Pontifícia Universidade Católica
do Rio de Janeiro

Enrico Vergolino Gnani

Protegendo o KAPIO: Um Knowledge API Orchestrator para múltiplos sistemas heterogêneos com diferentes donos

Projeto Final de Graduação

Relatório de Projeto Final, apresentado ao programa Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do Bacharel em Engenharia de Computação.

Orientador : Prof. Vitor Pinheiro de Almeida
Co-orientador: Prof. Anderson Oliveira da Silva

Rio de Janeiro
Dezembro de 2025

Pontifícia Universidade Católica
do Rio de Janeiro

Enrico Vergolino Gnani

Protegendo o KAPIO: Um Knowledge API Orchestrator para múltiplos sistemas heterogêneos com diferentes donos

Relatório de Projeto Final, apresentado ao programa Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do Bacharel em Engenharia de Computação.

Prof. Vitor Pinheiro de Almeida
Orientador
Departamento de Informática – PUC-Rio

Prof. Anderson Oliveira da Silva
Co-orientador
Departamento de Informática – PUC-Rio

Rio de Janeiro, 21 de Dezembro de 2025

Enrico Vergolino Gnani

Graduando em Engenharia de Computação na PUC-Rio.

Ficha Catalográfica

Vergolino Gnani, Enrico

Protegendo o KAPIO: Um Knowledge API Orchestrator para múltiplos sistemas heterogêneos com diferentes donos / Enrico Vergolino Gnani; orientador: Vitor Pinheiro de Almeida; co-orientador: Anderson Oliveira da Silva. – 2025.

45 f: il. color. ; 30 cm

Projeto Final (Graduação) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2025.

Inclui bibliografia

1. Informática – Relatórios. 2. Cibersegurança. 3. Sistemas de Sistemas. 4. Gerenciamento de APIs. 5. GraphQL. I. Pinheiro de Almeida, Vitor. II. Oliveira da Silva, Anderson. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Assegurando KAPIO: A Knowledge API Orchestrator para múltiplos sistemas heterogêneos com diferentes donos.

CDD: 004

Agradecimentos

Aos meus pais, pela educação, atenção e carinho de todas as horas.

Ao meu orientador Professor Vitor Pinheiro de Almeida pelo estímulo, parceria e paciência para a realização deste trabalho.

Ao meu co-orientador Professor Anderson Oliveira da Silva, pela instrução e ótima introdução ao mundo da Segurança da Informação.

A todos os professores e funcionários do Departamento pelos ensinamentos e pela ajuda.

Resumo

Vergolino Gnani, Enrico; Pinheiro de Almeida, Vitor; Oliveira da Silva, Anderson. **Protegendo o KAPIO: Um Knowledge API Orchestrator para múltiplos sistemas heterogêneos com diferentes donos**. Rio de Janeiro, 2025. 45p. Projeto Final de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho investiga vulnerabilidades de segurança e desafios de governança em uma camada de orquestração de APIs baseadas em GraphQL, que integram múltiplos sistemas. Nesta pesquisa, é utilizado o orquestrador de APIs KAPIO: Knowledge API Orchestrator, que além de atuar como proxy para as APIs conectadas a ele, também possui a capacidade de processar dados provindos de diferentes endpoints, tanto em paralelo quanto de forma sequencial.

A pesquisa aprofunda questões de segurança ao reunir dados de origens distintas, com variados níveis de permissão e sensibilidade, em um ambiente único, destacando riscos associados a arquiteturas do tipo sistema de sistemas (SoS), caracterizadas pela união de sistemas independentes para desempenhar novas funcionalidades.

A metodologia empregada incluiu uma revisão sistemática da literatura sobre segurança em GraphQL e uma análise estática e dinâmica de código da solução original. Como foco do trabalho, busca-se criar um piloto capaz de manter confidencialidade mediante criptografia de dados sensíveis, controle de acesso com permissões granulares por perfil de usuário, permitindo selecionar atributos específicos em cada endpoint e detecção de anomalias e ciclos.

A pesquisa conclui com uma discussão sobre possíveis melhorias e direções futuras, como a adoção de técnicas de mitigação para os riscos de segurança identificados, oferecendo um caminho para a implementação segura de um orquestrador de APIs para integração de múltiplos sistemas em um cenário de SoS.

Palavras-chave

Cibersegurança; Sistemas de Sistemas; Gerenciamento de APIs; GraphQL.

Abstract

Vergolino Gnani, Enrico; Pinheiro de Almeida, Vitor (Advisor); Oliveira da Silva, Anderson (Co-Advisor). **Securing KAPIO: A Knowledge API Orchestrator for multiple heterogeneous systems with different ownerships**. Rio de Janeiro, 2025. 45p. Projeto Final de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

This work investigates security vulnerabilities and governance challenges in a GraphQL-based API orchestration layer that integrates multiple systems. This research uses the KAPIO API orchestrator: Knowledge API Orchestrator, which, in addition to acting as a proxy for the APIs connected to it, also has the ability to process data from different endpoints, both in parallel and sequentially.

The research delves into security issues by bringing together data from different sources, with varying levels of permission and sensitivity, in a single environment, highlighting risks associated with system-of-systems (SoS) architectures, characterized by the union of independent systems to perform new functionalities.

The methodology used included a systematic review of the literature on GraphQL security and a static and dynamic code analysis of the original solution. The work focuses on creating a pilot project capable of maintaining confidentiality through encryption of sensitive data, access control with granular permissions per user profile, allowing the selection of specific attributes at each endpoint, and detection of anomalies and cycles.

The research concludes with a discussion of possible improvements and future directions, such as the adoption of mitigation techniques for the identified security risks, offering a path for the secure implementation of an API orchestrator for the integration of multiple systems in a SoS scenario.

Keywords

Cybersecurity; System of Systems; API Management; GraphQL.

Sumário

Lista de figuras

Lista de Códigos

Lista de Abreviaturas

API – *Application Programming Interface*

SoS – *System of Systems*

DOS – *Denial of Service*

DDOS – *Distributed Denial of Service*

JWT – *JSON Web Token*

SSDLC – *Software Development Lifecycle*

CI/CD – *Continuous Integration / Continuous Delivery*

1
Introdução

As Interfaces de Programação de Aplicações (APIs) representam um elemento fundamental, já consolidado na arquitetura moderna. Um passo para ampliar sua eficácia é a implementação de Sistemas de Sistemas (SoS), oferecendo um mecanismo padronizado para a integração e comunicação entre componentes heterogêneos (INOCÊNCIO; GONZALES; HORITA, 2019). A combinação dessas tecnologias cria oportunidades significativas para organizações que buscam maximizar o valor de seus ativos digitais através da interoperabilidade e da criação de ecossistemas integrados. Um Sistema de Sistemas (SoS) constitui uma categoria de sistema complexo na qual diversos sistemas independentes se unem tendo cada um suas próprias funcionalidades e objetivos, em uma estrutura coesa que gera valor superior ao que seria possível através da operação isolada de cada componente.

Esta abordagem oferece benefícios em termos de agilidade organizacional, redução de custos operacionais e capacidade de inovação. Através da implementação de APIs bem projetadas, as organizações podem acelerar o desenvolvimento de novos produtos e serviços, aproveitando funcionalidades existentes sem a necessidade de reconstruir componentes do zero. Além disso, a arquitetura baseada em SoS permite maior flexibilidade na escolha de fornecedores e tecnologias, evitando o aprisionamento tecnológico e facilitando a evolução gradual dos sistemas empresariais.

1.1
Desafios de um SoS

A criação e manutenção de um Sistema de Sistemas (SoS) apresenta diversos desafios, seja pela integração de dados ou pela integração de interfaces gráficas, que podem ser particularmente complexos quando se trata da unificação de dados provenientes de múltiplas origens com diferentes características

de governança, segurança e estrutura. Estes desafios tornam-se especialmente críticos em ambientes onde dados de diversas empresas e organizações precisam ser integrados em uma única plataforma, mantendo simultaneamente os requisitos específicos de cada fonte.

O principal obstáculo reside na heterogeneidade dos níveis de permissão e sensibilidade dos dados. Diferentes organizações possuem políticas de segurança distintas, marcos regulatórios específicos e classificações variadas para seus ativos de informação. Quando estes dados precisam ser consolidados em um SoS, surge a necessidade de implementar uma arquitetura de segurança que seja capaz de respeitar e aplicar simultaneamente múltiplos conjuntos de regras de acesso, sem comprometer a integridade ou a confidencialidade das informações.

A governança de dados em um ambiente SoS também demanda estruturas organizacionais específicas que possam coordenar as necessidades e requisitos de múltiplas partes interessadas, estabelecendo acordos claros sobre responsabilidades, propriedade dos dados e processos de resolução de disputas. Esta coordenação torna-se ainda mais desafiadora quando as organizações envolvidas possuem culturas corporativas distintas e objetivos nem sempre alinhados.

É importante salientar que esse trabalho foca nos desafios que a parte da orquestração de APIs possui, dessa forma, não será discutido desafios que fujam dessa premissa.

2
Trabalhos relacionados

Para o desenvolvimento deste trabalho, foi conduzida uma revisão sistemática da literatura científica com o objetivo de identificar e analisar as principais contribuições acadêmicas relacionadas à segurança dos Sistemas de Sistemas (SoS) e à integração de APIs em arquiteturas empresariais complexas.

A metodologia adotada utilizou ferramentas tecnológicas avançadas para otimizar o processo de busca e análise bibliográfica. Especificamente, foram empregados o Consensus App (CONSENSUS, 2022) e o NotebookLM (GOOGLE, 2023), duas plataformas que incorporam recursos de inteligência artificial para facilitar a descoberta e síntese de conhecimento científico.

O Consensus App foi utilizado como ferramenta principal para a identificação de artigos científicos relevantes através de consultas baseadas em linguagem natural, palavras-chave e outros filtros, como por exemplo a data do artigo. Esta plataforma permite a realização de buscas por meio de prompts estruturados, possibilitando a formulação de perguntas específicas sobre os temas de interesse e obtendo respostas fundamentadas em evidências científicas. A capacidade do Consensus de processar consultas complexas e fornecer sínteses baseadas em múltiplas fontes acadêmicas contribuiu significativamente para a abrangência da revisão bibliográfica.

Complementarmente, o NotebookLM da Google foi empregado para, além de tentar aumentar o repertório de artigos, aprofundar a análise dos materiais identificados e facilitar a organização do conhecimento coletado. Esta ferramenta demonstrou particular eficácia na capacidade de processar grandes volumes de texto acadêmico e auxiliar na identificação de padrões, tendências e lacunas na literatura existente, como por exemplo na Figura 2.1, onde utilizando vários estudos como base, podemos pesquisar informações de maneira rápida, como na Figura 2.2.

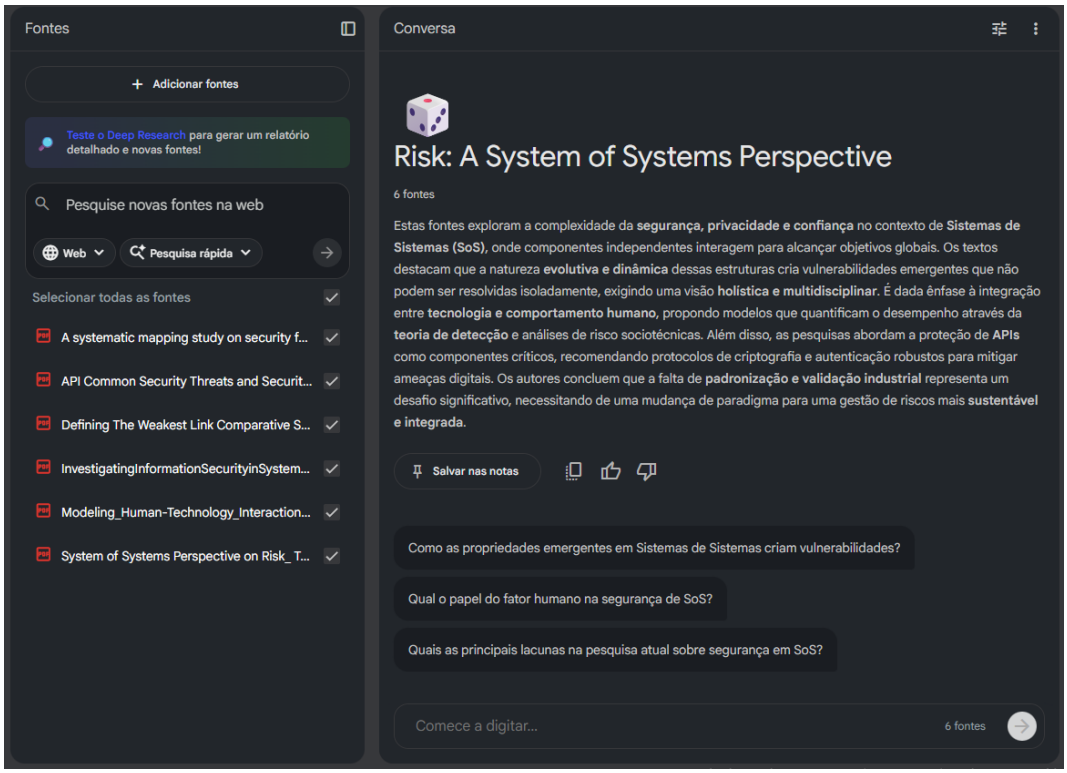

Figura 2.1: Exemplo de uso do NotebookLLM - Adicionar bibliografia

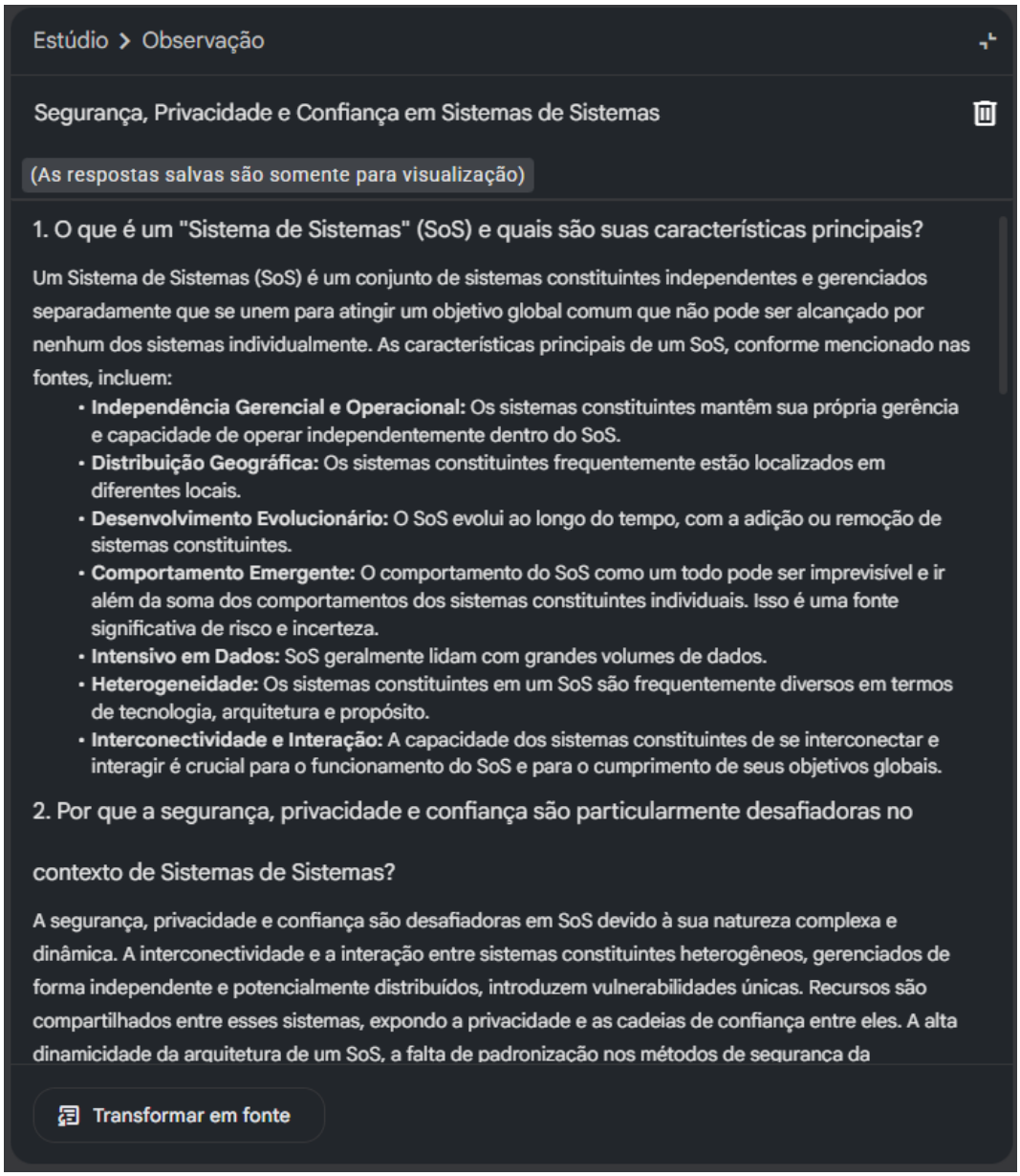

Figura 2.2: Exemplo de uso do NotebookLLM - Pesquisa utilizando a bibliografia adiconada

2.1 Análise da literatura - SoS

Durante a revisão sistemática da literatura foi notado uma grande falta de artigos sobre o tema de segurança aplicado em SoS, e suas possíveis falhas. Como observado na pesquisa apresentada em Dias, Zacarias, Varella, dos Santos (2022) e principalmente pela conclusão do Estudo de Mapeamento Sistemático (SMS) (OLIVERO; BERTOLINO; DOMINGUEZ-MAYO, 2024) que analisou 1828 artigos:

"Although several different papers have been published in the last decade, this SMS shows that little attention has been paid to the challenges imposed by the security in the SoS context where the resources are shared with the aim of achieving a common goal." (OLIVERO; BERTOLINO; DOMINGUEZ-MAYO, 2024)

Os poucos artigos que abordaram o tópico concluíam que a segurança no ambiente digital contemporâneo constitui uma questão complexa e multidimensional que demanda análise integrada com as preocupações relacionadas à privacidade e à confiança. Esta interdependência entre os três elementos representa um desafio fundamental, uma vez que a ausência de composabilidade entre segurança, privacidade e confiança torna significativamente mais difícil a determinação e implementação dessas propriedades no contexto dos Sistemas de Sistemas. (PINTO; MCSHANE; BOZKURT, 2012)

A natureza interconectada desses aspectos significa que não é possível abordar adequadamente questões de segurança sem considerar simultaneamente os impactos na privacidade dos dados e na confiança entre os componentes do sistema. Esta característica apresenta desafios particulares em arquiteturas de Sistemas de Sistemas, onde múltiplos sistemas independentes devem operar de forma coordenada, cada um potencialmente com diferentes padrões e requisitos de segurança, privacidade e confiança. (DIAS et al., 2022)

A dificuldade em estabelecer propriedades compostas de segurança surge justamente da impossibilidade de simplesmente somar ou combinar as características individuais de cada componente para obter o comportamento desejado do sistema como um todo. Esta limitação exige abordagens mais sofisticadas para o design e implementação de soluções que possam garantir simultaneamente a proteção adequada, a preservação da privacidade e a manutenção da confiança em ambientes distribuídos e heterogêneos.

2.2 Análise da literatura - Microserviços

A análise da literatura específica sobre segurança e análise de riscos em SoS evidenciou uma lacuna relevante: há um número reduzido de trabalhos que tratam diretamente desses aspectos considerando as características particulares de SoS, como autonomia dos sistemas constituintes, evolução contínua e interdependências dinâmicas. Diante dessa limitação, tornou-se necessário ampliar o escopo da revisão, incorporando estudos mais gerais da área de segurança de software e de arquiteturas distribuídas, com o objetivo de identificar conceitos, técnicas e abordagens potencialmente aplicáveis ao contexto de SoS.

Nesse sentido, arquiteturas baseadas em microserviços e no uso intensivo de APIs apresentam similaridades estruturais com SoS, especialmente no que se refere à distribuição de responsabilidades, comunicação entre componentes independentes e aumento da superfície de ataque. O estudo de Chatterjee e Prinz (2022) propõe uma solução integrada de segurança para arquiteturas de microserviços, combinando o Spring Security Framework com a plataforma de gerenciamento de identidades Keycloak e o protocolo OAuth2. Os autores demonstram que a adoção conjunta de mecanismos de autenticação, autorização e criptografia contribui para mitigar ameaças comuns em ambientes distribuídos, como ataques de negação de serviço (DDOS), man-in-the-middle (MITM) e acessos não autorizados (CHATTERJEE; PRINZ, 2022).

Embora o trabalho de CHATTERJEE não tenha como foco explícito Sistemas de Sistemas, seus resultados são relevantes para este contexto, pois evidenciam a importância de camadas de segurança integradas e da gestão centralizada de identidades em ambientes compostos por múltiplos sistemas autônomos. Em SoS, onde sistemas constituintes podem pertencer a diferentes organizações e possuir ciclos de vida independentes, tais mecanismos podem auxiliar na redução de riscos associados à interoperabilidade e à confiança entre sistemas.

Complementarmente, a revisão sistemática conduzida por Mousavi et al.

(2025) analisa o problema do uso incorreto de APIs de segurança, destacando que a má utilização dessas interfaces representa uma fonte significativa de vulnerabilidades em sistemas modernos. Os autores identificam diferentes tipos de APIs de segurança, incluindo criptografia, SSL/TLS, OAuth e frameworks como o Spring Security, e classificam diversas formas recorrentes de uso inadequado, que podem comprometer confidencialidade, integridade e autenticação (MOUSAVI et al., 2025).

Os resultados desse estudo são particularmente relevantes para SoS, uma vez que esses sistemas frequentemente dependem da integração de múltiplas APIs desenvolvidas de forma independente. Assim, mesmo quando mecanismos de segurança estão disponíveis, seu uso incorreto por desenvolvedores ou integradores pode introduzir riscos sistêmicos difíceis de detectar. A revisão também evidencia a necessidade de ferramentas e métodos que auxiliem na detecção de usos inadequados de APIs de segurança, aspecto que pode ser explorado como parte de estratégias de mitigação de riscos em SoS.

Dessa forma, ainda que a literatura específica sobre análise de riscos e segurança em Sistemas de Sistemas seja limitada, estudos mais gerais sobre segurança em arquiteturas distribuídas, proteção de APIs e uso correto de frameworks de segurança fornecem fundamentos conceituais e técnicos relevantes. Esses trabalhos contribuem para a compreensão de ameaças, vulnerabilidades e mecanismos de mitigação que podem ser adaptados e estendidos para o contexto de SoS, servindo como base para o desenvolvimento da abordagem proposta neste trabalho.

2.3 Autenticação e Autorização

A autenticação e a autorização desempenham um papel crítico na segurança dos sistemas web, e como essas serão parte principal da implementação nesse projeto, houve uma busca por trabalhos que discutissem a implementação em ambientes web.

A autenticação é o processo responsável por verificar a identidade de um usuário, sistema ou serviço que tenta acessar uma aplicação. Seu objetivo é responder à pergunta “quem é você?”, assegurando que a entidade que solicita acesso é realmente quem afirma ser. Esse processo pode ser realizado por meio de diferentes mecanismos, como credenciais tradicionais (usuário e senha), certificados digitais, tokens, autenticação multifator (MFA) ou protocolos modernos baseados em identidade, como OAuth 2.0 e OpenID Connect. Em sistemas computacionais, a autenticação constitui a primeira camada de segurança, pois somente após a confirmação da identidade é que o sistema pode aplicar políticas de acesso adequadas aos recursos disponíveis.(PYROH; TERESHCHUK; TOROSHANKO, 2025)

A autorização, por sua vez, ocorre após a autenticação e tem como finalidade determinar o que um usuário autenticado está autorizado a fazer dentro do sistema. Esse processo define quais recursos podem ser acessados e quais ações podem ser executadas, com base em regras e políticas de segurança previamente estabelecidas. A autorização pode ser implementada de forma simples, por meio de perfis ou papéis (roles), ou de maneira mais avançada, utilizando autorização granular, na qual o controle de acesso é definido em níveis mais detalhados, como operações específicas, atributos do usuário, contexto da requisição ou dados sensíveis individuais. A autorização granular é especialmente importante em sistemas complexos e distribuídos, pois permite maior flexibilidade, princípio do menor privilégio e um controle mais preciso sobre o acesso às informações, reduzindo riscos de uso indevido ou exposição não autorizada de dados.(THOTA; UDAYARAJU; KUMARI, 2025)

3
Objetivos

O principal objetivo deste trabalho é verificar possíveis falhas de segurança e, caso possível, implementar contramedidas para um Sistema de Sistemas (SoS) que teve seu desenvolvimento iniciado a partir do projeto de conclusão de curso do ex-aluno de graduação Eduardo Dantas Luna (LUNA, 2024), e gerar um piloto seguro da aplicação.

Este trabalho propõe então:

- Analisar possíveis falhas de segurança no sistema atual;
- Focar principalmente na implementação de métodos de autenticação, autorização e detecção de ciclos;
- Desenvolver um piloto com configurações presentes para tentar mitigar as falhas encontradas.

4
Atividades Realizadas

4.1
Estudos Preliminares

O aluno não tinha prévia experiência com segurança da informação. Para suprir essa deficiência foi empreendido tempo na disciplina INF1416 - Segurança da Informação da PUC-Rio, do co-orientador Prof. Anderson Oliveira da Silva, onde foi visto, entre outras coisas, os fundamentos de segurança da informação e os mecanismos de segurança para implantar os cinco pilares da segurança da informação: (i) controle de integridade; (ii) controle de autenticidade; (iii) controle de confidencialidade (sigilo); (iv) controle de acesso; e (v) controle de disponibilidade.

Nessa disiplina, nosso principal objetivo foi aplicarmos os pilares da segurança da informação no desenvolvimento de software seguro. Para isso, foi estudada a arquitetura de criptografia Java na prática para desenvolvimento de programas Java seguros e em conformidade com as normas de segurança. Além disso, vimos também segurança ofensiva, onde aprendemos sobre as ameaças aos sistema e riscos de segurança, e segurança defensiva, que por sua vez, aborda as técnicas usadas para combater as ameaças.

Esses estudos preliminares foram fundamentais para fornecer a base conceitual necessária à análise de riscos e à proposição de soluções de segurança no contexto deste trabalho, especialmente considerando a complexidade inerente a arquiteturas distribuídas e Sistemas de Sistemas.

4.2
Estudos da Tecnologia

Após a consolidação dos fundamentos teóricos de segurança da informação, iniciou-se o estudo das tecnologias que seriam utilizadas no desenvolvimento da solução proposta. Inicialmente, foi realizado um estudo sobre o

GraphQL, com o objetivo de compreender seu modelo de consulta, suas vantagens em relação a APIs REST tradicionais e os impactos de seu uso em arquiteturas distribuídas.

Para o projeto, optou-se pela utilização do *framework Netflix DGS* (Netflix, 2024) integrado ao *Spring Boot* (VMware, Inc., 2024). Esses frameworks são caracterizados pelo suporte nativo a funcionalidades relevantes para arquiteturas distribuídas, tais como *data loaders*, controle de profundidade de consultas e facilidade de integração com mecanismos de segurança providos pelo *Spring Security.*(SPRING FRAMEWORK, 2025)

O Spring Security é um framework do ecossistema Spring responsável por prover mecanismos robustos e extensíveis de segurança para aplicações Java, especialmente aplicações web e APIs REST. Ele atua principalmente no controle de autenticação (verificação da identidade do usuário) e autorização (definição do que cada usuário pode acessar), integrando-se de forma transparente ao Spring Boot. Por meio de configurações baseadas em código ou anotações, é possível definir regras de acesso a endpoints, proteger recursos sensíveis, configurar autenticação baseada em sessões, tokens JWT ou OAuth2, além de aplicar filtros de segurança na cadeia de requisições HTTP. Dessa forma, o Spring Security permite que a segurança da aplicação seja centralizada, flexível e alinhada às boas práticas de desenvolvimento seguro, reduzindo riscos e aumentando a confiabilidade do sistema. (SPRING FRAMEWORK, 2025)

Além disso, o DGS é amplamente utilizado em ambientes industriais de grande escala, como na própria Netflix, onde é empregado para integrar centenas de serviços de backend por meio de um gateway central baseado em GraphQL. Essa característica o torna especialmente adequado ao contexto de Sistemas de Sistemas, nos quais diferentes sistemas autônomos precisam ser integrados de forma escalável, segura e evolutiva.

Os estudos realizados nessa etapa permitiram não apenas a seleção das tecnologias mais adequadas ao escopo do trabalho, mas também embasaram

as decisões arquiteturais adotadas, especialmente no que se refere à aplicação de mecanismos de segurança com o Spring Security.

5
Implementações

5.1
Desenvolvimento do Sistema

O SoS escolhido e já desenvolvido previamente como alvo desse projeto pode ser representado em três camadas: a camada de microsserviços, a camada externa e o KAPIO. A camada externa é formada por diferentes usuários em sistemas que requisitam dados originados dos microsserviços por meio da camada do orquestrador KAPIO, que por sua vez se comunica diretamente com a camada externa, como representado na Figura 5.1.

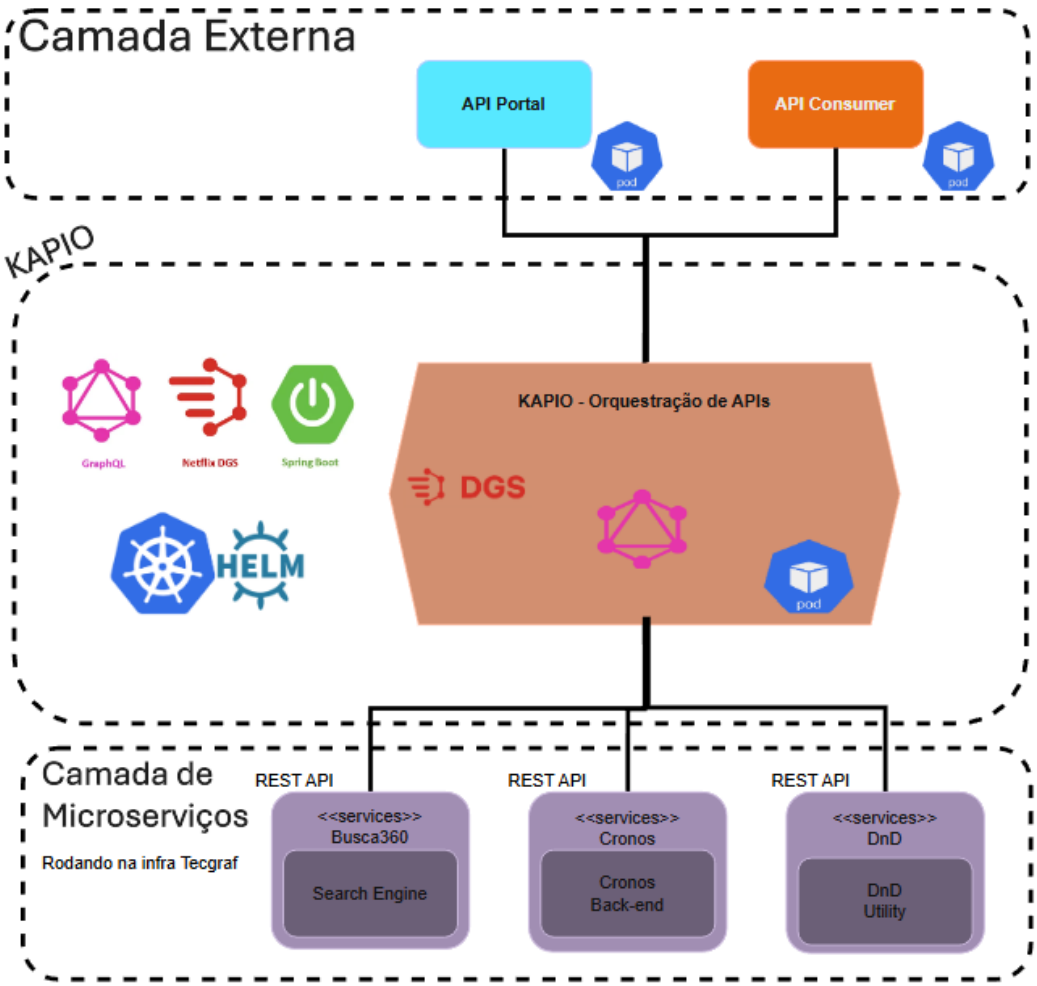

Figura 5.1: Modelo KAPIO - Diagrama da infraestrutura inicial

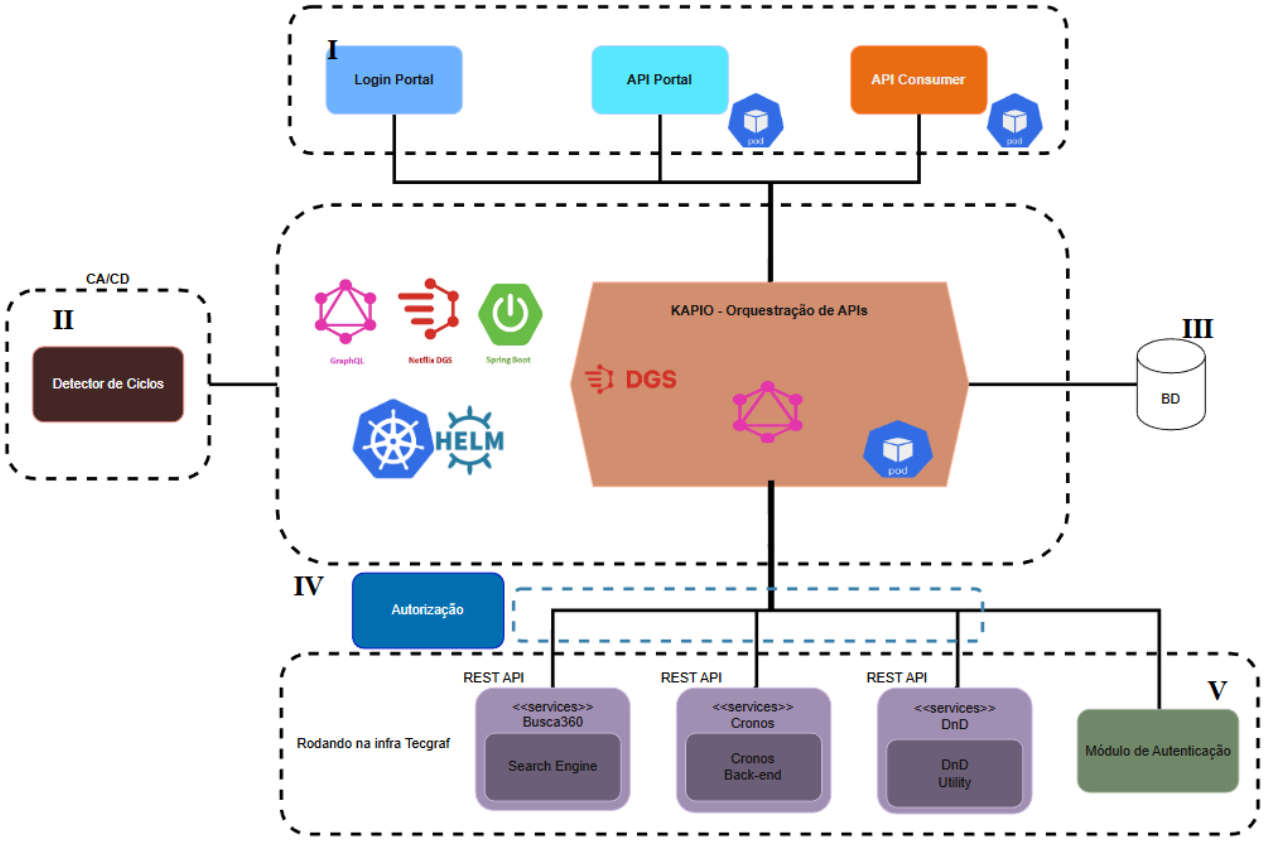

Figura 5.2: Modelo KAPIO - Diagrama da infraestrutura nova

Durante esse projeto, foram desenvolvidos diversas medidas de segurança que alteraram a infraestrutura original prévia do SoS, sendo elas, uma tela de login(I), um detector de ciclos(II), um banco de dados(III), um sistema de autorização(IV) e um módulo de autenticação(V), criando assim a infraestrutura vista na Figura 5.2. Cada uma dessas alterações feitas serão explicadas e justificadas na Seção 5.2.

5.2 Avaliação de Risco

Em um SoS, no qual múltiplos sistemas autônomos são integrados por meio de um orquestrador de APIs implementado com o framework Netflix DGS, a centralização do acesso aos serviços traz benefícios de interoperabilidade e abstração, mas também introduz riscos relevantes do ponto de vista de segurança, desempenho e disponibilidade. Nesta seção, são analisados potenciais casos de mau uso, seguidos de propostas de solução e considerações de

implementação no contexto da arquitetura adotada.

Inicialmente, para realizarmos a avaliação de risco do sistema, foram consideradas a utilização de ferramentas matemáticas e estatísticas, para buscar atribuir valores específicos aos custos das medidas de segurança e à quantidade de danos que podem ocorrer (LO; CHEN, 2012). Porém, como esses métodos exigem um extenso trabalho preliminar para coletar valores precisos de todos os elementos, incluindo o valor dos ativos, a frequência das ameaças, a eficácia das medidas de segurança e seus custos, essa ideia foi descartada, uma vez que a pesquisa por trás do KAPIO é ainda muito nova, não tendo uma ideia muito precisa dos requisitos e limitações do cliente.

Dessa maneira, foi realizada uma avaliação de risco básica, com os conceitos aprendidos nas aulas do co-orientador Prof. Anderson Oliveira da Silva e conceitos, onde focamos principalmente nos pilares de controle de autenticidade e controle de acesso.

5.2.1
Autenticação inadequada ou inexistente

No nosso cenário, o KAPIO atua como ponto único de entrada para múltiplos sistemas constituintes. A ausência de mecanismos robustos de autenticação ou a adoção de autenticação fraca pode permitir que usuários não autorizados acessem o gateway GraphQL, explorando consultas para obter dados sensíveis de diferentes sistemas integrados. Esse risco é amplificado na nossa arquitetura, pois uma única consulta pode agregar dados de múltiplas fontes.

5.2.1.1
Proposta de solução

A contramedida escolhida para mitigar essa vunerabilidade foi a criação de um sistema de login, garantindo que apenas clientes autenticados possam realizar consultas ao gateway.

5.2.1.2 Implementação

Para implementar o sistema de login foi necessário a criação de três principais componentes: Um banco de dados para guardar os usuários e seus dados de login; uma tela de login; um componente que conecta os dois, grarantindo a autenticação.

O banco de dados foi implementado utilizando SQLite (HIPP, 2025), representado pelo símbolo com um III na Figura 5.2. No banco de dados de usuários é esperado três campos por usuário, o username usado para login, o hash da senha do usuário e o cargo do usuário, que será mais importante na Seção 5.2.2.

É importante ressaltar que não foi implementado um sistema de cadastro de usuários diretamente pelo sistema web. Para testar a aplicação, o banco de dados de usuários foi preenchido previamente, utilizando um script Java.

A segurança das senhas é um aspecto crucial do sistema que criamos. Quando um novo usuário é criado através do método createUser no CustomUserDetailsService, a senha em texto plano é imediatamente transformada em um hash BCrypt antes de ser salva no banco de dados, como no código abaixo. O BCrypt é um algoritmo especialmente projetado para ser lento e custoso computacionalmente, o que o torna extremamente resistente a ataques de força bruta. (SPRING FRAMEWORK, 2025)

Código 1: Função utilizada para criar um usuário no BD

```
1 public User createUser(String username, String rawPassword) {
2     if (userRepository.existsByUsername(username)) {
3         throw new RuntimeException("UsuÃ¡rio jÃ¡ existe: " +
              username);
4     }
5
6     User user = new User();
7     user.setUsername(username);
8     // Encripta a senha usando BCrypt antes de salvar
```

```
9       user.setPassword(passwordEncoder.encode(rawPassword));
10      user.setEnabled(true);
11      user.setRole("ROLE_USER");
12
13      return userRepository.save(user);
14  }
```

O BCrypt também adiciona automaticamente um "salt"único para cada senha, o que significa que mesmo que dois usuários tenham a mesma senha, os hashes armazenados no banco serão completamente diferentes. Este salt previne ataques usando tabelas rainbow e adiciona uma camada extra de segurança. (SPRING FRAMEWORK, 2025)

Durante o processo de login, quando o Spring Security precisa validar as credenciais, método do módulo representado com um V na Figura 5.2, ele usa o mesmo algoritmo BCrypt para comparar a senha fornecida com o hash armazenado, configurado no código abaixo.

Código 2: Função de configuração da autenticação

```
1  /**
2  * AuthenticationManager - gerencia o processo de autenticaÃ§Ã£o.
3  * Conecta nosso UserDetailsService customizado com o encoder de
       senha.
4  */
5  @Bean
6  public AuthenticationManager authenticationManager(HttpSecurity http
       ) throws Exception {
7       AuthenticationManagerBuilder authBuilder = http.getSharedObject(
            AuthenticationManagerBuilder.class);
8
9       authBuilder
10          .userDetailsService(userDetailsService)
11          .passwordEncoder(BCryptPasswordEncoder());
12
13      return authBuilder.build();
14  }
```

Dessa forma, utilizando o Spring Security para configurar um autenticação automática, comparando o hash gerado na função BCrypt com a senha digitada, e os dados do usuário resgatados do banco de dados a partir da função contida em userDetailsService onde buscamos o username digitado pelo usuário no banco de dados.

Código 3: Função de busca por username

```
1 @Override
2 public UserDetails loadUserByUsername(String username) throws
      UsernameNotFoundException {
3     User user = userRepository.findByUsername(username)
4             .orElseThrow(() -> new UsernameNotFoundException(
5                     "UsuÃ¡rio nÃ£o encontrado: " + username));
6
7     // Converte nossa entidade User para o UserDetails do Spring
          Security
8     return org.springframework.security.core.userdetails.User.
          builder()
9             .username(user.getUsername())
10            .password(user.getPassword()) // JÃ¡ estÃ¡ encriptada no
                  banco
11            .authorities(Collections.singleton(new
                  SimpleGrantedAuthority(user.getRole())))
12            .accountExpired(false)
13            .accountLocked(false)
14            .credentialsExpired(false)
15            .disabled(!user.isEnabled())
16            .build();
17 }
```

Importante notar que nunca é possível "descriptografar"um hash BCrypt de volta para a senha original, a comparação é feita gerando um novo hash da senha fornecida e verificando se ele corresponde ao hash armazenado.

Concluindo, o processo de autenticação segue este fluxo: primeiro, o usuário preenche o formulário de login com suas credenciais. Em seguida, o Spring Security pega essas credenciais e as passa para o nosso CustomUserDetails-

Service, que busca o usuário no banco de dados SQLite. A senha informada é então comparada com o hash BCrypt armazenado no banco. Se as credenciais estiverem corretas, uma sessão é criada e o usuário é redirecionado para a página principal de query no KAPIO como na Figura 5.3.

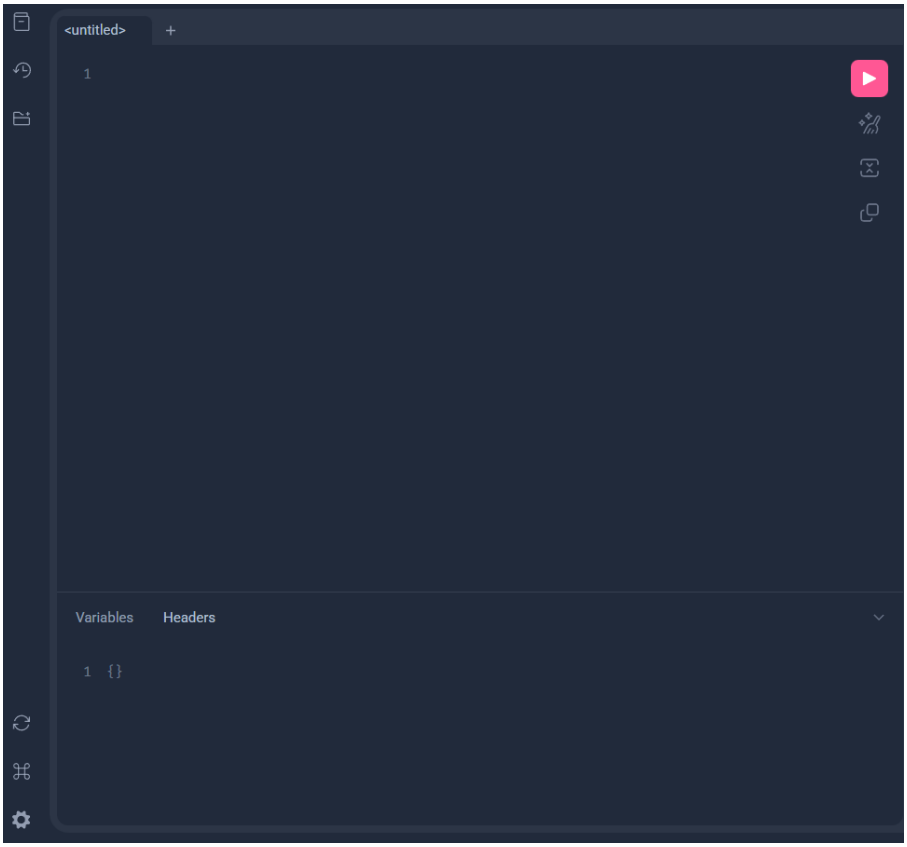

Figura 5.3: Tela principal de queries

Caso contrário, ele volta para a página de login, representado pelo símbolo com um I na Figura 5.2, com uma mensagem de erro, como na Figura 5.4.

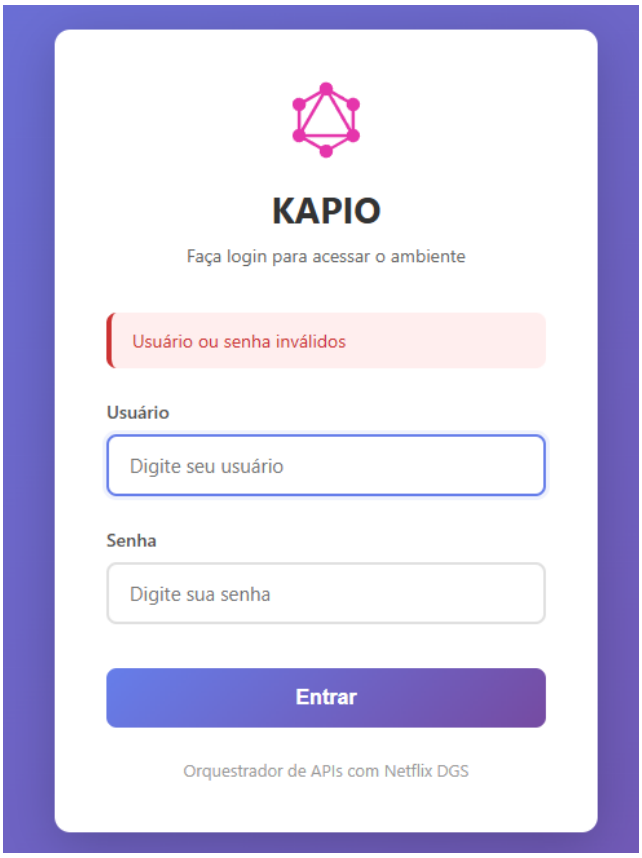

Figura 5.4: Tela de login com erro de autentincação

5.2.2 Falhas de autorização

Mesmo com autenticação adequada, um usuário pode tentar acessar dados ou operações para os quais não possui permissão. Em um SoS, diferentes sistemas podem possuir políticas de acesso distintas, e a falta de controle de autorização no KAPIO pode resultar em exposição indevida de informações ou execução de operações críticas.

5.2.2.1 Proposta de solução

Para mitigar esse risco, o GraphQL possui uma característica única que o torna especialmente interessante para controle de acesso granular. Diferente de APIs REST tradicionais, onde cada endpoint retorna um conjunto fixo de dados, no GraphQL você tem uma flexibilidade natural que pode ser aproveitada para implementar segurança no nível de campo.

Dessa forma, utilizando o GraphQL é possível a criação de diferentes perfis de acesso para diferentes tipos de usuário onde cada um desses perfis deve ter acesso a diferentes informações do sistema. Assim, a maneira adotada para adicionar essa solução de autorização foi por meio da implementação de duas camadas:

Primeira camada - Autorização no nível de tipo: Nesta camada você decide se o usuário tem permissão para acessar um determinado tipo de recurso. Por exemplo, "usuários do tipo 'ADMIN' podem acessar a query 'buscaPalavraChave'?".

Segunda camada - Autorização no nível de campo: Implementação do controle granular. Mesmo que o usuário possa acessar um tipo de recurso, você controla campo por campo o que ele pode ver.

5.2.2.2 Implementação

Utilizando o valor de cargo do usuário salvo no banco de dados no momento de cadastro, podemos implementar as camadas de autorização. Os cargos são agrupamentos convenientes de permissões que correspondem a funções no sistema. Por exemplo, você pode ter roles como ADMIN, DEVELOPER, ANALYST, VIEWER. Cada cargo tem um conjunto de permissões associadas. Permissões são strings que descrevem ações específicas que podem ser realizadas, como "BUSCA360READ", que permite o usuário utilizar a query buscaPalavraChave do sistema Busca360.

O passo mais crítico foi criar e adaptar o PermissionDataLoader para nossas necessidades. Esse componente tem como principal função, definir quais permissões que existem, como no Código 4, e quais cargos terão quais permissões, como sendo definido no Código 5.

Código 4: Exemplo de criação de permissões

```
1 private void createPermissionsIfNotExist(){
2     createPermission(
```

```
        "plataforma:cronos:calculateRti",
        "Permite fazer a query calculateRti"
    );
    createPermission(
        "plataforma:cronos:rti:data",
        "Permite ler os dados de RTI_PRE da calculateRti"
    );
    createPermission(
        "plataforma:cronos:rti:index",
        "Permite ler o dado indice_RTI_pre da calculateRti"
    );

    ...
}
```

Código 5: Exemplo de criação de cargos/roles

```
Role rti_viewer = createRole(
    "RTI-VIEWER",
    "UsuÃ¡rio com acesso somente leitura a todas as plataformas"
);
if (viewer != null) {
    addPermissionToRole(viewer, "plataforma:cronos:read");
    addPermissionToRole(viewer, "plataforma:dnd:read");
    addPermissionToRole(viewer, "plataforma:busca360:read");
    roleRepository.save(viewer);
}
```

A primeira camada, a autorização no nível de tipo, protege queries e mutations inteiras. Quando você coloca @RequiresPermission("BUSCA360READ") em uma query, como no código 6, está dizendo ao sistema "ninguém executa esta query sem esta permissão específica". É simples, declarativo, e extremamente efetivo. Se um usuário tenta executar a query sem a permissão, ele nem chega perto dos dados - recebe um erro claro de acesso negado imediatamente, como na Figura 5.5.

Código 6: Exemplo de autorização de tipo

```
@DgsQuery(field = "buscaPalavraChave")
@RequiresPermission("BUSCA360READ")
public Mono<String> buscaPalavraChave(@InputArgument("palavrasChave"
    ) String palavrasChave, DgsDataFetchingEnvironment dfe)
{
...
}
```

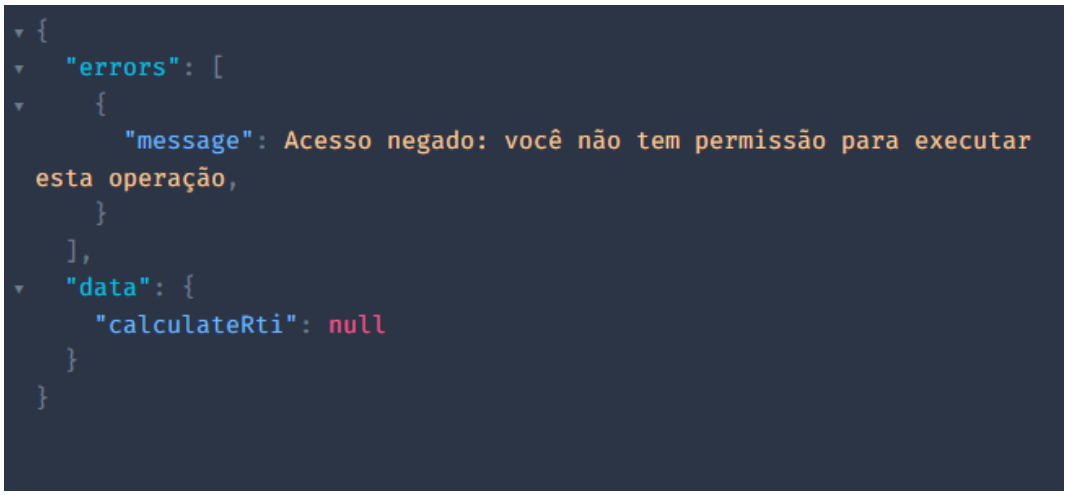

Figura 5.5: Exemplo de erro na autorização de uma query

A segunda camada, a autorização no nível de campo permite que você controle, campo por campo, o que cada usuário vê nos resultados. Imagine que é necessário limitar quem pode ler um campo "HP"vindo da query "monster". Com este sistema, você implementa isso simplesmente adicionando @RequiresFieldPermission nos resolvers de campo apropriados. O mesmo endpoint GraphQL serve adequadamente todos esses diferentes tipos de usuários, cada um recebendo exatamente os dados que tem permissão de ver, como demonstrado na Figura 5.6, onde não apenas um erro é levantado, informando o usuário, como também o campo negado é retornado como nulo.

Código 7: Exemplo de autorização granular

```
@DgsQuery(field = "monster")
@RequiresFieldPermission(field:dnd:hp)
public Mono<Monster> monster(@InputArgument String name,
    DgsDataFetchingEnvironment dfe)
```

```
4 {
5 ...
6 }
```

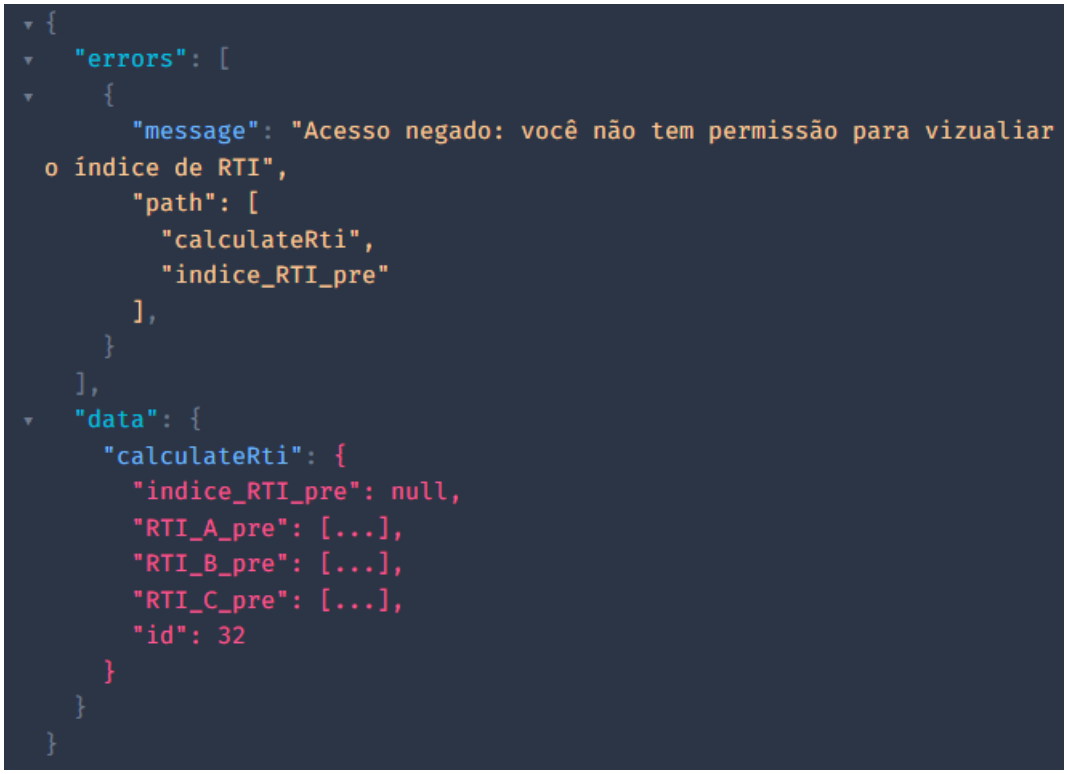

Figura 5.6: Exemplo de erro na autorização granular de uma query

5.2.3 Problema de ciclos

Embora o GraphQL se prevenha automaticamente contra queries cíclicas em um sistema, onde por exemplo uma query A chama uma query B que chama A novamente (MICROSOFT, 2025), um SoS tem um comportamento anormal. No SoS, onde além das queries de um único sistema existem outros sistemas com suas queries correspondentes, pode haver uma dependência cíclia entre queries de diferentes sistemas que passaria sem ser notada pelo GraphQL. Essa dependência aconteceria, por exemplo, no caso de duas queries cujos atributos GraphQL não pussuem dependência, mas os serviços utilizados para gerar esses atributos sim.

Dessa forma, um cliente malicioso ou mal configurado pode explorar a estrutura do GraphQL para gerar consultas que resultem em múltiplas chamadas recursivas. Em um SoS, esse comportamento pode causar degradação significativa de desempenho e sobrecarga dos sistemas integrados.

5.2.3.1 Proposta de solução

Criar código para ser utilizado futuramente como linter, durante o CI/CD. Que detecta um ciclo de dependências entre serviços de sistemas diferentes, ao ler os arquivos de serviço.

5.2.3.2 Implementação

Para a implementação do linter, representado na Figura 5.2 com um número II, utilizamos da característica da arquitetura do código onde todos os *DataFetchers* - aqueles que buscam os dados - de um dado sistema estão divididos entre as pastas *"atomicServices"* ou *"compositeServices"* que está em uma pasta com o nome do sistema.

Primeiramente, o código Python passa por todos os arquivos de serviço de cada sistema e salva quais sistemas utilizam que serviços nos seus *DataFetchers*, em um dicionário python com as chaves sendo os nomes dos sistemas e o valor uma lista dos serviços necessários.

Para descobrir quais serviços são utilizados em um *DataFetchers*, procura-se quais serviços estão sendo importados, a partir de um regex "import+.*?resolver(.*?Service)?;".

Código 8: Criação do dicionário de dependências

```
found_dependencies: dict = {}

for system_folder in SYSTEMS_PATH.iterdir():
    if not system_folder.is_dir():
        continue

```

```
7       found_dependencies[system_folder.name] = []
8
9       for service_folder_name in SERVICE_FOLDERS:
10          service_folder = system_folder / service_folder_name
11
12          if service_folder.is_dir():
13              for file_path in service_folder.iterdir():
14                  if file_path.is_file():
15
16                      try:
17                          with open(file_path, 'r', encoding='utf-8')
                                as f:
18                              for line in f:
19                                  service_match = SERVICE_PATTERN.
                                        search(line)
20                                  if service_match:
21                                      found_dependencies[system_folder
                                            .name].append(service_match.
                                            group(1))
22                                      continue
23
24                      except Exception as e:
25                          raise Exception(f"Erro ao ler o arquivo {
                                file_path.name}: {e}")
```

Após o dicionário ser populado com todas as dependências de todos os sistemas, ele é percorrido, e caso haja uma dependência comum entre sistemas, um aviso é preparado e disparado após todas os sistemas terem sido checados.

Código 9: Detecção de possíveis ciclos

```
1 lst_of_dependencies = []
2 warning = ""
3 for system, dependencies in found_dependencies.items():
4     for dependency in dependencies:
5         if dependency in lst_of_dependencies:
6             warning += f"PossÃvel ciclo detectado: O serviÃ§o '{
                  dependency}' aparece em dois ou mais sistemas.\n"
7         lst_of_dependencies.append(dependency)
```

```
8 if warning:
9     raise Exception(warning)
```

É importante notar que essa implementação busca diminuir ao máximo a chance de falsos negativos durante a detecção, assim deixando possível a detecção de falsos positivos; um linter preciso o bastante para não ser necessário permitir falsos positivos, é possível mas foi um desafio grande demais para o escopo do projeto. Dessa maneira, foi entendido que permitir falsos positivos atrapalha menos a aplicação, ao comparar com falsos negativos, que permitiriam uma vulnerabilidade no sistema.

5.2.4 Query Depth

Uma característica do GraphQL é a possibilidade de criar consultas altamente aninhadas. Um usuário pode explorar essa funcionalidade para criar consultas com profundidade excessiva, resultando em alto consumo de recursos computacionais e possíveis ataques de negação de serviço (DDOS / DOS).

5.2.4.1 Proposta de solução

Essa vulnerabilidade pode ser mitigada ao simplesmente adicionar um limite à profundidade de um query.

5.2.4.2 Implementação

Novamente utilizando o recurso do Spring Security de configurar detalhes do funcionamento do sistema, podemos implementar esse limite.

Código 10: Limite de profundidade de Query

```
1 @Bean
2 public Instrumentation maxQueryDepthInstrumentation() {
3     // Define o limite mÃ¡ximo de profundidade em 10 nÃveis
4     // Este eh um valor equilibrado que permite queries complexas
          razoÃ¡veis
```

```
5       // mas previne ataques de profundidade excessiva
6       int maxDepth = 10;
7
8       return new MaxQueryDepthInstrumentation(maxDepth, (depth, ctx)
            -> {
9           // Este callback eh executado quando uma query excede o
                limite
10          String queryName = ctx.getOperationDefinition().getName() !=
                null
11              ? ctx.getOperationDefinition().getName()
12              : "anonymous";
13
14          System.err.println(String.format(
15              "Query '%s' foi bloqueada por exceder profundidade
                    mÃ¡xima. " +
16              "Profundidade detectada: %d, Limite: %d",
17              queryName, depth, maxDepth
18          ));
19
20          // Retorna true para continuar com a validaÃ§Ã£o
21          return true;
22      });
23  }
```

5.2.5
Ausência de Rate Limiting e Throttling

Sem mecanismos de limitação de requisições, um cliente pode enviar um grande volume de consultas ao gateway GraphQL, seja de forma intencional ou acidental, comprometendo a disponibilidade dos sistemas constituintes.

5.2.5.1
Proposta de solução

Utilizar a característica de configuração do Spring Security, para criar um método de limitação de requisições.

5.2.5.2 Implementação

Implementado através do RateLimitFilter, onde um máximo de requisições de entrada por segundo pode ser configurado, este filtro intercepta todas as requisições HTTP antes de qualquer outro processamento acontecer. Ele funciona como um porteiro que conta quantas vezes cada pessoa já entrou no prédio hoje e nega entrada para quem já excedeu o limite permitido. Esta proteção é eficaz contra ataques de volume bruto, onde um atacante tenta sobrecarregar o servidor simplesmente enviando muitas requisições, independentemente do conteúdo delas.

Código 11: RateLimitFilter

```
1  @Component
2  public class RateLimitFilter extends OncePerRequestFilter {
3
4      private final RateLimiterService rateLimiterService;
5
6      public RateLimitFilter(RateLimiterService rateLimiterService) {
7          this.rateLimiterService = rateLimiterService;
8      }
9
10     @Override
11     protected void doFilterInternal(
12             HttpServletRequest request,
13             HttpServletResponse response,
14             FilterChain filterChain) throws ServletException,
                   IOException {
15
16         // Identifica o cliente que estÃ¡ fazendo a requisiÃ§Ã£o
17         String clientIdentifier = getClientIdentifier(request);
18
19         // Verifica se o usuÃ¡rio estÃ¡ autenticado
20         // Isso afeta os limites aplicados (autenticados tÃªm
               limites mais generosos)
21         boolean isAuthenticated = isAuthenticated();
22
```

```
23          // Verifica se a requisiÃ§Ã£o deve ser permitida
24          boolean allowed = rateLimiterService.allowRequest(
                clientIdentifier, isAuthenticated);
25
26          if (allowed) {
27              // RequisiÃ§Ã£o permitida - adiciona headers
                    informativos e continua
28              addRateLimitHeaders(response, clientIdentifier);
29              filterChain.doFilter(request, response);  // Passa a
                    requisiÃ§Ã£o adiante no pipeline
30          } else {
31              // RequisiÃ§Ã£o bloqueada - retorna erro 429
32              handleRateLimitExceeded(response, clientIdentifier);
33          }
34      }
```

6
Conclusão e trabalhos futuros

Esta avaliação de riscos demonstra que uma arquitetura de Systems of Systems (SoS), embora poderosa e altamente flexível, exige a adoção de uma abordagem de defesa em profundidade para garantir níveis adequados de segurança. As implementações propostas ao longo deste trabalho contemplam os vetores de ataque básicos e desafios operacionais, estabelecendo múltiplas camadas de proteção que atuam de forma complementar e integrada, conforme o estado da arte observado.

A combinação de mecanismos de autenticação, autorização granular, proteção contra abuso de recursos e monitoramento contínuo constitui a base para a construção de um sistema GraphQL seguro e resiliente. Nesse contexto, o trabalho alcançou seu objetivo ao desenvolver um piloto com características de segurança básicas compatíveis com os requisitos inicialmente propostos, demonstrando a viabilidade da solução apresentada.

Ressalta-se, contudo, que a eficácia dessas medidas depende de sua implementação integrada, da realização de testes periódicos e da capacidade de evolução contínua frente ao surgimento de novas ameaças. Além disso, para trabalhos futuros será importante propor e implementar medidas de contra falhas de segurança que não foram abordadas nesse estudo, como por exemplo o roubo de credencial de acesso, ataques de força bruta e a interceptação com modificação de mensagem. Para guiar os próximos passos, é de suma importante a adoção e avaliação do OWASP Top 10 (OWASP Foundation, 2025).

É importante destacar que existem abordagens alternativas para a implementação de controles de segurança, como o uso de API Management (APIM) e outras metodologias, as quais podem ser exploradas e comparadas também em trabalhos futuros.

Embora o framework Spring Security tenha sido amplamente utilizado durante o desenvolvimento do projeto, sua robustez e abrangência indicam a necessidade de estudos mais aprofundados para explorar todo o seu potencial. Por fim, destaca-se que as medidas de segurança adotadas neste relatório seguiram os requisitos e diretrizes vigentes do sistema analisado; entretanto, a constante verificação e reavaliação dos requisitos do cliente é essencial para assegurar que as estratégias de segurança permaneçam adequadas e eficazes ao longo do tempo.

7
Referências bibliográficas

INOCÊNCIO, T. J.; GONZALES, G. R.; HORITA, F. E. A. Pasos: Processo para definição da arquitetura de sistemas-de-sistemas. In: **Anais do Brazilian Workshop on Large-Scale Critical Systems (BWARE)**. Salvador: Sociedade Brasileira de Computação, 2019. p. 25–28. Citado na página 11.

CONSENSUS. **Consensus App**. 2022. Acesso em: 3 abr. 2025. Disponível em: https://consensus.app/. Citado na página 13.

GOOGLE. **NotebookLM**. 2023. Acesso em: 20 abr. 2025. Disponível em: https://notebooklm.google.com/. Citado na página 13.

OLIVERO, M. A.; BERTOLINO, A.; DOMINGUEZ-MAYO, F. J. e. a. A systematic mapping study on security for systems of systems. **International Journal of Information Security**, v. 23, p. 787–817, 2024. Citado 2 vezes nas páginas 15 e 16.

PINTO, C. A.; MCSHANE, M. K.; BOZKURT, I. **System of Systems Perspective on Risk: Towards a Unified Concept**. 2012. Finance Faculty Publications. Disponível em: https://digitalcommons.odu.edu/finance_facpubs/1. Citado na página 16.

DIAS, R. M. et al. Investigating information security in systems-of-systems. In: **Proceedings of the XVIII Brazilian Symposium on Information Systems**. New York, NY, USA: Association for Computing Machinery, 2022. (SBSI '22). ISBN 9781450396981. Disponível em: https://doi.org/10.1145/3535511.3535523. Citado na página 16.

CHATTERJEE, A.; PRINZ, A. Applying spring security framework with keycloak-based oauth2 to protect microservice architecture apis: A case study. **Sensors**, v. 22, n. 5, 2022. ISSN 1424-8220. Disponível em: https://www.mdpi.com/1424-8220/22/5/1703. Citado na página 17.

MOUSAVI, Z. et al. Detecting misuse of security apis: A systematic review. **ACM Computing Surveys**, Association for Computing Machinery, v. 1, n. 1, p. 1–37, 2025. Preprint disponível no arXiv:2306.08869v4. Disponível em: https://arxiv.org/abs/2306.08869. Citado na página 18.

PYROH, M.; TERESHCHUK, G.; TOROSHANKO, O. Authentication principles as security aspects of web development. **MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES**, 2025. Citado na página 19.

THOTA, S.; UDAYARAJU, P.; KUMARI, K. A. Implementing a two-stage authentication and authorization protocol for improving the user level security in cloud applications. **2025 Third International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)**, p. 498–507, 2025. Citado na página 19.

LUNA, E. D. **Towards Building Digital Twins Based in Systems of Systems**. Monografia (Trabalho de Conclusão de Curso (Graduação em Ciência da Computação)) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, nov. 2024. Orientador: Prof. Vitor Pinheiro de Almeida. Citado na página 20.

Netflix. **Domain Graph Service (DGS) Framework**. 2024. https://netflix.github.io/dgs/. Acesso em: 15 dez. 2025. Citado na página 22.

VMware, Inc. **Spring Boot**. 2024. https://spring.io/projects/spring-boot. Acesso em: 15 dez. 2025. Citado na página 22.

SPRING FRAMEWORK. **Spring Security**. 2025. Página oficial do projeto. Acesso em: 15 dez. 2025. Disponível em: https://spring.io/projects/spring-security. Citado na página 22.

LO, C.-C.; CHEN, W.-J. A hybrid information security risk assessment procedure considering interdependences between controls. **Expert Systems with Applications**, v. 39, n. 1, p. 247–257, 2012. ISSN 0957-4174. Disponível em: https://www.sciencedirect.com/science/article/pii/S0957417411009778. Citado na página 26.

HIPP, R. D. **SQLite**. 2025. Disponível em: https://www.sqlite.org/index.html. Citado na página 27.

SPRING FRAMEWORK. **BCrypt**. 2025. Documentação da API. Acesso em: 15 dez. 2025. Disponível em: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCrypt.html. Citado 2 vezes nas páginas 27 e 28.

MICROSOFT. **learn.microsoft: GraphQL APIs overview**. 2025. Acesso em: 12 abr. 2025. Disponível em: https://learn.microsoft.com/en-us/azure/api-management/graphql-apis-overview. Citado na página 35.

OWASP Foundation. **OWASP Top 10:2025**. 2025. Disponível em: https://owasp.org/Top10/2025/. Citado na página 42.