para usuário sem a permissão exigida

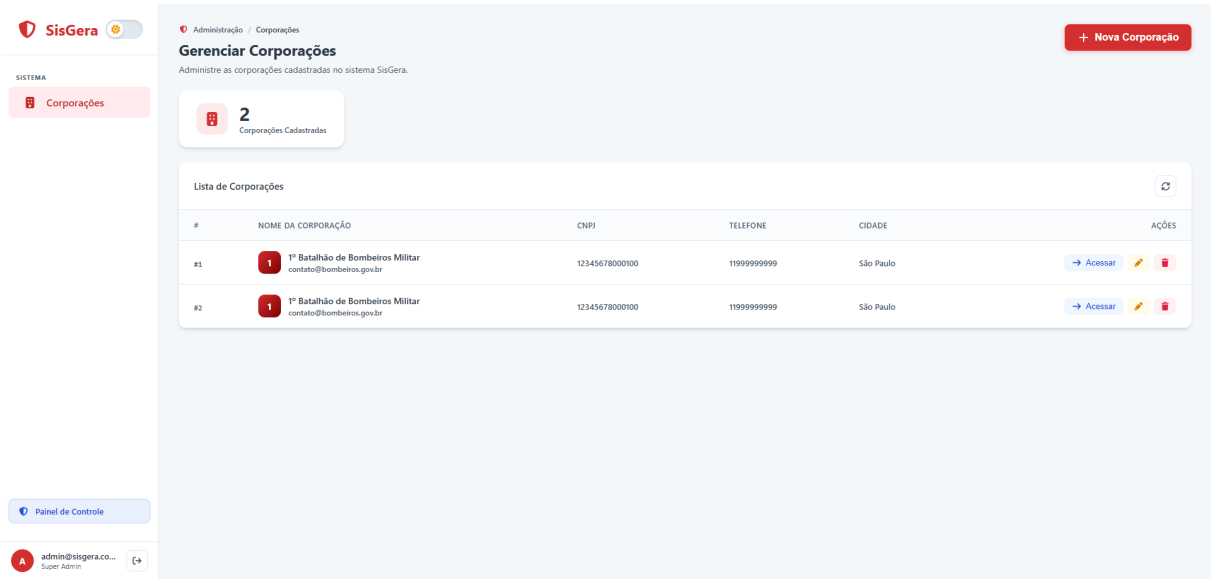


Fonte: elaborado pelo autor.

4.2.2.0.4 Isolamento por corporação.

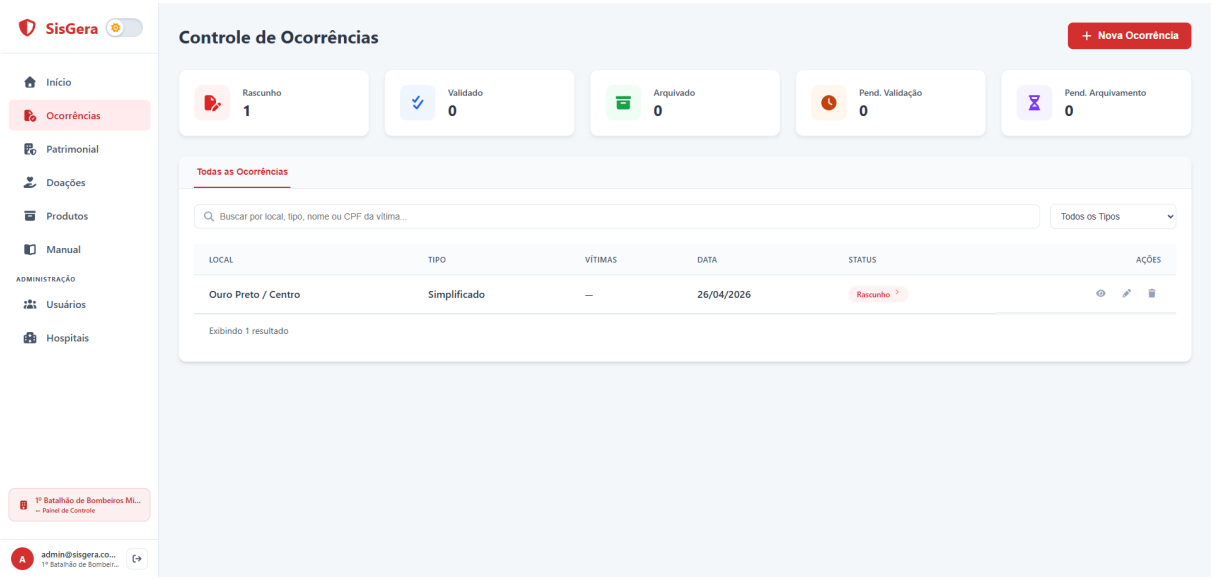
O escopo multicorporação descrito na Seção 3.6 também foi verificado com dados reais, a partir de duas corporações distintas cadastradas no ambiente de homologação. A Figura 22 mostra o cadastro de corporações consultado por um usuário administrador, com acesso irrestrito e visão global do sistema. Ao consultar, em seguida, a listagem de ocorrências de cada corporação separadamente, os conjuntos de dados retornados são distintos entre si, como mostram as Figuras 23 e 24, confirmando que os dados de cada corporação permanecem segregados mesmo estando armazenados no mesmo banco de dados do serviço, conforme o campo `idCorporacao` discutido na Seção 3.5.

Figura 22 – Cadastro de corporações, com duas corporações registradas em homologação



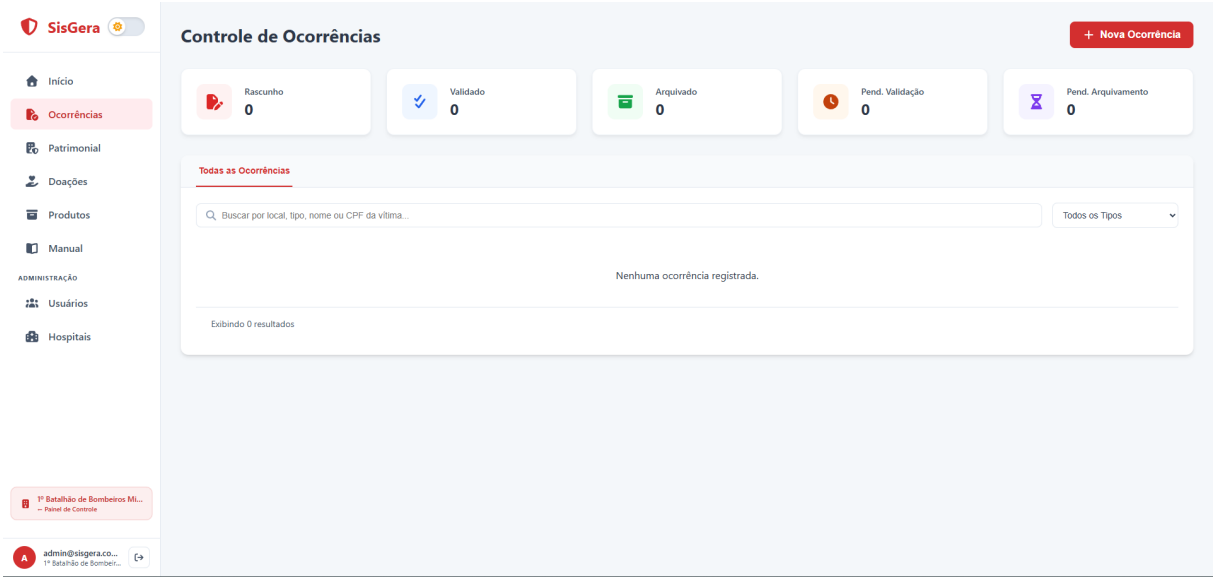
Fonte: elaborado pelo autor.

Figura 23 – Ocorrências da primeira corporação



Fonte: elaborado pelo autor.

Figura 24 – Ocorrências da segunda corporação, com um conjunto de dados distinto



Fonte: elaborado pelo autor.

4.2.2.0.5 Validação de regras de negócio.

As validações declaradas nos modelos de entrada, descritas na Seção 3.4, também se comportam como esperado no ambiente de homologação. Por exemplo, uma requisição de criação de ocorrência com hora de chegada ao local posterior à hora de chegada ao hospital é rejeitada pela occurrence-api antes de qualquer escrita no banco:

```
{
  "message": "A hora de chegada ao local não pode ser posterior
              à hora de chegada ao hospital.",
  "error": "Bad Request",
  "statusCode": 400
}
```

4.2.2.0.6 Auditoria automática.

Operações de escrita realizadas durante os testes foram registradas de forma automática na tabela `Audit` de cada serviço, sem qualquer chamada explícita nos controladores envolvidos, confirmando o funcionamento do mecanismo de interceptação descrito na Seção 3.7. O objeto JSON abaixo apresenta um registro real, com o usuário responsável, a entidade afetada, a ação realizada e os dados antes e depois da alteração.

```
    {
      "idAudit": 7,
```

```
"idUsuario": 1,
"idCorporacao": null,
"entidade": "Vitima",
"acao": "Excluir",
"dadosAntigos": {
  "cpf": "12345678901",
  "nome": "João da Silva",
  "nomeMae": "Maria da Silva",
  "idGenero": 1,
  "idVitima": 1,
  "telefone": "31999999999",
  "nomeSocial": "Joana",
  "dataCadastro": "2026-07-05T17:23:09.947Z",
  "dataExclusao": null,
  "idOcorrencia": 1,
  "dataAlteracao": "2026-07-05T17:23:09.947Z",
  "idTipoDestino": 1,
  "dataNascimento": "1990-05-15T00:00:00.000Z",
  "idVeiculoOcorrencia": 1,
  "caminhoArquivoRecusa": null
},
"dadosNovos": {
  "cpf": "12345678901",
  "nome": "João da Silva",
  "nomeMae": "Maria da Silva",
  "idGenero": 1,
  "idVitima": 1,
  "telefone": "31999999999",
  "nomeSocial": "Joana",
  "dataCadastro": "2026-07-05T17:23:09.947Z",
  "dataExclusao": "2026-07-05T17:23:23.425Z",
  "idOcorrencia": 1,
  "dataAlteracao": "2026-07-05T17:23:23.427Z",
  "idTipoDestino": 1,
  "dataNascimento": "1990-05-15T00:00:00.000Z",
  "idVeiculoOcorrencia": 1,
  "caminhoArquivoRecusa": null
},
"dataCadastro": "2026-07-05T17:23:23.433Z"
```

```
},
```

4.3 O pipeline em operação

O *pipeline* descrito na Seção 3.10 preservou a metodologia da esteira de Campos (2025), estruturada nos estágios de integração contínua, entrega contínua e reversão, e substituiu sua implementação, antes baseada em acesso SSH a uma hospedagem cPanel, pela combinação de GitHub Actions e Render. Esta seção percorre uma execução real da esteira, etapa por etapa, no mesmo formato adotado por aquele trabalho.

4.3.1 Estágio de integração contínua

O estágio de verificação é executado a cada envio de código ao repositório e a cada solicitação de integração, e compreende as seguintes etapas:

1. **Detecção de mudanças:** como os três serviços residem em um único repositório, a primeira etapa identifica quais deles foram de fato alterados no envio. As etapas seguintes são executadas apenas para os serviços afetados, o que evita reconstruir e reimplantar o que não mudou;
2. **Verificação de estilo:** o código do serviço é analisado pelo ESLint, que rejeita a execução caso existam violações das regras de qualidade e de tipagem segura definidas no projeto;
3. **Checagem de tipos:** o compilador TypeScript verifica a consistência de tipos de todo o serviço, sem gerar artefatos;
4. **Verificação do banco de dados:** um banco PostgreSQL temporário é criado no próprio executor, as *migrations* do serviço são aplicadas nele do zero e, em seguida, verifica-se que não há divergência entre o esquema declarado e o esquema resultante das *migrations*. Essa etapa garante que o histórico de *migrations* está íntegro e reproduzível;
5. **Testes unitários:** a suíte de testes do serviço é executada por completo, e qualquer falha interrompe a esteira;
6. **Construção:** por fim, a aplicação é compilada para produção, confirmando que o serviço é implantável.

A Figura 17 apresenta uma execução real desse estágio no GitHub Actions, com as etapas concluídas com sucesso para os serviços alterados.

4.3.2 Estágio de entrega contínua

O estágio de entrega é executado somente em envios diretos ao repositório, nunca em solicitações de integração, e apenas quando todas as verificações do estágio anterior passam. Confirmada essa condição, a implantação de cada serviço alterado é acionada por meio dos ganchos de publicação (*deploy hooks*) do Render, que reconstroem e reiniciam o serviço correspondente no ambiente de homologação. Serviços não alterados não são reimplantados.

Aqui está a principal diferença de implementação em relação à esteira de Campos (2025): naquele trabalho, a entrega era feita por acesso remoto (SSH) ao servidor compartilhado, com a publicação atômica realizada pela troca de um link simbólico entre versões. Na nova esteira, essa mecânica é delegada à plataforma gerenciada, que executa a construção e a substituição da versão em execução, preservando o mesmo efeito prático com menor complexidade operacional.

4.3.3 Estágio de reversão

Assim como a esteira original previa um estágio dedicado à reversão, a nova esteira oferece um procedimento de *rollback* acionável manualmente. Por meio de um fluxo de execução sob demanda no GitHub Actions, o operador seleciona o serviço e o ambiente desejados, e a esteira utiliza a interface de programação do Render para retornar o serviço à sua versão implantada anterior. O que na implementação original era resolvido pela troca do link simbólico para a versão precedente é aqui resolvido pela plataforma, com o mesmo objetivo: restaurar rapidamente um estado funcional em caso de problema na implantação.

4.3.4 Síntese do paralelo entre as esteiras

Em resumo, a adaptação preservou de Campos (2025) a estrutura em três estágios, o condicionamento da entrega ao sucesso das verificações e a preocupação com a reversibilidade das implantações. Foram substituídos o mecanismo de publicação (de SSH e links simbólicos para ganchos de implantação em PaaS) e o alvo da implantação (de uma hospedagem compartilhada para serviços gerenciados em nuvem), além da adição de etapas que não existiam na esteira original, como a detecção de mudanças por serviço, a verificação de integridade das *migrations* e o monitoramento de erros pós-implantação com o Sentry.

4.4 Validação das funcionalidades

A validação do sistema ocorreu em dois níveis. No nível de código, os 259 testes unitários são executados em dois momentos: localmente, antes de cada envio ao repositório, e na esteira de integração contínua, que bloqueia a implantação de qualquer alteração cujos testes falhem. No nível de sistema, os três serviços foram implantados no ambiente de homologação e exercitados por meio da documentação interativa (Swagger) e de uma aplicação *front-end* desenvolvida em trabalho paralelo, que consome as três APIs.

Os fluxos principais foram verificados de ponta a ponta nesse ambiente: autenticação e revogação de sessão, cadastro e consulta de ocorrências com vítimas e avaliações clínicas, gestão de inventário, restrição de acesso por permissão e por corporação, e registro automático de auditoria para as operações de escrita. O monitoramento de erros também foi validado em condições reais: as exceções provocadas em homologação foram capturadas e apresentadas no painel do Sentry, com o rastreamento completo até a linha de código responsável, como mostrado na Figura 16.

Por fim, cabe registrar o que não foi medido. A avaliação empírica de desempenho e de escalabilidade, como testes de carga sobre os serviços, não fez parte do escopo desta validação, pois os planos gratuitos utilizados no ambiente de homologação impõem limites de recursos que distorceriam os resultados. Essa avaliação está registrada como trabalho futuro, a ser conduzida sobre um ambiente de produção adequadamente dimensionado.

4.5 Considerações finais

Este capítulo mostrou que cada limitação do sistema legado encontrou uma solução concreta na nova arquitetura, que a esteira de integração e entrega contínuas preservou a metodologia do trabalho anterior com uma implementação adequada ao novo contexto, e que as funcionalidades foram validadas por testes automatizados e pela operação em ambiente de homologação. O capítulo seguinte encerra o trabalho, retomando seus objetivos e apontando as direções para trabalhos futuros.

5 Conclusão

Este trabalho conduziu a refatoração do SisGera, um sistema de gestão para corporações de bombeiros voluntários, de uma arquitetura monolítica para uma arquitetura de microsserviços. Como discutido ao longo do texto, as limitações que motivaram a reformulação não estavam no escopo funcional do sistema, já consolidado por trabalhos anteriores, mas na forma como ele havia sido arquitetado e implementado, o que comprometia atributos de qualidade como manutenibilidade, testabilidade e escalabilidade.

A partir do diagnóstico do sistema legado, apresentado no Capítulo 3, o *back-end* foi reconstruído e dividido em três serviços autônomos, correspondentes aos contextos de identidade e acesso, de ocorrências e de inventário. Cada serviço adota uma organização em camadas bem definidas, possui seu próprio banco de dados e é desenvolvido, testado e implantado de forma independente. Sobre essa base, foram implementados um modelo estruturado de controle de acesso por papéis e permissões, um mecanismo de auditoria automática das operações e uma esteira de integração e entrega contínuas, além da publicação dos serviços em nuvem com monitoramento de erros.

Cada uma das limitações identificadas no sistema legado deu origem a uma solução concreta na nova arquitetura. O forte acoplamento e a ausência de camadas foram tratados pela separação em serviços e pela estrutura de controlador, serviço e repositório, que também viabilizou a criação de uma suíte de testes automatizados, ausente na versão anterior. O controle de acesso frágil foi substituído por um modelo de papéis e permissões aplicado de forma sistemática, e os problemas de modelagem foram corrigidos com a adoção de tipos adequados, relacionamentos explícitos e enumerações. A obsolescência tecnológica, por fim, foi resolvida pela migração para uma plataforma moderna e com suporte ativo.

Quanto aos objetivos definidos no Capítulo 1, a nova arquitetura de microsserviços foi implementada, o *pipeline* de integração e entrega contínuas de Campos (2025) foi adaptado ao novo contexto, e o comportamento dos serviços foi validado por meio de testes automatizados e da operação real em ambiente de homologação, conforme apresentado no Capítulo 4. A avaliação de desempenho e qualidade entre a arquitetura original e a nova foi conduzida sob o aspecto qualitativo, evidenciado pelo comparativo da Seção 4.2 e pelas evidências de funcionamento apresentadas na Seção 4.2.2. Uma avaliação quantitativa, em especial dos ganhos de escalabilidade, depende de testes de carga e de um ambiente de produção definitivo, apontados como trabalhos futuros. Mais do que entregar um sistema reescrito, este trabalho procurou documentar as decisões de projeto que orientaram a refatoração, de modo a servir de referência e de base sustentável para que futuros Trabalhos

de Conclusão de Curso deem continuidade à evolução do SisGera.

5.1 Trabalhos futuros

A refatoração conduzida neste trabalho estabeleceu uma base sobre a qual novas funcionalidades e melhorias podem ser construídas. A seguir são apresentadas as principais direções identificadas para a continuidade do projeto.

- **Evolução da autenticação:** o fluxo de autenticação pode ser ampliado com a recuperação de senha, que permite ao usuário redefinir o acesso quando o esquece, com o *login* por provedores externos, como contas Google, e com a autenticação de dois fatores (2FA), que adiciona uma camada extra de segurança ao acesso.
- **Armazenamento de arquivos e imagens:** como discutido na Seção 3.5, o modelo de dados já reserva campos para o caminho de arquivos, como a imagem da corporação e o termo de recusa da vítima. Resta integrar um serviço de armazenamento de objetos (*object storage*) compatível com o padrão S3, responsável por guardar os arquivos e devolver referências temporárias de acesso, sem que o esquema do banco precise ser alterado.
- **Processamento assíncrono e concorrência:** a comunicação entre os serviços ocorre hoje apenas de forma indireta, por meio do *token*. A introdução de uma fila de mensagens permitiria a comunicação assíncrona entre serviços e o tratamento de operações concorrentes, endereçando o desafio de consistência de dados inerente à arquitetura de microsserviços, discutido no Capítulo 2.
- **Operação offline:** por atender a corporações de bombeiros, que atuam em campo e nem sempre dispõem de conexão estável, o sistema poderia oferecer a capacidade de registrar ocorrências sem conexão. A abordagem usual para esse cenário é uma fila de sincronização local, em que o cliente grava a ocorrência em um banco local no próprio dispositivo, com um identificador temporário, e um processo em segundo plano envia os registros pendentes para a occurrence-api assim que a conectividade é restabelecida, substituindo o identificador temporário pelo gerado no servidor. Como o cadastro inicial de uma ocorrência normalmente não envolve edição concorrente por múltiplos usuários, o risco de conflito nessa sincronização é baixo, o que tornaria viável uma estratégia de fila simples, sem a necessidade de resolução de conflitos mais elaborada.
- **Testes de integração e ponta a ponta:** a estratégia de testes concentrou-se nos testes unitários da camada de serviço, conforme a Seção 3.9. Um trabalho futuro é a adição de testes de integração e de ponta a ponta, que validem o comportamento entre serviços e o fluxo completo de uma requisição, do cliente ao banco de dados.

- **Avaliação de escalabilidade:** um dos benefícios atribuídos à arquitetura de microsserviços é a possibilidade de escalar cada serviço de forma independente. Testes de carga sobre os serviços de maior demanda permitiriam avaliar empiricamente esse ganho e dimensionar os recursos necessários.

- **Proteção contra força bruta no *login*:** o *endpoint* de autenticação não impõe hoje um limite de tentativas por usuário ou por origem, o que o deixa suscetível a ataques de adivinhação de senha. A introdução de um mecanismo de *rate limiting*, associado a um bloqueio temporário após tentativas malsucedidas sucessivas, reforçaria a segurança do acesso, especialmente relevante em um sistema que armazena dados sensíveis das vítimas atendidas.

- **Ambiente de produção e custos:** a implantação atual utiliza os planos gratuitos das plataformas, adequados à homologação. Um trabalho futuro é o levantamento dos custos de hospedagem e o provisionamento de um ambiente de produção definitivo, com recursos dimensionados para o uso real das corporações. Esse provisionamento inclui a conclusão da etapa de entrega para produção na esteira de integração e entrega contínuas, hoje preparada apenas para o ambiente de homologação (Seção 3.10).

Referências

ARANTES, V. M. *Um sistema de informação para apoio ao registro de ocorrências atendidas por grupos de bombeiros voluntários*. Dissertação (Mestrado) — Universidade Federal de Ouro Preto, João Monlevade, 2018. Monografia (graduação em Sistemas de Informação). Citado 4 vezes nas páginas 15, 27, 31 e 76.

Auth0. *JSON Web Tokens — jwt.io*. 2025. <https://jwt.io>. Acesso em: 22 jun. 2026. Citado na página 34.

CAMPOS, R. N. S. *Implementação de um pipeline CI/CD para o projeto SisGera*. Dissertação (Mestrado) — Universidade Federal de Ouro Preto, João Monlevade, 2025. Monografia (graduação em Sistemas de Informação). Citado 9 vezes nas páginas 16, 30, 31, 50, 55, 57, 66, 67 e 69.

class-validator. *class-validator: Decorator-based property validation*. 2025. <https://github.com/typestack/class-validator>. Acesso em: 22 jun. 2026. Citado na página 35.

COHN, M. *Desenvolvimento de Software com Scrum: Aplicando Métodos Ágeis com Sucesso*. [S.l.]: Bookman, 2011. Citado na página 24.

Docker, Inc. *Docker: Accelerated container application development*. 2025. <https://www.docker.com>. Acesso em: 22 jun. 2026. Citado na página 35.

Functional Software, Inc. *Sentry: Application Performance Monitoring and Error Tracking*. 2025. <https://sentry.io>. Acesso em: 9 jul. 2026. Citado 2 vezes nas páginas 36 e 48.

GitHub. *GitHub Actions*. 2025. <https://github.com/features/actions>. Acesso em: 9 jul. 2026. Citado na página 36.

Husky. *Husky: Git hooks made easy*. 2025. <https://typicode.github.io/husky>. Acesso em: 22 jun. 2026. Citado na página 35.

Meta Open Source. *Jest: Delightful JavaScript testing*. 2025. <https://jestjs.io>. Acesso em: 22 jun. 2026. Citado na página 35.

Microsoft. *TypeScript: JavaScript with syntax for types*. 2025. <https://www.typescriptlang.org>. Acesso em: 22 jun. 2026. Citado na página 34.

NestJS. *NestJS: A progressive Node.js framework*. 2025. <https://nestjs.com>. Acesso em: 22 jun. 2026. Citado na página 34.

NEWMAN, S. *Migrando sistemas monolíticos para microsserviços: Padrões evolutivos para transformar seu sistema monolítico*. [S.l.]: Novatec Editora, 2020. Citado 2 vezes nas páginas 18 e 19.

npm. *bcrypt.js: Optimized bcrypt in JavaScript*. 2025. <https://www.npmjs.com/package/bcryptjs>. Acesso em: 22 jun. 2026. Citado na página 34.

OLIVEIRA, S. S. M. *Sistema de informação para controle de materiais e doações aos bombeiros voluntários*. Dissertação (Mestrado) — Universidade Federal de Ouro Preto, João Monlevade, 2018. Monografia (graduação em Sistemas de Informação). Citado na página 15.

OpenJS Foundation. *ESLint: Find and fix problems in your JavaScript code*. 2025. <https://eslint.org>. Acesso em: 22 jun. 2026. Citado na página 35.

Passport. *Passport: Simple, unobtrusive authentication for Node.js*. 2025. <https://www.passportjs.org>. Acesso em: 22 jun. 2026. Citado na página 34.

Prettier. *Prettier: An opinionated code formatter*. 2025. <https://prettier.io>. Acesso em: 22 jun. 2026. Citado na página 35.

Prisma. *Prisma: Next-generation Node.js and TypeScript ORM*. 2025. <https://www.prisma.io>. Acesso em: 22 jun. 2026. Citado na página 34.

Redis. *Redis: The open source, in-memory data store*. 2025. <https://redis.io>. Acesso em: 22 jun. 2026. Citado na página 34.

Render. *Render: Cloud Application Platform*. 2025. <https://render.com>. Acesso em: 9 jul. 2026. Citado na página 36.

SmartBear Software. *Swagger: API documentation and design tools*. 2025. <https://swagger.io>. Acesso em: 22 jun. 2026. Citado na página 35.

Software Freedom Conservancy. *Git*. 2025. <https://git-scm.com>. Acesso em: 9 jul. 2026. Citado na página 36.

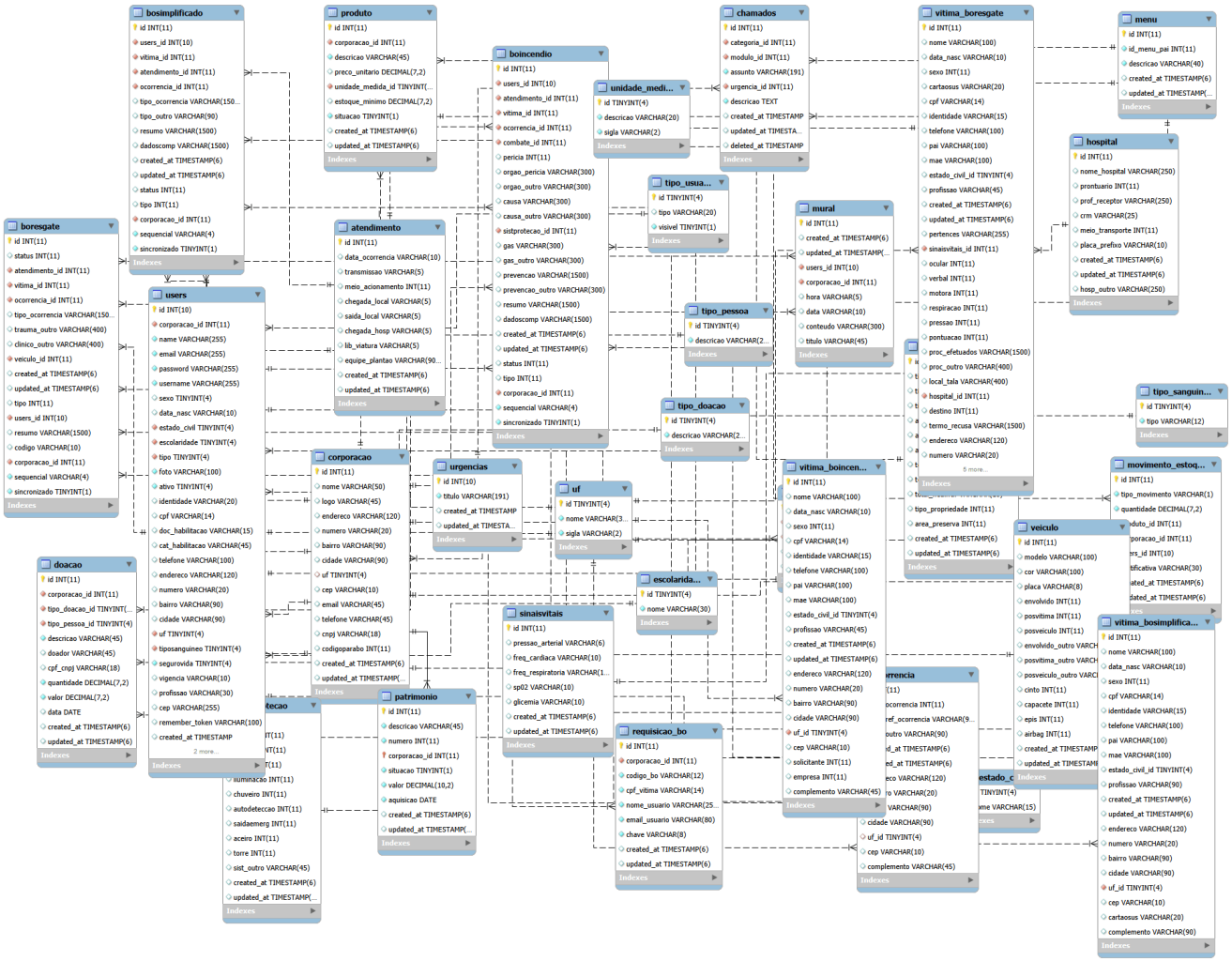
The PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source relational database*. 2025. <https://www.postgresql.org>. Acesso em: 22 jun. 2026. Citado na página 34.

Apêndices

APÊNDICE A – Diagrama entidade-relacionamento completo do sistema legado

Este apêndice apresenta o diagrama entidade-relacionamento completo do banco de dados do sistema legado, cujo recorte foi discutido na Seção 3.2.3. A página é apresentada em orientação paisagem para permitir a visualização da totalidade das tabelas e dos relacionamentos.

Figura 25 – Diagrama entidade-relacionamento completo do banco de dados do sistema legado



Fonte: elaborado pelo autor, com base na estrutura de banco de dados de Arantes (2018).